Maciej Borowiecki

Informatyka na czasie 2

Podręcznik dla liceum ogólnokształcącego i technikum

Zakres rozszerzony

nowa era Twoje mocne strony

Informatyka na czasie

Podręcznik dopuszczony do użytku szkolnego przez ministra właściwego do spraw oświaty i wychowania i wpisany do wykazu podręczników przeznaczonych do kształcenia ogólnego do nauczania informatyki, na podstawie opinii rzeczoznawców:
mgr inż. Anny Koludo, dr. hab. Henryka Nogi, mgr Teresy Zawiszy-Chlebowskiej.

Etap edukacyjny: III
Typ szkoły: liceum ogólnokształcące i technikum
Rok dopuszczenia: 2020

Numer ewidencyjny w wykazie MEN: 1037/2/2020

Podręcznik został opracowany na podstawie *Programu nauczania informatyki w liceach i technikach. „Informatyka na czasie". Zakres rozszerzony* autorstwa Zbigniewa Talagi.

Nabyta przez Ciebie publikacja jest dziełem twórcy i wydawcy. Prosimy o przestrzeganie praw, jakie im przysługują. Zawartość publikacji możesz udostępnić nieodpłatnie osobom bliskim lub osobiście znanym, ale nie umieszczaj jej w internecie. Jeśli cytujesz jej fragmenty, to nie zmieniaj ich treści i koniecznie zaznacz, czyje to dzieło. Możesz skopiować część publikacji jedynie na własny użytek.

Szanujmy cudzą własność i prawo. Więcej na **www.legalnakultura.pl**

Wykorzystywanie treści niniejszego Utworu, w jakiejkolwiek formie lub w jakimkolwiek procesie, do celów eksploracji tekstu i danych, uczenia maszynowego, rozwoju i/lub szkolenia i/lub wzbogacania sztucznej inteligencji jakiegokolwiek rodzaju jest surowo zabronione.

© Copyright by Nowa Era Sp. z o.o. 2024
ISBN 978-83-267-5070-0

Konsultacja merytoryczno-dydaktyczna: Bartosz Bieganowski, Jacek Detlaf, Bożena Karolak, Paweł Mateja, Damian Kurpiewski, Paweł Perekietka.

Redakcja merytoryczna: Anna Nasiadka, Przemysław Szydzik.
Współpraca redakcyjna: Krzysztof Fit, Joanna Sawicka, Agnieszka Sieczak.
Redakcja językowa: Marta Zuchowicz, Paulina Szulim, Aleksandra Hofman, Joanna Sawicka.
Korekta językowa: Paulina Szulim.
Nadzór artystyczny: Kaia Pichler.
Opieka graficzna: Ewa Kaletyn.
Projekt okładki: Maciej Galiński.
Projekt graficzny: Sławomir Włodarczyk.
Opracowanie graficzne: Sławomir Włodarczyk, Klaudia Jarocka-Ciok, Marcin Kołacz, Halszka Zwalińska, Studio Graficzne Fastwork Grzegorz Afeltowicz.
Ilustracje: Sławomir Włodarczyk, Krzysztof Mrawiński, Natalia Helman.
Realizacja projektu graficznego: Artur Polakowski.
Fotoedycja: Katarzyna Iwan-Malawska.
W podręczniku wykorzystano teksty autorstwa Pawła Perekietki (wybrane teksty na stronach: 8–12, 73, 138–139, 318, 322–330, 333–335, 339–340; infografiki: s. 45, 74; *A to ciekawe*: s. 83, 105; *Warto wiedzieć*: s. 18, 21 (pierwsze od góry), 23 (drugie od góry), 81 (drugie od góry)) oraz teksty autorstwa Zbigniewa Talagi (wybrane teksty na stronach: 43–44; tabela 1.2 s. 15; zadania 8 i 9 s. 47).

Nowa Era Sp. z o.o.
Aleje Jerozolimskie 146 D, 02-305 Warszawa
www.nowaera.pl, e-mail: nowaera@nowaera.pl, Centrum Kontaktu: 58 721 48 00

Spis treści

1. Algorytmy na liczbach całkowitych i tekstach ... 7

Kartka z historii ... 8

1. Od problemu do programu ... 10
   - 1.1. Od problemu do algorytmu ... 10
   - 1.2. Sposoby zapisu algorytmów ... 15
   - 1.3. Od algorytmu do programu w języku C++ ... 18
2. Systemy liczbowe i reprezentacja danych w komputerze ... 27
   - 2.1. Systemy liczbowe ... 27
   - 2.2. Reprezentacja liczb całkowitych w komputerze ... 32
   - 2.3. Reprezentacja znaków i napisów ... 36
   - 2.4. Operacje logiczne, reprezentacja wartości logicznych ... 40
   - 2.5. Reprezentacja obrazów, dźwięków i animacji ... 43
3. Algorytmy zamiany reprezentacji liczb między systemami liczbowymi ... 48
   - 3.1. Algorytm zamiany reprezentacji liczby z dziesiętnej na dwójkową ... 48
   - 3.2. Algorytmy zamiany liczby dwójkowej na dziesiętną ... 50
   - 3.3. Uogólnienie algorytmów zamiany reprezentacji liczb między systemami pozycyjnymi ... 53
   - 3.4. Algorytm szybkiego podnoszenia do potęgi ... 57
4. Czy to jest palindrom? ... 61
   - 4.1. Sprawdzamy, czy wyraz jest palindromem ... 61
   - 4.2. Znajdujemy słowa palindromy w zdaniu ... 64
   - 4.3. Sprawdzamy, czy zdanie jest palindromem ... 69
5. Czy ta liczba jest pierwsza? ... 73
   - 5.1. Liczby pierwsze i złożone ... 73
   - 5.2. Sprawdzamy, czy liczba jest pierwsza ... 75
   - 5.3. Rozkładamy liczbę na czynniki pierwsze ... 79
   - 5.4. Liczby bliźniacze ... 81
6. Działania na liczbach w systemach innych niż dziesiętny ... 86
   - 6.1. Dodawanie liczb w różnych systemach pozycyjnych ... 86
   - 6.2. Odejmowanie liczb w różnych systemach pozycyjnych ... 90
   - 6.3. Mnożenie liczb w różnych systemach pozycyjnych ... 94
   - 6.4. Dzielenie liczb w różnych systemach pozycyjnych ... 98
7. Algorytm Euklidesa i działania na ułamkach ... 102
   - 7.1. Algorytm Euklidesa z wykorzystaniem odejmowania ... 102
   - 7.2. Algorytm Euklidesa z wykorzystaniem dzielenia ... 106
   - 7.3. Reprezentacja liczb wymiernych ... 109
   - 7.4. Dodawanie i mnożenie liczb wymiernych ... 110

8. Szyfr Cezara i inne szyfry podstawieniowe ... 117
8.1. Szyfrujemy napis szyfrem Cezara ... 117
8.2. Szyfrujemy tekst wczytany z pliku tekstowego ... 121
8.3. Szyfrujemy tekst z pliku z uwzględnieniem polskich znaków diakrytycznych ... 124
8.4. Przykłady innych szyfrów podstawieniowych ... 126

Wiesz, umiesz, zdasz ... 130

2. Rozwiązywanie problemów z wykorzystaniem struktur danych ... 137

Kartka z historii ... 138

9. Łamiemy szyfr Cezara ... 140
9.1. Metody łamania szyfru Cezara ... 140
9.2. Zliczamy wystąpienia liter w tekście ... 142
9.3. Znajdujemy maksimum ... 146
9.4. Wyznaczamy klucz szyfrowania i odszyfrowujemy tekst ... 147

10. Poszukujemy liczby ... 151
10.1. Sformułowanie problemu wyszukiwania ... 151
10.2. Poszukujemy liczby w zbiorze nieuporządkowanym ... 153
10.3. Poszukujemy liczby w zbiorze uporządkowanym ... 154
10.4. Jednoczesne znajdowanie minimum i maksimum ... 159

11. Jak ocenić złożoność obliczeniową algorytmu? ... 164
11.1. Czasowa złożoność obliczeniowa ... 164
11.2. Pamięciowa złożoność obliczeniowa ... 167
11.3. Algorytm optymalny ... 168

12. Metody sortowania prostego ... 170
12.1. Sformułowanie problemu sortowania ... 170
12.2. Sortowanie bąbelkowe (przez prostą zamianę) ... 171
12.3. Sortowanie przez wybieranie ... 174
12.4. Sortowanie przez wstawianie ... 175
12.5. Porównanie metod sortowania prostego ... 178

13. Szyfry przestawieniowe, anagramy ... 182
13.1. Przykłady szyfrów przestawieniowych ... 182
13.2. Sprawdzamy, czy dwa słowa są anagramami ... 185

14. Sito Eratostenesa ... 190
14.1. Jak działa sito Eratostenesa? ... 190
14.2. Implementacja algorytmu sita Eratostenesa ... 192

15. Szukamy różnych podciągów ... 197
15.1. Szukamy długości najdłuższego spójnego podciągu niemalejącego ... 197
15.2. Szukamy najdłuższego spójnego podciągu niemalejącego ... 199
15.3. Szukamy maksymalnej sumy podciągu spójnego ... 203

15.4. Porównujemy algorytmy znajdowania maksymalnej sumy podciągu spójnego ... 206
15.5. Szukamy podciągu spójnego o maksymalnej sumie ... 207

16. W poszukiwaniu lidera i idola ... 212
16.1. Jak znaleźć lidera w zbiorze? ... 212
16.2. Efektywny algorytm poszukiwania lidera ... 215
16.3. Porównanie złożoności czasowej algorytmów poszukiwania lidera ... 218
16.4. Jak znaleźć idola w zbiorze? ... 218
16.5. Efektywny algorytm poszukiwania idola ... 223
16.6. Porównanie złożoności algorytmów poszukiwania idola ... 224

Wiesz, umiesz, zdasz ... 226

3. Metody algorytmiczne ... 233

Kartka z historii ... 234

17. Iteracja a rekurencja ... 236
17.1. Rekurencja w matematyce ... 236
17.2. Podział liczby na cyfry z wykorzystaniem rekurencji ... 240
17.3. Rekurencyjny algorytm szybkiego podnoszenia do potęgi ... 241
17.4. Jak wejść na wieżę Eiffla, czyli liczby Fibonacciego ... 244
17.5. Rozszerzony algorytm Euklidesa ... 250

18. Metoda zachłanna ... 255
18.1. Wydawanie reszty metodą zachłanną ... 255
18.2. Optymalne wykorzystanie sali ... 259
18.3. Maksymalna suma w kwadracie – metoda zachłanna ... 262
18.4. Pakowanie plecaka metodą zachłanną ... 265

19. Programowanie dynamiczne ... 271
19.1. Wydawanie reszty metodą programowania dynamicznego ... 271
19.2. Maksymalna suma w kwadracie – programowanie dynamiczne ... 277
19.3. Pakowanie plecaka metodą programowania dynamicznego ... 280
19.4. Znajdowanie najdłuższego wspólnego podciągu ... 285

20. Dziel i zwyciężaj, czyli sortujemy sprawniej ... 292
20.1. Sortowanie szybkie ... 292
20.2. Modyfikacje algorytmu sortowania szybkiego ... 297
20.3. Sortowanie przez scalanie ... 298

Wiesz, umiesz, zdasz ... 310

Z informatyką w przyszłość ... 318

Projekt zespołowy ... 319

Dodatki ... 321

Indeks ... 341

O czym jest podręcznik?

Podręcznik *Informatyka na czasie 2* pomoże ci pogłębić wiedzę na temat algorytmiki i programowania w języku C++. Poznasz różne sposoby zapisu algorytmów, rozwiążesz interesujące problemy algorytmiczne oraz poznasz podstawy jednego z najpopularniejszych języków programowania na świecie.

Czemu służą poszczególne elementy podręcznika?

Cele lekcji

Informacje na temat tego, czym będziesz się zajmować na danej lekcji.

Ćwiczenie 1

Proste zadania w toku lekcji. Odpowiedzi nie należy zapisywać w podręczniku.

Zadania

Zestaw poleceń o różnym stopniu trudności. Odpowiedzi nie należy zapisywać w podręczniku.

Dobra rada

Podpowiedzi ułatwiające wykonanie omawianych czynności.

Dobra rada

Wskazówki szczególnie przydatne w kształceniu kluczowych umiejętności.

A to ciekawe

Ciekawostki pokazujące omawiane zagadnienia z zaskakującej perspektywy.

Warto wiedzieć

Informacje poszerzające wiedzę o omawianych zagadnieniach.

Podsumowanie

Najważniejsze informacje z lekcji podane w punktach, porządkujące nowo zdobytą wiedzę.

Projekt zespołowy

Zadanie do wykonania w grupie, z wykorzystaniem wiedzy z kilku rozdziałów.

Kartka z historii

Wydarzenia najważniejsze dla rozwoju danej dziedziny informatyki.

Zapamiętaj

Najważniejsze pojęcia omawiane na lekcji, które należy zapamiętać.

Z informatyką w przyszłość

Przykłady zastosowania treści omawianych w podręczniku w przyszłym życiu zawodowym.

Oznaczenia w podręczniku

- Odsyłacz do zagadnienia omawianego szerzej w innym miejscu
- Wskaźnik pojęcia; pojęcia obok wskaźników układają się w siatkę pojęć z danej lekcji
- *** Oznaczenie stopnia trudności zadań

WIESZ, UMIESZ, ZDASZ

Sposób na zadania

Zadania rozwiązane krok po kroku, które ułatwiają opanowanie różnych typów zadań.

Trening

Zestaw zadań pozwalających utrwalić wiedzę z danego rozdziału.

1 Algorytmy na liczbach całkowitych i tekstach

1. Od problemu do programu
2. Systemy liczbowe i reprezentacja danych w komputerze
3. Algorytmy zamiany reprezentacji liczb między systemami liczbowymi
4. Czy to jest palindrom?
5. Czy ta liczba jest pierwsza?
6. Działania na liczbach w systemach innych niż dziesiętny
7. Algorytm Euklidesa i działania na ułamkach
8. Szyfr Cezara i inne szyfry podstawieniowe

Wiesz, umiesz, zdasz

Algorytmika i programowanie

Już ok. 1800 r. p.n.e. starożytni Egipcjanie stosowali algorytm mnożenia – nazwany potem **mnożeniem egipskim** – w którym używano wyłącznie operacji podwajania i dodawania. Najstarszy znany opis tego algorytmu zawiera papirus Rhinda, spisany ok. 1650 r. p.n.e. przez skrybę Ahmesa i odnaleziony dopiero w połowie XIX w. Papirus ten jest obecnie częścią zbiorów British Museum w Londynie.

1800 r. p.n.e.

VII w. p.n.e.

Grecki poeta Archiloch w jednym z dzieł wspomniał o przyrządzie do szyfrowania zwanym *skutálē*, używanym w Sparcie od VII w. p.n.e. Nadawca nawijał na laskę pasek pergaminu i pisał tekst na stykających się brzegach. Adresat posiadał identyczną laskę, umożliwiającą odczytanie wiadomości. Dziś ten **szyfr przestawieniowy** nazywa się metodą płotu.

Jednym z najstarszych algorytmów obliczeniowych jest **algorytm Euklidesa**, czyli metoda wyznaczania największego wspólnego dzielnika dwóch liczb, opisana przez greckiego matematyka Euklidesa w dziele *Elementy*. Pierwotnie algorytm ten służył do szukania odcinka, który byłby wspólną miarą dla dwóch dłuższych odcinków.

300 r. p.n.e.

III w. p.n.e.

Greckiemu matematykowi Eratostenesowi z Cyreny przypisuje się wynalezienie metody znajdowania liczb pierwszych z przedziału [2; *n*]. Polega ona na odsiewaniu liczb, które są wielokrotnościami innych liczb, stąd nosi nazwę **sita Eratostenesa**. Eratostenes był także astronomem i geografem – przeprowadził eksperyment, dzięki któremu obliczył obwód Ziemi.

Pierwszym opisanym **szyfrem podstawieniowym** był prawdopodobnie szyfr z czasów wojen galijskich, stosowany w korespondencji przez Juliusza Cezara. Szyfr polega na cyklicznym przesuwaniu liter alfabetu. Cezar za literę A podstawiał literę D, za B – E itd., a za Z – C. Mimo że tekst zaszyfrowany w ten sposób bardzo łatwo odszyfrować, szyfru Cezara używano jeszcze na początku XX w.

I w. p.n.e.

I w. n.e.

Heron z Aleksandrii – starożytny grecki matematyk, mechanik i wynalazca – w dziele *Metrica* opisał efektywną metodę iteracyjnego wyznaczania przybliżeń pierwiastka kwadratowego z użyciem operacji dodawania, mnożenia i dzielenia. Dziś tę metodę nazywa się **algorytmem Herona**. Za największe osiągnięcia Herona uznaje się sformułowanie wzoru na pole trójkąta (o znanych długościach boków) i wymyślenie pierwowzoru maszyny parowej.

Muhammad Ibn Musa Al-Chuwarizmi, bibliotekarz Domu Mądrości w Bagdadzie, opisał efektywne sposoby pisemnego wykonywania obliczeń z użyciem systemu dziesiętnego, znanego wcześniej wyłącznie w Indiach. Dzięki jego pismom system ten dotarł do Europy. Od łacińskiej wersji nazwiska tego uczonego pochodzi słowo **algorytm**.

IX w.

1202 r.

Leonardo Fibonacci w dziele *Liber abaci* z 1202 r. rozważał problem rozmnażania się populacji królików. W rozwiązaniu pojawia się ciąg liczb: 1, 1, 2, 3, 5, 8, 13, 21, …, w którym: 2 = 1 + 1, 3 = 1 + 2, 5 = 2 + 3, 8 = 3 + 5 itd. Ciąg ten opisuje **zależność rekurencyjna**: następna liczba jest sumą dwóch poprzednich. Kolejne liczby Fibonacci wypisał na prawym marginesie rękopisu.

1. Od problemu do programu

Codziennie spotykasz się z różnymi sytuacjami problemowymi. W niektórych z nich, np. przy wiązaniu sznurówek, działasz automatycznie. Gdy jednak robisz coś po raz pierwszy, postępujesz krok po kroku według ustalonego przepisu. Można podać wiele przykładów problemów, których rozwiązanie polega na wykonywaniu ciągu kolejnych instrukcji. W tym temacie poznasz podstawy algorytmicznego rozwiązywania problemów.

Cele lekcji

- Zrozumiesz kolejne etapy rozwiązywania problemów.
- Poznasz różne sposoby przedstawiania algorytmów.
- Rozwiążesz proste problemy algorytmiczne dotyczące liczb całkowitych.
- Zastosujesz podstawowe konstrukcje programistyczne w języku C++.
- Napiszesz w języku C++ proste programy przedstawiające rozwiązania omawianych problemów.

Na początku rozwiązywania problemu z wykorzystaniem komputera musisz określić dane początkowe i efekt (wynik), który chcesz osiągnąć. Następnie opracowujesz **algorytm**. Trzeba nauczyć się wyróżniać kroki algorytmu, a w ich opisie stosować operacje, które będzie można potem zapisać w języku programowania. Kod źródłowy programu komputerowego to właśnie zapis algorytmu w języku programowania.

Algorytm to precyzyjnie zapisany sposób postępowania, prowadzący do rozwiązania problemu.

1.1. Od problemu do algorytmu

Omówimy teraz kilka przykładów problemów. Dla każdego z nich przedstawimy algorytm jego rozwiązania.

Problem 1. Jak zapleść warkocz klasyczny?

Wyobraź sobie, że musisz wytłumaczyć komuś, jak zapleść warkocz klasyczny. Zapewne najłatwiej będzie ci pokazać to na przykładzie. Jeśli nie masz takiej możliwości, musisz precyzyjnie opisać sposób postępowania. Algorytm zaplatania warkocza może wyglądać następująco:

1. `Rozdziel włosy na trzy pasma.`
2. `Dopóki nie spleciesz warkocza po końcówki włosów, wykonuj kroki 3 i 4.`
   3. `Przełóż lewe pasmo nad środkowym. Teraz to pasmo staje się środkowym.`
   4. `Przełóż prawe pasmo nad środkowym. Teraz to pasmo staje się środkowym.`

Warto wiedzieć

Ludzie od tysięcy lat zaplatają warkocze – zdobili sobie nimi głowy już starożytni Egipcjanie. Poniższe zdjęcie przedstawia warkocz klasyczny.

Ćwiczenie 1

Odpowiedz na pytania dotyczące podanej instrukcji zaplatania warkocza klasycznego.

a. Które czynności można uznać za podstawowe (niepodzielne)?
b. Czy opis każdej czynności jest dla ciebie jednoznaczny?
c. W którym kroku jest określone zakończenie działania algorytmu?

Dobra rada

Kiedy podajesz instrukcję postępowania, czyli algorytm, staraj się to robić maksymalnie precyzyjnie.

Problem 2. Mnożenie liczb sposobem pisemnym

Mnożenia liczb sposobem pisemnym uczyliście się w szkole podstawowej. Wróćmy teraz do tego algorytmu i przyjrzyjmy mu się bliżej. Warto to zrobić m.in. dlatego, że ta metoda, odpowiednio zoptymalizowana i dostosowana do **systemu dwójkowego** (**binarnego**), jest stosowana w technice komputerowej.

System dwójkowy (binarny), s. 28–29

Załóżmy, że chcemy pomnożyć dwie liczby całkowite dodatnie, przynajmniej dwucyfrowe, np. 2782 i 213. Mnożenie pisemne wyglądałoby tak jak poniżej.

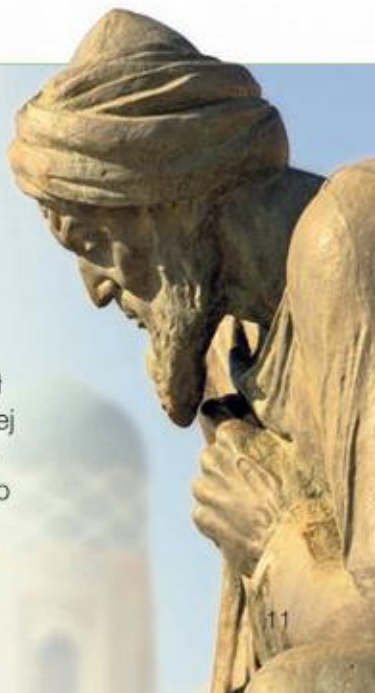
```
  2782
· 213
------
  8346  (2782 pomnożone przez 3)
 2782   (2782 pomnożone przez 1)
+5564   (2782 pomnożone przez 2)
------
 592566
```

Istotą zastosowanego algorytmu mnożenia dwóch liczb a i b jest utworzenie sum pośrednich (przesuniętych iloczynów cząstkowych), z których każda reprezentuje iloczyn liczby a przez jedną z cyfr liczby b. Wartości te są odpowiednio przesuwane w lewo i następnie dodawane.

A to ciekawe

Skąd pochodzi słowo „algorytm”?

Słowo „algorytm” pochodzi od *Algorithmus* – łacińskiego przekładu ostatniego z członów nazwiska perskiego matematyka i astronoma, który był znany jako Muhammad Ibn Musa Al-Chuwarizmi. Żył on na przełomie VIII i IX w. n.e. i był bibliotekarzem Domu Mądrości w Bagdadzie. Zajmował się m.in. pozycyjnym systemem dziesiętnym zapisu liczb, znanym wcześniej wyłącznie w Indiach. Opisał efektywne sposoby pisemnego wykonywania obliczeń. W XII w. przetłumaczono jego dzieło na łacinę, co zapoczątkowało w Europie stopniowe odejście w arytmetyce handlowej od posługiwania się żetonami i stołami obliczeniowymi (abakami).



Specyfikacja problemu

Zapiszemy teraz algorytm pisemnego mnożenia dwóch liczb całkowitych dodatnich, przynajmniej dwucyfrowych. Opis poprzedzimy **specyfikacją problemu**, czyli określeniem danych, wyniku oraz związku między danymi i wynikiem.

Specyfikacja

Dane: a, b – liczby całkowite, $a > 9$, $b > 9$.

Wynik: iloczyn liczb a i b.

1. Zapisz liczbę a.
2. Zapisz liczbę b poniżej liczby a tak, żeby jej cyfry jedności, dziesiątek, setek itd. znajdowały się odpowiednio pod cyframi jedności, dziesiątek, setek itd. liczby a.
3. Dla każdej z cyfr liczby b, począwszy od cyfry jedności, traktowanych jako liczby jednocyfrowe, wykonaj kroki 4 i 5.
   4. Oblicz iloczyn bieżącej liczby jednocyfrowej i liczby a.
   5. Jeśli iloczyn jest pierwszym cząstkowym iloczynem, to zapisz go poniżej liczby b (jedności pod jednościami, dziesiątki pod dziesiątkami itd.). Kolejne iloczyny zapisuj z odpowiednim przesunięciem w lewo, tj. cyfrę jedności iloczynu zapisz pod cyfrą dziesiątek poprzedniego iloczynu, cyfrę dziesiątek iloczynu zapisz pod cyfrą setek poprzedniego iloczynu itd.
6. Zsumuj cząstkowe iloczyny.

Warto wiedzieć

Wiele instrukcji służących ratowaniu życia człowieka przyjmuje postać algorytmu – precyzyjny opis może być niezwykle pomocny w stresującej sytuacji. W postaci algorytmów są podawane np. sposób użycia defibrylatora lub instrukcja ewakuacji z budynku.

Zwróć uwagę, że niektóre kroki wymagają rozwiązania mniejszych problemów (podproblemów). Z taką sytuacją mamy do czynienia np. w kroku 3 (wyodrębnianie cyfr liczby), kroku 4 (mnożenie pisemne przez liczbę jednocyfrową) i kroku 6 (pisemne dodawanie liczb całkowitych dodatnich). W przyszłości często będziesz się spotykać z takim podejściem: dzieleniem złożonego problemu na podproblemy.

Problem 3. Podział liczby na cyfry

Teraz przedstawimy algorytm wypisujący kolejne cyfry liczby całkowitej dodatniej, zaczynając od cyfry jedności (od prawej do lewej).

Specyfikacja

Dane: n – liczba całkowita, $n > 0$.

Wynik: kolejne cyfry liczby n, zaczynając od cyfry jedności.

Do wykonania zadania wykorzystamy dzielenie przez 10. Weźmy przykładową liczbę 213. Po podzieleniu jej przez 10 otrzymamy część całkowitą 21 oraz resztę 3, która jest wyodrębnioną cyfrą jedności.

21 wykorzystamy do wyodrębnienia cyfry dziesiątek – ponownie wykonujemy dzielenie przez 10. Wyniki kolejnych działań dla liczby 213 przedstawia tabela 1.1.

Liczba	Część całkowita z dzielenia przez 10	Reszta z dzielenia przez 10
213	21	3
21	2	1
2	0	2

Tabela 1.1. Wyniki dzielenia prowadzące do wyodrębnienia cyfr liczby 213

Algorytm podziału liczby na cyfry może wyglądać tak jak poniżej.

1. `Dopóki liczba jest większa od zera, wykonuj kroki 2 i 3.`
   2. `Wypisz resztę z dzielenia liczby przez 10.`
   3. `Zastąp liczbę częścią całkowitą z dzielenia liczby przez 10.`

Ćwiczenie 2

Zamień w powyższym algorytmie dzielenie przez 10 na dzielenie przez 2. Ustal, co będzie wynikiem.

Problem 4. Podział na równoliczne grupy

Czasami zachodzi potrzeba podziału grupy osób na mniejsze równoliczne grupy, np. podczas treningów piłkarskich (gry kontrolne) czy na lekcjach WF. Chcemy podać algorytm, po którego wykonaniu dowiemy się, na ile sposobów można to zrobić. Może się zdarzyć, że grupy nie da się tak podzielić.

Przyjmijmy, że chcemy się dowiedzieć, na ile sposobów można podzielić grupę n osób na równoliczne, co najmniej dwuosobowe zespoły. Rozwiązanie zadania sprowadza się do policzenia, ile **dzielników właściwych** większych od 1 ma dana liczba.

Dzielniki właściwe to dzielniki mniejsze od danej liczby. Każda liczba całkowita większa od 1 ma przynajmniej jeden dzielnik właściwy – liczbę 1.

Specyfikacja

Dane: n – liczba całkowita, $n > 0$.

Wynik: liczba dzielników właściwych większych od 1.

W algorytmie wykorzystamy zmienną `licznik_dzielników`. **Zmienna** to miejsce przechowywania danych, które ma nazwę i w którym można zapisać pewną wartość. Do wartości odwołujemy się za pomocą nazwy zmiennej.

Zmienna

Największym możliwym dzielnikiem właściwym liczby n jest jej połowa (dla liczby parzystej to dokładnie jej połowa, dla nieparzystej – liczba mniejsza niż połowa). Dlatego w kroku 2 podanego niżej algorytmu zliczającego dzielniki właściwe przeglądamy liczby od 2 do połowy n.

1. `Ustaw licznik_dzielników na 0.`
2. `Dla liczb od 2 do połowy n wykonuj krok 3.`
   3. `Jeśli reszta z dzielenia n przez liczbę jest równa zero, powiększ wartość zmiennej licznik_dzielników o 1.`

Warto wiedzieć

Często dla danego problemu można znaleźć różne metody rozwiązania, czyli różne algorytmy.

Zapiszmy inny algorytm rozwiązujący ten sam problem. Zwróć uwagę, że gdy znajdziemy jeden dzielnik, automatycznie znaleźliśmy drugi. Jeśli liczba n dzieli się bez reszty przez liczbę d_1, to dzieli się także przez liczbę $d_2 = n : d_1$ ($n = d_1 \cdot d_2$). Należy tylko uważać na sytuację, gdy n jest kwadratem jakiejś liczby, czyli $d_1 = d_2$. Wówczas trzeba zwiększyć wartość zmiennej `licznik_dzielników` o jeden, a nie o dwa.

1. `Ustaw licznik_dzielników na 0.`
2. `Ustaw dzielnik d na 2.`
3. `Dopóki d · d < n, wykonuj kroki 4 i 5.`
   4. `Jeśli reszta z dzielenia n przez d jest równa zero, powiększ licznik_dzielników o 2.`
   5. `Powiększ d o jeden.`
6. `Jeśli d · d = n, powiększ licznik_dzielników o 1.`

Ćwiczenie 3

Ile operacji dzielenia jest wykonywanych w każdym z algorytmów zliczających dzielniki właściwe danej liczby większe od 1?

Często można podać różne algorytmy rozwiązania danego problemu, różniące się m.in. liczbą wykonywanych operacji. Cechą dobrego algorytmu jest jego duża **efektywność**, czyli możliwie najmniejsza liczba wykonywanych operacji.

Efektywność algorytmu

Inne cechy, które powinien mieć algorytm, to:

Skończoność algorytmu

- **skończoność** – algorytm kończy swoje działanie w skończonej liczbie kroków dla każdego zestawu danych zgodnych ze specyfikacją;

Poprawność (całkowita poprawność) algorytmu

- **poprawność** (**całkowita poprawność**) – algorytm zwraca poprawne wyniki dla każdego zestawu poprawnych danych.

Warto wiedzieć

Algorytm może też cechować się częściową poprawnością – zwraca wtedy poprawne wyniki dla poprawnych danych, o ile jego wykonanie się zatrzyma. Algorytm nie ma wówczas cechy skończoności.

Zapamiętaj

Algorytm to skończony ciąg precyzyjnie określonych instrukcji prowadzący do rozwiązania problemu. Specyfikacja problemu algorytmicznego to opis danych, wyniku oraz związku między danymi a wynikiem.

1.2. Sposoby zapisu algorytmów

Istnieją różne sposoby przedstawiania algorytmów. Jednym z nich jest **lista kroków**. Składa się ona z zapisanych słownie kolejnych pojedynczych instrukcji, nazywanych krokami, które mogą odwoływać się do siebie nawzajem. Zauważ, że w zagadnieniu 1.1 tego tematu algorytmy rozwiązania podanych problemów przedstawiliśmy w postaci listy kroków.

Lista kroków

Ćwiczenie 4

Zapisz w postaci listy kroków algorytm opisujący, jak zawiązać sobie krawat. Jeśli nie umiesz wiązać krawata, możesz np. poszukać filmu instruktażowego w internecie.

Możesz się spotkać także z przedstawieniem algorytmu w postaci **schematu blokowego**. Jest to graficzny sposób pokazania algorytmu, w którym określone figury geometryczne powiązane są z typami instrukcji i połączone zgodnie z kolejnością wykonywania kroków rozwiązywanego problemu.

Schemat blokowy

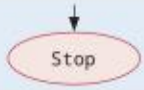
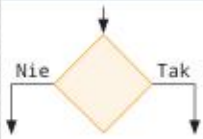
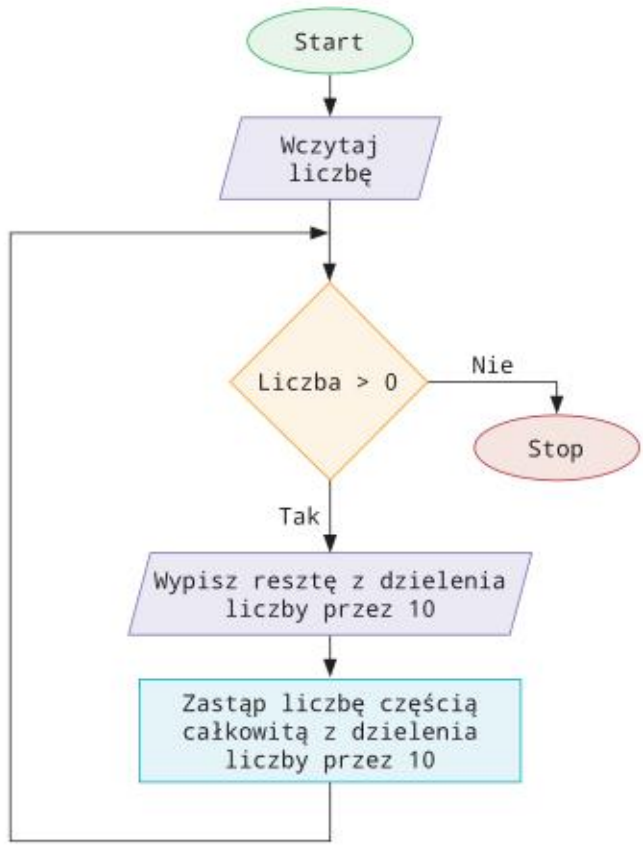
Symbol	Operacja	Wyjaśnienie
	Początek	Miejsce rozpoczęcia działania algorytmu
	Koniec	Miejsce zakończenia algorytmu
	Wejście/wyjście	Wprowadzenie/wyprowadzenie danych
	Przetwarzanie (obliczanie)	Przetwarzanie danych (wykonywanie obliczeń)
	Decyzja	Operacja umożliwiająca wybór jednej z alternatywnych dróg działania
	Przepływ danych	Kierunek przepływu danych
	Łączenie	Łączenie dróg przepływu danych
	Skrzyżowanie	Skrzyżowanie dróg przepływu danych bez powiązania między nimi

Tabela 1.2. Znaczenie elementów w schematach blokowych

Warto wiedzieć

Schemat blokowy nazywany jest także siecią działań lub schematem wykonawczym.

Gdybyśmy chcieli przedstawić za pomocą schematu blokowego omawiany wcześniej algorytm podziału liczby na cyfry, mógłby wyglądać tak jak na rysunku 1.1.



Rys. 1.1. Schemat blokowy algorytmu podziału liczby na cyfry

Ćwiczenie 5

Przedstaw w postaci schematu blokowego jeden z algorytmów obliczających dla danej liczby całkowitej dodatniej liczbę jej dzielników właściwych większych od 1.

Pseudokod

Jedną z częściej spotykanych metod zapisu algorytmów jest **pseudokod**. W odróżnieniu od schematu blokowego nie ma ściśle określonych reguł składniowych, ale charakteryzuje się prostotą i czytelnością. Pseudokod zachowuje strukturę charakterystyczną dla kodu zapisanego w języku programowania. Ze względu na te cechy w dalszej części podręcznika będziemy zapisywać algorytmy w pseudokodzie.

Algorytm podziału liczby na cyfry w pseudokodzie może wyglądać następująco:

```
dopóki liczba > 0 wykonuj
    wypisz liczba mod 10
    liczba ← liczba div 10
```

Operacja wyznaczania reszty z dzielenia jest bardzo ważna i często wykorzystywana w algorytmice. W różnych językach programowania jest różnie oznaczana, w pseudokodzie najczęściej oznaczamy ją skrótem **mod** (od słowa „modulo"). Operację dzielenia całkowitego będziemy oznaczać skrótem **div**. Kolejnym znakiem użytym w zapisie algorytmu jest ←. Oznacza on **instrukcję przypisania wartości zmiennej**.

Instrukcja przypisania wartości zmiennej w pseudokodzie

Wróćmy do drugiego z algorytmów obliczających dla całkowitej liczby dodatniej n liczbę jej dzielników właściwych większych od 1. Zapis tego algorytmu w pseudokodzie może być następujący:

```
licznik ← 0
dzielnik ← 2
dopóki dzielnik * dzielnik < n wykonuj
    jeśli n mod dzielnik = 0 to licznik ← licznik + 2
    dzielnik ← dzielnik + 1
jeśli dzielnik * dzielnik = n to licznik ← licznik + 1
wypisz licznik
```

Warto wiedzieć

Instrukcja przypisania wartości zmiennej często jest oznaczana także znakiem := lub po prostu znakiem równości.

Zwróć uwagę na wcięcia w pseudokodzie. Nie tylko poprawiają one czytelność zapisu, lecz także określają kolejność wykonywania instrukcji. W powyższym przykładzie wskazują instrukcje, które są powtarzane.

Ćwiczenie 6

Zapisz w postaci pseudokodu algorytm wypisujący dzielniki właściwe liczby całkowitej dodatniej.

Warto się zastanowić, jaki niezbędny zestaw instrukcji potrzebny jest do zapisu algorytmów. Powinniśmy mieć do dyspozycji pewne operacje elementarne – np. podstawowe działania arytmetyczne, **operacje logiczne** czy instrukcję przypisania wartości zmiennej. Musimy mieć również możliwość ustalania kolejności wykonywania operacji, a więc dysponować pewnymi instrukcjami sterującymi. Dwie niezbędne to:

Logika matematyczna, operatory logiczne i matematyczne, s. 324–327

- **instrukcja warunkowa (warunkowe rozgałęzienie)**
  `jeśli Q to A w przeciwnym przypadku B`
  Jeśli warunek logiczny Q ma wartość **prawda**, wykonaj instrukcję A, w przeciwnym przypadku wykonaj instrukcję B. Część „w przeciwnym przypadku" jest opcjonalna, czyli może nie występować.
- **iteracja warunkowa (instrukcja pętli warunkowej)**
  `dopóki Q wykonuj A`
  Dopóki warunek logiczny Q ma wartość **prawda**, powtarzaj instrukcję A.

Instrukcje A i B mogą być nie tylko operacjami elementarnymi, lecz także instrukcjami sterującymi lub **instrukcją złożoną** – sekwencyjnym ciągiem instrukcji.

Instrukcja złożona

1.3. Od algorytmu do programu w języku C++

Teraz zajmiemy się zapisem algorytmów w języku programowania, czyli tworzeniem kodów źródłowych programów komputerowych. Aby komputer mógł wykonać napisany przez nas program, kod musi zostać przetłumaczony na język wewnętrzny procesora (kod maszynowy).

Zintegrowane środowisko programistyczne (IDE)

Zintegrowane środowiska programistyczne (**IDE**, od ang. *integrated development environment*) umożliwiają nie tylko pisanie kodu źródłowego programu, lecz także jego przetłumaczenie oraz wykonanie.

Translator

Programy tłumaczące nazywamy **translatorami**. Możemy je podzielić na dwie kategorie:

Kompilator

- **kompilatory** – tłumaczą cały kod źródłowy, po zakończeniu tłumaczenia powstaje kod wynikowy – żeby go wykonać, nie jest już potrzebny kompilator,

Interpreter

- **interpretery** – interpretują i wykonują polecenia kodu źródłowego instrukcja po instrukcji; dlatego za każdym razem, gdy chcemy sprawdzić, jak działa program, trzeba uruchamiać interpreter.

Przykładem języka kompilowanego jest język C++, natomiast interpretowanego – język Python. W tym podręczniku skupimy się na tworzeniu programów w języku C++.

Struktura programu w języku C++

Zanim napiszemy i uruchomimy pierwszy program, warto poznać kilka reguł obowiązujących podczas pracy w języku C++.

Słowo kluczowe języka programowania oznacza określoną deklarację lub instrukcję i nie można go przedefiniować. Słowa kluczowe są wyróżniane pogrubioną czcionką i/lub określonym kolorem.

- Wielkość liter ma znaczenie – **słowa kluczowe** języka piszemy małymi literami.
- Każdą instrukcję kończymy średnikiem.
- W instrukcjach nie używamy polskich znaków.
- Nawiasy klamrowe pełnią funkcję początku i końca bloku.
- Tekst po znakach // jest komentarzem. Kompilator pomija komentarze, dlatego można używać w nich polskich znaków.
- Na początku kodu umieszczamy dyrektywę `#include <iostream>` – powoduje ona dołączenie biblioteki z obsługą standardowego wejścia (czytanie danych, domyślnie z klawiatury) i wyjścia (wypisywanie wyników, domyślnie na ekranie).

Warto wiedzieć

Nazwa `iostream` pochodzi od angielskiego wyrażenia *input-output stream*, które możemy rozumieć jako strumień danych wejścia i wyjścia.

- Za pomocą instrukcji `using namespace std;` określamy użycie tzw. standardowej przestrzeni nazw, ułatwiającej odwołanie się m.in. do instrukcji czytania i pisania.
- Każdy program składa się z funkcji.

Funkcja `main`

- Główną część programu stanowi **funkcja** `main`:

```
int main()    // nagłówek funkcji main
{             // początek bloku instrukcji
              // tu będą instrukcje funkcji
}             // koniec bloku instrukcji
```

Programy tworzone w językach programowania mają ustaloną strukturę. **Podstawowa struktura kodu źródłowego programu w języku C++** wygląda następująco:

Podstawowa struktura kodu źródłowego programu w języku C++

```
1. #include <iostream>
2.
3. using namespace std;
4.
5. int main()
6. {
7.
8. }
```

Dobra rada

Utworzenie pliku zawierającego podstawową strukturę programu przyspieszy pisanie nowych programów. Programy tworzone na jego podstawie zapisuj pod nowymi nazwami.

Obok kodu źródłowego programu znajdują się numery, które są oznaczeniami kolejnych linii kodu. Ułatwią one omawianie znaczenia poszczególnych instrukcji. Programy pisane w języku C++ będziemy zapisywać i uruchamiać w zintegrowanym środowisku programistycznym Code::Blocks.

Aby w środowisku Code::Blocks utworzyć nowy, pusty plik, należy wybrać **File → New → Empty file**. Przy zapisywaniu pliku (**File → Save file as...**) środowisko Code::Blocks domyślnie proponuje podanie nazwy z rozszerzeniem *c* (właściwym dla języka C), dlatego trzeba zmienić rozszerzenie na *cpp*, odpowiadające językowi C++.

Dobra rada

Aktualną wersję programu Code::Blocks pobierzesz ze strony codeblocks.org. Dostępne są na niej wersje programu działające pod różnymi systemami operacyjnymi. Pracującym w systemie Windows zalecamy pobranie wersji oprogramowania zawierającej kompilator języka C++ (GCC/G++).

Ćwiczenie 7

W środowisku Code::Blocks utwórz nowy, pusty plik, a następnie przepisz do niego podstawową strukturę programu podaną powyżej. Zapisz plik pod nazwą *szablon.cpp*.

Dobra rada

Jeśli nie wiesz, jak wymawiać obcojęzyczne nazwy umieszczone w podręczniku, skorzystaj z translatora, który daje możliwość odsłuchania tekstu, np. Tłumacza Google lub Bing Microsoft Translator.

Program sprawdzający, czy liczba jest parzysta

Program, który napiszemy, wczyta z klawiatury liczbę całkowitą, sprawdzi, czy jest parzysta, i wypisze na ekranie odpowiedni napis. Zakres danych ograniczymy ze względu na zakres reprezentowanych liczb całkowitych. Więcej o tym dowiesz się w następnym temacie. Specyfikacja problemu oraz zapis algorytmu w pseudokodzie są następujące:

Specyfikacja

Dane: n – liczba całkowita, $-1\,000\,000\,000 < n < 1\,000\,000\,000$.

Wynik: napis „Parzysta", gdy n jest liczbą parzystą, napis „Nieparzysta", gdy n jest liczbą nieparzystą.

```
jeśli n mod 2 = 0 to wypisz "Parzysta"
w przeciwnym przypadku wypisz "Nieparzysta"
```

Kod źródłowy programu realizującego algorytm zapisany w pseudokodzie wygląda następująco:

Kod źródłowy programu sprawdzającego, czy podana liczba jest parzysta

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  int main()
6.  {
7.      int n;
8.      cin>>n;
9.      if (n%2==0)
10.         cout<<"Parzysta";
11.     else
12.         cout<<"Nieparzysta";
13.     return 0;
14. }
```

Dobra rada

Program C++ ignoruje tzw. białe znaki, czyli np. spacje, tabulacje. Dzięki temu możesz dostosować czytelność kodu do swoich preferencji – ustawić takie wcięcia, jakie chcesz.

Kod bloku instrukcji programu (funkcji `main`) znajduje się pomiędzy nawiasami klamrowymi w liniach 7–13. W linii 7 została zadeklarowana zmienna o nazwie `n`. **Deklaracja zmiennej** w języku C++ składa się z nazwy zmiennej oraz określenia **typu danych**, jakie może ona przechowywać. Typ **`int`** oznacza liczby całkowite.

Deklaracja zmiennej

Typ danych

W linii 8 za pomocą **instrukcji wejścia `cin`** wczytujemy liczbę ze standardowego wejścia, czyli klawiatury. Operator `>>` wskazuje, dokąd zostanie przesłana dana. W tym przypadku będzie zapamiętana w zmiennej `n`.

Instrukcja wejścia `cin`

Linie 9–12 odpowiadają zapisowi algorytmu w pseudokodzie. **Instrukcja warunkowa `if else`** w języku C++ ma postać:

Instrukcja warunkowa `if else`

```
if (warunek) instrukcja na tak;
else instrukcja na nie;
```

Warunek logiczny musi być ujęty w nawiasy okrągłe. W linii 9 badamy, czy reszta z dzielenia wartości zmiennej `n` przez 2 jest równa zero, czyli czy `n` jest liczbą parzystą. Do **porównywania wartości** służy operator `==`. Operacja arytmetyczna reszty z dzielenia (**`mod`** w pseudokodzie) to znak `%`. **Instrukcja wyjścia `cout`** służy do wypisania wyniku na ekranie komputera. Operator `<<` wskazuje dane do przekazania na standardowe wyjście. W tym przypadku danymi są teksty „Parzysta" lub „Nieparzysta". Ciąg znaków (zwany też łańcuchem) w kodzie źródłowym musi znajdować się w tzw. amerykańskim cudzysłowie, czyli między znakami cala: " ". Ostatnia instrukcja programu (linia 13) kończy wykonywanie funkcji `main` (a w konsekwencji całego programu) oraz informuje system operacyjny o poprawnym zakończeniu działania (kod `0`).

Porównywanie wartości w języku C++

Instrukcja wyjścia `cout`

Warto wiedzieć

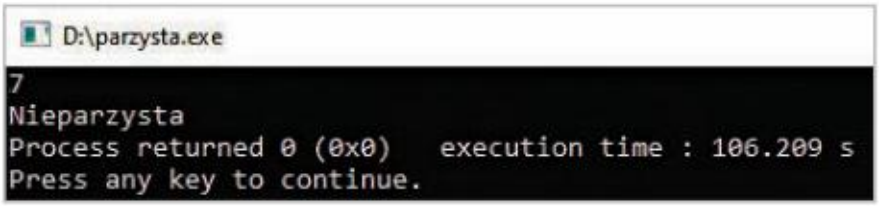
Polecenie **`cin`** pochodzi od angielskiego wyrażenia *console input*, a polecenie **`cout`** – od wyrażenia *console output*.

Po napisaniu programu w środowisku Code::Blocks należy go skompilować i uruchomić – służy do tego opcja **Build → Build and run**.

Jeśli kod programu zapisaliśmy zgodnie z regułami języka, otrzymamy nowy plik: w systemie Windows będzie to plik z rozszerzeniem *exe*. Zostanie on zapisany w tym samym katalogu co plik źródłowy. Jednocześnie Code::Blocks uruchomi program. Można sprawdzić działanie programu, podając liczbę całkowitą (rys. 1.2). Skompilowany program uruchamiamy ponownie, wybierając **Build → Run**.

Warto wiedzieć

W systemach Linux i macOS pliki wykonywalne nie są wyróżniane żadnym rozszerzeniem. Aby można było je uruchomić, muszą mieć ustawiony atrybut wykonywalności.



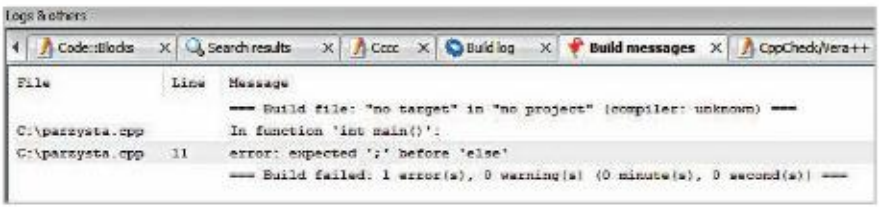
Rys. 1.2. Przykład wywołania programu *parzysta.exe*

Jeśli w kodzie programu znajdzie się **błąd składniowy**, czyli błąd wynikający z zapisania kodu niezgodnie z regułami języka, w oknie programu Code::Blocks wyświetli się odpowiedni komunikat. Na przykład jeśli zapomnimy wstawić średnik na końcu instrukcji w linii 10, otrzymamy informację jak na rysunku 1.3.

Błąd składniowy

Warto wiedzieć

Konsolę zawierającą informację o błędach składniowych włącza się i wyłącza klawiszem **F2**.



Rys. 1.3. Komunikat o braku średnika na końcu instrukcji w linii 10

Kod źródłowy programu może zawierać także **błędy logiczne**. Są to błędy wynikające np. z błędów w algorytmie. Kompilator nie wykryje błędu logicznego – program się skompiluje, jednak dla poprawnych danych nie zwróci poprawnego wyniku.

Błąd logiczny

Ćwiczenie 8

W środowisku Code::Blocks utwórz plik *parzysta.cpp*. Wpisz kod źródłowy programu sprawdzającego, czy dana liczba jest parzysta (zastosuj wcięcia), i zapisz wprowadzone zmiany. Następnie skompiluj i uruchom program. Jeśli wystąpią błędy kompilacji, popraw kod, a następnie ponownie skompiluj i uruchom program. Sprawdź działanie programu dla różnych danych, zarówno liczb parzystych, jak i nieparzystych.

Program dzielący liczbę na cyfry

Program, który teraz napiszemy, będzie implementował (realizował) algorytm omówiony wcześniej w tym temacie.

Specyfikacja

Dane: n – liczba całkowita, $0 < n < 1\,000\,000\,000$.

Wynik: kolejne cyfry liczby n, zaczynając od cyfry jedności (od prawej do lewej).

Kod źródłowy programu wypisującego cyfry wczytanej liczby w pionie (każda cyfra w oddzielnym wierszu) może być następujący:

Kod źródłowy programu dzielącego liczbę na cyfry

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  int main()
6.  {
7.      int n;
8.      cin>>n;
9.      while (n>0)
10.     {
11.         cout<<n%10<<endl;
12.         n=n/10;
13.     }
14.     return 0;
15. }
```

W liniach kodu 9–13 przeglądane są kolejne cyfry wczytanej liczby. Realizuje to **instrukcja iteracji (pętla) `while`**. Jej składnia w języku C++ jest następująca:

Instrukcja iteracji (pętla) `while`

```
while (warunek) instrukcja;
```

Dopóki `warunek` przyjmuje wartość logiczną **`prawda`**, instrukcja w pętli jest powtarzana. Zwróć uwagę, że jeżeli `warunek` pętli na początku przyjmie wartość logiczną **`fałsz`**, to instrukcja w pętli nie wykona się ani razu. W powyższym programie tak się stanie, jeśli wczytamy liczbę ujemną lub zero. Wykonywana w pętli instrukcja musi mieć też wpływ na warunek logiczny pętli, czyli zmieniać wartości, od których ten warunek zależy. W przeciwnym przypadku warunek będzie miał cały czas wartość logiczną **`prawda`** i pętla będzie się wykonywała w nieskończoność. W takich sytuacjach mówimy, że „program się pętli".

Warto wiedzieć

W języku C++ znak / oznacza zarówno dzielenie całkowite (część całkowitą z dzielenia), jak i dzielenie z częścią ułamkową. Interpretacja znaczenia zależy od typu argumentów.

W linii 12 zmienna `n` zmniejsza swoją wartość (jej dotychczasowa wartość jest dzielona całkowicie przez 10). Znak `=` służy do oznaczania przypisania wartości. **Instrukcja przypisania wartości zmiennej w języku C++** ma postać:

Instrukcja przypisania wartości zmiennej w języku C++

```
zmienna = wartość wyrażenia;
```

W linii 11 wypisywana jest reszta z dzielenia wartości zmiennej n przez 10 oraz następuje przejście do następnego wiersza (wstawienie znacznika końca wiersza po drugim operatorze <<, słowo specjalne **endl**). Zwróć uwagę na linie 10 i 13. Są tam nawiasy klamrowe wyznaczające blok instrukcji powtarzanych w pętli – są one konieczne, jeśli w pętli (lub innej instrukcji, np. warunkowej **if else**) chcemy wykonać więcej niż jedną instrukcję. Blok instrukcji jest traktowany jak pojedyncza instrukcja.

Warto wiedzieć

W nazwach zmiennych oprócz małych i wielkich liter oraz cyfr (te nie mogą być na początku nazwy) można używać również znaku podkreślenia _.

Warto wiedzieć

Zapis **endl** pochodzi od angielskiego wyrażenia *end line*, które oznacza koniec linii (wiersza) tekstu.

Ćwiczenie 9

Napisz program dzielący liczbę na cyfry i przetestuj jego działanie.

Programy realizujące algorytmy podziału na równoliczne grupy

Przedstawiliśmy wcześniej dwa algorytmy rozwiązania tego problemu. Napiszemy programy realizujące obydwa algorytmy.

Specyfikacja

Dane: n – liczba całkowita, $0 < n < 1\,000\,000\,000$.

Wynik: liczba dzielników właściwych n większych od 1.

W pierwszym z algorytmów badaliśmy podzielność n przez kolejne liczby od 2 do połowy n. Kod programu może wyglądać następująco:

Kod źródłowy programu liczącego dzielniki właściwe danej liczby większe od 1 – algorytm 1

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  int main()
6.  {
7.      int n, d, ld=0;
8.      cout<<"n="; cin>>n;
9.      for (d=2;d<=n/2;d++)
10.         if (n%d==0) ld++;
11.     cout<<"Liczba dzielnikow wlasciwych wiekszych od 1: ";
12.     cout<<ld<<endl;
13.     return 0;
14. }
```

W linii 7 oprócz n zadeklarowane są jeszcze dwie zmienne typu całkowitego: d – potencjalny dzielnik i ld – licznik dzielników, któremu nadana jest od razu wartość początkowa 0. Wczytanie danej n zostało poprzedzone wypisaniem na ekranie napisu informującego, o jakie dane prosimy, tzw. napisu zachęty (linia 8).

Instrukcja iteracji (pętla) for

W liniach 9 i 10 zastosowaliśmy **instrukcję iteracji (pętlę) for**. Stosuje się ją zwykle w sytuacjach, kiedy z góry znamy liczbę powtórzeń. Jej składnia jest następująca:

```
for (instrukcja inicjująca; warunek powtarzania; instrukcja modyfikująca) instrukcja;
```

Wartością początkową zmiennej d jest 2. Wykonywanie instrukcji pętli odbywa się, dopóki wartość zmiennej d jest mniejsza lub równa n/2. Po każdym wykonaniu instrukcji pętli wartość zmiennej d jest powiększana o 1 (d++). W pętli (linia 10) licznik dzielników ld jest powiększany o 1, gdy aktualna wartość d jest dzielnikiem n (reszta z dzielenia jest równa 0).

Wypisanie na ekranie wyniku (linia 12), czyli wartości licznika dzielników, poprzedzone jest wypisaniem komentarza (linia 11).

Poniżej znajduje się kod źródłowy drugiego algorytmu znajdującego liczbę dzielników właściwych danej liczby większych od 1. Zwróć uwagę na modyfikację warunku pętli **for** (linia 9).

Warto wiedzieć

Pętlę **for** można zapisać za pomocą pętli **while**:

```
instrukcja inicjująca;
while (warunek
  powtarzania)
{
  instrukcja;
  instrukcja
  modyfikująca;
}
```

Warto wiedzieć

Instrukcja d++ jest równoznaczna z instrukcją d=d+1.

Kod źródłowy programu liczącego dzielniki właściwe danej liczby większe od 1 – algorytm 2

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  int main()
6.  {
7.      int n, d, ld=0;
8.      cout<<"n="; cin>>n;
9.      for (d=2;d*d<n;d++)
10.         if (n%d==0) ld=ld+2;
11.     if (d*d==n) ld++;
12.     cout<<"Liczba dzielnikow wlasciwych wiekszych od 1: ";
13.     cout<<ld<<endl;
14.     return 0;
15. }
```

Ćwiczenie 10

Zapisz, uruchom i przetestuj obie przedstawione w temacie wersje programów realizujących algorytmy podziału na równoliczne grupy.

Zapamiętaj

Napisanie programu komputerowego stanowi tylko jeden z etapów rozwiązywania problemu algorytmicznego. Najpierw należy precyzyjnie sformułować problem i podać jego specyfikację, następnie poszukać rozwiązania – algorytmu. Dopiero potem można rozpocząć pisanie programu, pamiętając o testowaniu i ewentualnych poprawkach.

Podsumowanie

- Wyróżniamy następujące etapy rozwiązywania problemów z wykorzystaniem komputera: określenie problemu (zadania), podanie specyfikacji, sformułowanie rozwiązania – algorytmu, zaprogramowanie rozwiązania, testowanie rozwiązania.
- Specyfikacja zadania to określenie danych i wyniku oraz związku między nimi.
- Algorytm to skończony ciąg precyzyjnie określonych instrukcji prowadzący do rozwiązania problemu.
- Dany problem można przeważnie rozwiązać na wiele sposobów, czyli podać różne algorytmy rozwiązania.
- Zapis algorytmu w języku programowania to kod źródłowy programu.
- Testowanie programu polega na wielokrotnym uruchamianiu go dla różnych danych (spełniających warunki specyfikacji).
- Różne sposoby zapisu algorytmów to: lista kroków, schemat blokowy, pseudokod, zapis w języku programowania.
- Podstawowe instrukcje sterujące w językach programowania to instrukcja warunkowa (wykonywanie instrukcji pod pewnym warunkiem) oraz instrukcja iteracji, czyli pętli (wielokrotne powtarzanie instrukcji).
- Translatory (które dzielimy na kompilatory i interpretery) to programy tłumaczące kod źródłowy programu na język maszynowy.
- Zintegrowane środowisko programistyczne (IDE) to program posiadający edytor do pisania kodu źródłowego, translator oraz wiele funkcji ułatwiających uruchamianie programów.

Zadania

* **1** Wiele czynności życia codziennego możemy opisać algorytmicznie. Podaj kilka przykładów. Wybrany z nich zapisz w postaci listy kroków.

* **2** Znajdź w sieci film pokazujący wiązanie dowolnego węzła żeglarskiego. Zapisz algorytm wiązania tego węzła w postaci listy kroków.

* **3** Przedstaw w postaci listy kroków algorytm:
a. pisemnego dodawania dwóch liczb naturalnych,
b. pisemnego mnożenia liczby jednocyfrowej przez wielocyfrową.

* **4** Zapisz w wybranej przez siebie notacji (lista kroków, schemat blokowy, pseudokod) algorytm sumujący cyfry danej liczby całkowitej dodatniej.

* **5** Napisz, uruchom i przetestuj program sumujący cyfry wczytanej liczby całkowitej dodatniej. Najpierw zapisz specyfikację.

** **6** Napisz program sprawdzający, czy w zapisie dziesiętnym wczytanej liczby całkowitej dodatniej pierwsza cyfra od lewej jest równa cyfrze jedności. Program powinien wypisać słowo „TAK" lub „NIE".

** **7** Napisz program wczytujący 5 liczb całkowitych i wypisujący największą z nich.

** **8** Napisz program, który dla wczytanej liczby całkowitej dodatniej wypisze jej najmniejszą cyfrę.

** **9** Napisz program wczytujący 5 liczb całkowitych i sprawdzający, czy w takiej kolejności, w jakiej zostały podane, tworzą ciąg rosnący. Program powinien wypisać słowo „TAK" lub „NIE".

** **10** Napisz program sprawdzający, czy cyfry wczytanej liczby całkowitej dodatniej tworzą ciąg malejący. Program powinien wypisać słowo „TAK" lub „NIE".

** **11** Napisz program sprawdzający, czy wszystkie cyfry wczytanej liczby całkowitej dodatniej są jednocyfrowymi liczbami parzystymi. Program powinien wypisać słowo „TAK" lub „NIE".

*** **12** Napisz program wypisujący dzielniki wczytanej liczby całkowitej dodatniej n. Spróbuj rozwiązać zadanie tak, aby program wykonywał jak najmniej prób dzielenia (nie więcej niż $\sqrt{n}$).

*** **13** Napisz program wypisujący dzielniki wczytanej liczby całkowitej dodatniej n – dzielniki powinny być uporządkowane od najmniejszego do największego. Spróbuj rozwiązać zadanie tak, aby program wykonywał jak najmniej prób dzielenia (nie więcej niż $2 \cdot \sqrt{n}$).

*** **14** Napisz program, który sprawdzi, czy podana liczba całkowita dodatnia jest liczbą palindromiczną (czytana od lewej do prawej i od prawej do lewej ma tę samą wartość).

*** **15** Napisz program realizujący algorytm pisemnego mnożenia dwóch liczb. Możesz wykorzystywać następujące działania: dodawanie dwóch liczb, mnożenie liczby przez cyfrę oraz przez 10, wyznaczanie części całkowitej oraz reszty z dzielenia przez 10.

2. Systemy liczbowe i reprezentacja danych w komputerze

Gdy korzystasz z komputera, przechowujesz w jego pamięci różnego rodzaju informacje, np. teksty, obrazy, dźwięki czy filmy. Zapewne nie zastanawiasz się wtedy, jak komputer interpretuje i zapisuje te dane. Być może pamiętasz, że komputery przechowują informacje, wykorzystując tylko dwie wartości: 0 i 1. Jak to możliwe, że za pomocą tych dwóch znaków da się przedstawić różnorodne dane? Z tego tematu dowiesz się, jakich systemów liczbowych używa się w informatyce i jak za ich pomocą zapisać różne informacje.

Cele lekcji

- Poznasz systemy liczbowe wykorzystywane w informatyce.
- Przypomnisz sobie, czym są bit i bajt.
- Dowiesz się, jak są reprezentowane w komputerze liczby całkowite, wartości logiczne, znaki i napisy oraz obrazy, dźwięki i animacje.
- Nauczysz się dodawać liczby binarne.
- Wykonasz podstawowe operacje logiczne.

2.1. Systemy liczbowe

Powszechnie stosowanym systemem liczbowym jest **system dziesiętny** (**decymalny**), w którym do zapisu liczb wykorzystuje się dziesięć cyfr. System dziesiętny jest **systemem pozycyjnym**. Oznacza to, że z pozycją cyfry w liczbie wiąże się jej waga – cyfry oznaczają wielokrotności kolejnych potęg liczby będącej podstawą systemu, przy czym potęgi rosną dla cyfr od prawej do lewej. W systemie dziesiętnym cyfry reprezentują wielokrotności potęg liczby 10. W poniższym przykładzie najmniej znaczącą cyfrą jest 3, a najbardziej znaczącą 7.

System dziesiętny (decymalny)

System pozycyjny

$$783 = 7 \cdot 10^2 + 8 \cdot 10^1 + 3 \cdot 10^0$$

Zasady tworzenia liczb w różnych systemach pozycyjnych są takie same jak w przypadku systemu dziesiętnego. W systemie o podstawie p, gdzie $p > 1$, do zapisu liczb możemy wykorzystać p cyfr, a każdą liczbę można zapisać w postaci sumy wielokrotności potęg podstawy systemu.

Aby wskazać, w jakim systemie zapisaliśmy liczbę, będziemy stosowali oznaczenie a_p, gdzie a to liczba, natomiast p – podstawa systemu. W przypadku systemu dziesiętnego dla uproszczenia zapisu pominiemy jego podstawę.

Warto wiedzieć

System dziesiętny nie jest jedynym stosowanym na co dzień systemem pozycyjnym. Na przykład systemu sześćdziesiątkowego używamy do określania czasu: 1 godzina to 60 minut, a 1 minuta – 60 sekund. W krajach anglosaskich korzysta się z systemu o podstawie 12, np. długość jest mierzona w stopach i calach, a 1 stopa to 12 cali.

System dwójkowy

System dwójkowy (binarny)

Podstawowym systemem liczbowym używanym w informatyce jest system pozycyjny o podstawie $p = 2$, a więc wykorzystujący tylko dwie cyfry: 0 i 1. Nazywamy go **systemem dwójkowym** lub **binarnym**.

Aby obliczyć wartość dziesiętną liczby zapisanej binarnie, wystarczy wymnożyć cyfry liczby binarnej przez odpowiednie potęgi podstawy systemu (liczby 2), a następnie dodać wyniki. Na przykład:

$$10110011_2 = 1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 =$$
$$= 128 + 32 + 16 + 2 + 1 = 179$$

Cyfry 1 w zapisie binarnym informują więc o tym, które wartości potęg liczby 2 są wykorzystane – zależy to od pozycji cyfry.

Zapis w systemie binarnym	Zapis w systemie dziesiętnym
1	$2^0 = 1$
10	$2^1 = 2$
100	$2^2 = 4$
1000	$2^3 = 8$
10000	$2^4 = 16$
100000	$2^5 = 32$
1000000	$2^6 = 64$
10000000	$2^7 = 128$
100000000	$2^8 = 256$
1000000000	$2^9 = 512$
10000000000	$2^{10} = 1024$

Tabela 2.1. Zapis binarny kolejnych potęg liczby 2

Warto wiedzieć

W systemie dwójkowym, podobnie jak w innych systemach pozycyjnych o podstawie większej niż 1, można dopisywać cyfry 0 na początku liczby. Nie wpływa to na wartość liczby, np.:
$10_2 = 010_2 = 0010_2$

Ćwiczenie 1

Podaj zapis dziesiętny liczby, która w systemie dwójkowym jest największą liczbą:

a. ośmiocyfrową,

b. szesnastocyfrową.

Jak wyznaczyć cyfry binarne liczby dziesiętnej? W poprzednim temacie rozpatrywaliśmy **problem podziału liczby na cyfry**. Jeśli w podanym tam algorytmie dzielenie przez 10 zastąpimy dzieleniem przez 2, otrzymamy reszty z dzielenia, które będą poszukiwanymi cyframi binarnymi. Wyniki kolejnych działań dla liczby 212 przedstawia tabela 2.2.

Problem 3. Podział liczby na cyfry, s. 12–13

Liczba	Część całkowita z dzielenia przez 2	Reszta z dzielenia przez 2
212	106	0
106	53	0
53	26	1
26	13	0
13	6	1
6	3	0
3	1	1
1	0	1

Tabela 2.2. Wyniki dzielenia prowadzące do wyznaczenia cyfr binarnych liczby 212

W trzeciej kolumnie tabeli otrzymujemy kolejno cyfry binarne liczby dziesiętnej 212 zapisane od najmniej znaczącej. Kiedy odczytamy je od końca, czyli od ostatniego wiersza, otrzymamy reprezentację binarną liczby 212 równą 11010100_2.

Ćwiczenie 2

Zapisz w wybranej przez siebie notacji (lista kroków, pseudokod lub schemat blokowy) algorytm wypisujący cyfry binarne liczby dziesiętnej w odwrotnej kolejności (najpierw cyfrę przy potędze 2^0). Następnie napisz program realizujący ten algorytm i sprawdź jego działanie dla różnych danych.

Warto wiedzieć

Innym sposobem zamiany reprezentacji liczby w systemie dziesiętnym na reprezentację w systemie binarnym jest określenie, które wagi systemu (wartości potęg 2) należy wybrać, aby otrzymać podaną liczbę dziesiętną, np.:
$84 = 1 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 0 \cdot 1$
Zatem:
$84 = 1010100_2$

A to ciekawe

0 i 1 na papierze

Na początku XIX w. nastoletni Louis Braille, który w dzieciństwie stracił wzrok, opracował system zapisu informacji z wykorzystaniem dwóch stanów: punktu wypukłego i niewypukłego. Poszczególne znaki przedstawił jako kombinacje punktów umieszczonych w dwóch kolumnach, po trzy punkty w każdej. Za inspirację posłużył Braille'owi system używany przez wojska francuskie do przekazywania rozkazów w ciemności. Dzięki wynalazkowi młodego Francuza, nazwanemu później alfabetem Braille'a, niewidomi mają większy dostęp do wiedzy. Pierwszą książkę zapisaną tym alfabetem wydano w 1837 r. Objętość książki brajlowskiej jest kilka razy większa od zapisanej zwykłym alfabetem.

System szesnastkowy i system ósemkowy

System szesnastkowy (heksadecymalny), system ósemkowy (oktalny)

Ponieważ zapis dużych liczb w systemie binarnym jest długi i mało czytelny, w informatyce wykorzystuje się także systemy o podstawach 16 i 8, czyli **system szesnastkowy (heksadecymalny)** i **system ósemkowy (oktalny)**.

W systemie ósemkowym używamy ośmiu cyfr (od 0 do 7). W systemie szesnastkowym mamy do dyspozycji 16 znaków – cyfry od 0 do 9 oraz sześć liter alfabetu łacińskiego: A, B, C, D, E, F. Te litery w systemie szesnastkowym także są nazywane cyframi.

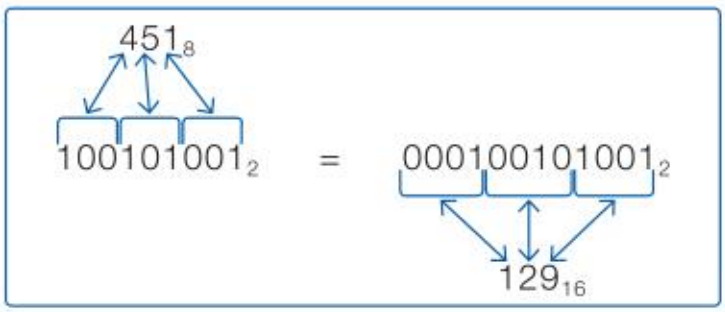
Liczba dziesiętna	Liczba dwójkowa	Liczba ósemkowa	Liczba szesnastkowa
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Tabela 2.3. Reprezentacja liczb dziesiętnych od 0 do 15 w systemach pozycyjnych o podstawach 2, 8 i 16

Warto wiedzieć

System szesnastkowy wykorzystuje się m.in. do adresowania komórek pamięci komputera i podawania adresów karty MAC. System ósemkowy jest używany znacznie rzadziej – stosują go np. użytkownicy systemu Linux przy ustawianiu praw dostępu do pliku.

Zwróć uwagę, że każdą cyfrę szesnastkową od 0 do F można zapisać za pomocą czterech cyfr binarnych oraz dwóch cyfr ósemkowych (mniejsze wartości uzupełniamy z lewej strony zerami nieznaczącymi), a każdą cyfrę ósemkową od 0 do 7 przy użyciu trzech cyfr binarnych. Korzystając z tych własności, można szybko zamieniać reprezentację liczb między systemami o podstawach 2, 8 i 16 bez pośrednictwa systemu dziesiętnego. Przykład takiej zamiany pokazuje rysunek 2.1.



Rys. 2.1. Zamiana zapisu liczby pomiędzy systemami ósemkowym i szesnastkowym

Ćwiczenie 3

Zapisz liczbę 10110011_2 w systemie szesnastkowym.

Wartość dziesiętną liczby zapisanej w systemie szesnastkowym możemy wyznaczyć podobnie jak w przypadku innych systemów pozycyjnych. Na przykład:

$A9F6_{16} = 10 \cdot 16^3 + 9 \cdot 16^2 + 15 \cdot 16^1 + 6 \cdot 16^0 = 40960 + 2304 + 240 + 6 = 43510$

Cyfry szesnastkowe liczby dziesiętnej można wyznaczyć tak jak cyfry binarne, korzystając z dzielenia przez podstawę systemu, czyli liczbę 16. Trzeba pamiętać, że reszcie z dzielenia równej 10 odpowiada cyfra A, reszcie 11 – cyfra B itd. Metodę wykorzystującą dzielenie z resztą przez podstawę systemu można zastosować przy zamianie liczby dziesiętnej na jej reprezentację w dowolnym systemie pozycyjnym. Tabela 2.4 przedstawia wyznaczanie cyfr liczby dziesiętnej 212 w systemie szesnastkowym. Wynika z niej, że $212 = D4_{16}$.

Warto wiedzieć

W niektórych urządzeniach elektronicznych z wyświetlaczem cyfrowym stosowany jest kod BCD (ang. *binary-coded decimal*), w którym kodowane dwójkowo są cyfry dziesiętne liczby, a nie cała liczba. Do zakodowania jednej cyfry dziesiętnej w tym kodzie wystarczają 4 cyfry dwójkowe.

Liczba	Część całkowita z dzielenia przez 16	Reszta z dzielenia przez 16
212	13	4
13	0	13 (D)

Tabela 2.4. Wyniki dzielenia prowadzące do wyznaczenia cyfr szesnastkowych liczby dziesiętnej 212

Zapamiętaj

Systemy pozycyjne o podstawie będącej potęgą liczby 2 mają szczególne znaczenie dla informatyki – zwłaszcza system dwójkowy (binarny) i szesnastkowy (heksadecymalny). Można łatwo zamieniać reprezentację liczb między tymi systemami bez używania systemu dziesiętnego. Korzystamy przy tym z faktu, że cztery cyfry binarne odpowiadają jednej cyfrze szesnastkowej.

2.2. Reprezentacja liczb całkowitych w komputerze

Bit ● **Bit** (oznaczany przez b) jest najmniejszą i niepodzielną porcją informacji. Pojedynczy bit może reprezentować jeden z dwóch stanów fizycznych (np. elektrycznych, magnetycznych), oznaczanych umownie przez 0 lub 1. Inną jednostką informacji jest **bajt** (oznaczany przez B). Współcześnie przyjmuje się, że jeden bajt jest równy 8 bitom. Zatem za pomocą bajta można przedstawić 2^8, czyli 256 różnych informacji. Uogólniając: ciąg n bitów pozwala zapisać 2^n symboli (znaków lub liczb). Jeśli za pomocą jednego bajta chcemy zapisać liczbę całkowitą bez znaku, to możemy zapamiętać w ten sposób liczbę z zakresu od 0 do 255 (tabela 2.5).

Bajt ●

Warto wiedzieć

Termin 1 bajt jako porcja informacji odpowiadająca 8 bitom upowszechnił się po roku 1964, kiedy firma IBM rozpoczęła produkcję komputerów System/360 przetwarzających 8-bitowe porcje danych.

Wartość dziesiętna	Jednobajtowa (ośmiobitowa) liczba całkowita bez znaku
0	00000000
1	00000001
2	00000010
...	...
127	01111111
128	10000000
129	10000001
...	...
253	11111101
254	11111110
255	11111111

Tabela 2.5. Zapis liczb całkowitych z zakresu od 0 do 255 w jednym bajcie

Warto wiedzieć

Jednostki pojemności pamięci w informatyce tworzy się podobnie jak jednostki znane z matematyki czy fizyki. Do nazwy podstawowej jednostki dodaje się odpowiedni przedrostek, np. kilo-, giga-. W informatyce jednak mnożnikiem jest odpowiednia potęga liczby 2, a nie liczby 10, np.:
1 kilobit (1 Kb) to 2^{10} b = = 1024 b
1 megabit (1 Mb) to 2^{20} b = = 1024 Kb
1 gigabit (1 Gb) to 2^{30} b = = 1024 Mb
1 terabit (1 Tb) to 2^{40} b = = 1024 Gb
Podobnie tworzy się jednostki będące wielokrotnościami bajta, np. 1 KB, 1 MB, 1 GB, 1 TB.

Jeśli chcemy reprezentować także liczby ujemne, jeden bit należy poświęcić na zapamiętanie znaku liczby. Wówczas za pomocą jednego bajta możemy przedstawić liczby z zakresu od -2^7 do $2^7 - 1$ (czyli od −128 do 127: liczby ujemne od −128 do −1, liczby nieujemne od 0 do 127). Tak mały zakres liczb w większości przypadków jest niewystarczający. Dlatego do reprezentacji liczb całkowitych wykorzystuje się więcej niż jeden bajt.

Ćwiczenie 4

a. Jaką największą liczbę całkowitą bez znaku można zapisać z użyciem 2 bajtów?

b. Jaka jest minimalna liczba bitów potrzebnych do zapisania liczby dziesiętnej 78?

Kod uzupełnień do 2 (U2)

W przypadku reprezentacji liczb ze znakiem na zapisanie znaku liczby przeznaczony jest **najbardziej znaczący bit**, zwany też **najstarszym bitem**. Jest to bit znajdujący się najbardziej z lewej strony. Bit znajdujący się najbardziej z prawej strony to **najmniej znaczący bit** (rys. 2.2).

- **Bit najbardziej znaczący (najstarszy)**
- **Bit najmniej znaczący**

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

Rys. 2.2. Numeracja bitów od najbardziej znaczącego (b_7) do najmniej znaczącego (b_0)

Najstarszy bit dla liczby nieujemnej ma wartość 0, a w przypadku liczby ujemnej 1. Na przykład liczby +15 i −15 zapamiętane z wykorzystaniem 8 bitów w **naturalnym zapisie binarnym** będą miały postać:

- **Naturalny zapis binarny** – zapis, w którym najstarszy bit jest bitem znaku, pozostałe bity pamiętają wartość bezwzględną liczby.

$+15 = 00001111_2$
$-15 = 10001111_2$

Taki zapis utrudnia jednak wykonywanie działań arytmetycznych. Pokażemy to na przykładzie dodawania liczb binarnych.

Weźmy dwie liczby dodatnie: 10 i 15. Zamieńmy je na postać binarną, a następnie dodajmy pisemnie. Powinniśmy otrzymać wynik 11001_2 (w zapisie dziesiętnym to liczba 25). Pisemne dodawanie liczb w systemie binarnym wykonujemy podobnie jak dodawanie w systemie dziesiętnym. W dodawaniu binarnym 0 + 0 = 0, 0 + 1 = 1, natomiast 1 + 1 = 10, co powoduje przeniesienie 1.

```
  111
   1010
 + 1111
 -------
  11001
```

Jak widzisz, w przypadku liczb dodatnich otrzymaliśmy oczekiwany wynik. Jeśli analogicznie dodalibyśmy liczbę 1 zapisaną przy użyciu 8 bitów (reprezentacja binarna to 00000001_2) do liczby ujemnej −1 (reprezentacja binarna to 10000001_2), otrzymalibyśmy liczbę 10000010_2, czyli −2 w systemie dziesiętnym. Żeby uzyskać właściwy wynik, należałoby sprawdzać oddzielnie znaki liczb.

Aby można było prawidłowo wykonywać działania na liczbach ujemnych (traktując znak liczby jako jej integralną cześć), wprowadzono specjalne kodowanie. Liczby całkowite w komputerze zapisywane są w **kodzie uzupełnień do 2** (w skrócie **U2**). Dla liczb nieujemnych zapis w kodzie U2 odpowiada naturalnemu zapisowi liczb binarnych.

- **Kod uzupełnień do 2 (U2)**

Aby przedstawić w kodzie U2 liczbę ujemną, należy najpierw dodać do niej wartość podwojonej wagi najstarszego bitu. Na przykład jeśli liczbę −1 chcemy zapisać w U2 za pomocą 8 bitów, dodajemy do niej liczbę 2^8 (najstarszy bit ma wagę 2^7, natomiast podwojona wartość bitu jest równa 2^8, bo $2 \cdot 2^7 = 2^8$). $2^8 = 256$, więc otrzymujemy: $-1 + 256 = 255$. Liczba −1 będzie zatem odpowiadać w kodzie U2 liczbie dziesiętnej 255, czyli liczbie binarnej bez znaku 11111111_2.

Warto wiedzieć

W naturalnym zapisie binarnym z uwzględnieniem liczb ujemnych liczba zero ma dwie reprezentacje. Na przykład w 8 bitach zero można zapisać jako 10000000_2 oraz 00000000_2. Przy wykorzystaniu obu reprezentacji liczby zero, za pomocą ośmiu bitów nie można reprezentować 256 różnych liczb, tylko 255.

Zatem:

$1 = 00000001_{U2}$

$-1 = 11111111_{U2}$

Po pisemnym dodaniu liczb 1 i –1 zapisanych w kodzie U2 (przeniesienie z najstarszego bitu pomijamy) otrzymujemy liczbę binarną 00000000, czyli oczekiwany wynik dziesiętny 0.

```
 1111111
  00000001
+ 11111111
  00000000
```

Ćwiczenie 5

Zapisz poniższe liczby dziesiętne w kodzie U2 z wykorzystaniem jednego bajta.

a. –100

b. 56

c. –15

Dobra rada

Podczas kodowania liczb pamiętaj, że kodowane wartości będą różnie wyglądały, gdy na ich reprezentację przeznaczysz różną liczbę bitów.

Należy pamiętać, że bit znaku liczby w kodzie U2 traktujemy jako integralną część reprezentacji liczby. Tabela 2.6 przedstawia reprezentację binarną wybranych liczb całkowitych ze znakiem, zapisanych w kodzie uzupełnień do 2 za pomocą jednego bajta.

Wartość dziesiętna	Zapis w kodzie U2
–128	10000000
–127	10000001
–126	10000010
…	…
–2	11111110
–1	11111111
0	00000000
1	00000001
2	00000010
…	…
125	01111101
126	01111110
127	01111111

Tabela 2.6. Zapis w kodzie U2 liczb dziesiętnych z zakresu od –128 do 127 z wykorzystaniem 1 bajta

Wiesz już, jak zapisać liczbę całkowitą ze znakiem, wykorzystując 8 bitów. W ogólnym przypadku, gdy liczba całkowita x jest reprezentowana za pomocą n bitów (najstarszy bit, czyli b_{n-1}, przeznaczony jest na znak liczby), jej wartość dziesiętną w kodzie U2 obliczysz ze wzoru:

$$x_{U2} = \begin{cases} x & \text{dla} \quad x \geqslant 0 \\ 2^n + x & \text{dla} \quad x < 0 \end{cases}$$

Obliczoną wartość dziesiętną trzeba zamienić na liczbę binarną.

Jeśli mamy liczbę zapisaną w kodzie U2 na n bitach, jej wartość dziesiętną możemy wyznaczyć według następującego schematu:

$$-b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + \ldots + b_1 \cdot 2^1 + b_0 \cdot 2^0$$

Na przykład:

$10101010_{U2} = -1 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 =$
$= -128 + 32 + 8 + 2 = -86$

Ćwiczenie 6

Podaj wartość dziesiętną liczby 11010101_{U2}.

Zapamiętaj

Dany układ bitów może być interpretowany jako liczba ze znakiem lub bez znaku. Liczby całkowite ze znakiem są reprezentowane w pamięci komputera w kodzie uzupełnień do 2 (U2). Umożliwia to wykonywanie działań na liczbach ujemnych. Bit znaku liczby jest traktowany jako integralna część zapisu liczby: znak zapamiętujemy w najstarszym bicie, przy czym 0 oznacza +, a 1 oznacza –. Podczas pisemnego dodawania liczb zapisanych w kodzie U2 przeniesienie z najstarszego bitu jest pomijane.

Wybrane typy danych w języku C++ umożliwiające zapisywanie liczb całkowitych

W programach napisanych do tej pory stosowaliśmy typ `int`, reprezentujący liczby całkowite ze znakiem. Do zapamiętania zmiennych tego typu wykorzystywane są 4 bajty (32 bity), a więc można reprezentować 2^{32} liczb z zakresu od -2^{31} do $2^{31} - 1$. To zakres, który w większości sytuacji jest wystarczający, dlatego w programach omawianych w tym podręczniku najczęściej będziemy używać tego typu.

Tabela 2.7 na s. 36 zawiera informacje o wybranych typach danych wykorzystywanych w języku C++ do zapisu liczb całkowitych ze znakiem dostępnych w kompilatorach 32-bitowych. Należy zwrócić uwagę, że w zależności od kompilatora C++ oraz wersji systemu (32-bitowy lub 64-bitowy) zakresy te (liczba bajtów przeznaczonych do reprezentacji poszczególnych typów) mogą się różnić.

Warto wiedzieć

W językach programowania zbiór liczb całkowitych jest ograniczony, a więc można podać największą liczbę całkowitą (zależy ona od liczby bajtów przeznaczonych do reprezentacji liczby). Wyjątkiem jest język Python, w którym zakres liczb całkowitych jest ograniczony pamięcią komputera dostępną dla programu, a więc w praktyce nieograniczony.

Pełna nazwa typu	Skrót nazwy typu	Liczba bajtów	Zakres
`short int`	`short`	2	od -2^{15} do $2^{15}-1$ czyli od –32 768 do 32 767
`int`		4	od -2^{31} do $2^{31}-1$ czyli od –2 147 483 648 do 2 147 483 647
`long int`	`long`	4	od -2^{31} do $2^{31}-1$ czyli od –2 147 483 648 do 2 147 483 647
`long long int`	`long long`	8	od -2^{63} do $2^{63}-1$ czyli od –9 223 372 036 854 775 808 do 9 223 372 036 854 775 807

Tabela 2.7. Wybrane typy służące do zapisu liczb całkowitych ze znakiem dostępne w języku C++ oraz ich zakresy dla kompilatorów 32-bitowych

Dobra rada

Przy pisaniu programów wykorzystujących zmienne typu całkowitego zwracaj uwagę na to, czy typ całkowity wybrany dla zmiennych ma wystarczający zakres.

Dodanie słowa kluczowego `unsigned` przed nazwą typu spowoduje, że dany typ będzie oznaczał liczby całkowite bez znaku, a więc zakres liczb dodatnich dwukrotnie się powiększy. Na przykład typ `unsigned int` będzie reprezentował liczby z zakresu od 0 do $2^{32}-1$, czyli od 0 do 4 294 967 295.

Zwrócenie przez program innego wyniku niż oczekiwany może być spowodowane tym, że został przekroczony zakres danych. Jeśli np. po zsumowaniu liczb ujemnych program zwróci liczbę dodatnią, należy sprawdzić, czy zmiennym przypisano właściwy typ danych.

Ćwiczenie 7

Napisz program, który zmiennej typu całkowitego przypisze wartość największej liczby całkowitej tego typu, a następnie doda do tej zmiennej 1 i wypisze jej wartość. Wykonaj ćwiczenie dla:

a. typu reprezentującego liczby całkowite bez znaku,

b. typu reprezentującego liczby całkowite ze znakiem.

Sprawdź działanie programu i wyjaśnij, dlaczego zwrócił taki wynik.

2.3. Reprezentacja znaków i napisów

Kod ASCII

Pojedyncze znaki (litery, cyfry i inne znaki) kodowane są w pamięci komputera za pomocą liczb. Najczęściej stosuje się **kod ASCII** (ang. *American Standard Code for Information Interchange*). Standard ASCII jest kodem siedmiobitowym, określa więc 128 znaków o kodach od 0 do 127. Wersja rozszerzona, ośmiobitowa, uwzględnia 256 znaków – znaki kodowane są liczbami z zakresu od 0 do 255.

Tabela 2.8 przedstawia **tablicę kodów ASCII** z zakresu od 32 do 127. Kody od 0 do 31 oraz 127 są zarezerwowane na tzw. **znaki sterujące**, nazywane też **kodami sterującymi** (ang. *control character, control code*). W wersji ośmiobitowej kody od 128 do 255 reprezentują różne znaki w zależności od systemu operacyjnego i wersji językowej. Polskie znaki diakrytyczne mają kody o wartościach powyżej 127 i w zależności od zastosowanej strony kodowej mogą mieć różne wartości.

Tablica kodów ASCII

Znak sterujący (kod sterujący) – znak, który nie jest znakiem drukowanym, służy do sterowania urządzeniem przetwarzającym dane, np. drukarką.

Kod dziesiętny	Znak	Kod dziesiętny	Znak	Kod dziesiętny	Znak	Kod dziesiętny	Znak
32	spacja	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	Del

Tabela 2.8. Tablica kodów ASCII z zakresu od 32 do 127

Dobra rada

W programach w języku C++ wypisujących napisy za pomocą instrukcji `cout` unikaj polskich znaków diakrytycznych. W edytorze tekstu i w oknie wyników (aplikacji konsolowej) mają one różne kody.

Warto wiedzieć

Standard ASCII został opracowany na podstawie kodu telegraficznego przez American National Standards Institute (ANSI) – instytut ustalający normy techniczne obowiązujące w USA. Prace rozpoczęły się w 1960 r., a pierwszy standard opublikowano w 1963 r.

W języku C++ znaki możemy reprezentować za pomocą typu `char`. Zmienne tego typu zajmują jeden bajt pamięci. Typ `char` reprezentuje też liczby jednobajtowe ze znakiem (z zakresu od –128 do 127). Do reprezentacji napisów, czyli ciągów znaków, w języku C++ służy typ `string`. Aby pokazać przykład użycia typów `char` i `string`, napiszemy prosty program zliczający wystąpienia litery w słowie.

Warto wiedzieć

Oprócz typu `char` można też używać typu `unsigned char`, reprezentującego liczby jednobajtowe bez znaku (od 0 do 255).

Warto wiedzieć

Słowo (nazwa typu) `string` (zdefiniowane w bibliotece `string`) nie jest słowem kluczowym języka C++, a więc można je przedefiniować.

Specyfikacja

Dane: `slowo` – słowo złożone z liter alfabetu łacińskiego, `litera` – litera alfabetu łacińskiego.

Wynik: liczba wystąpień litery w słowie.

Aby policzyć wystąpienia litery w słowie, musimy po kolei sprawdzać wszystkie litery słowa i jeśli aktualna litera jest taka sama jak ta, której wystąpienia zliczamy, powiększać licznik o jeden. Na początku licznik trzeba wyzerować. Oto zapis algorytmu w pseudokodzie:

```
ile ← 0
dla i ← 0, 1, ..., długość słowa - 1 wykonuj
    jeśli slowo[i] = litera to ile ← ile + 1
```

Indeks

Do poszczególnych znaków napisu odwołujemy się za pośrednictwem **indeksu**, czyli numeru elementu (tutaj numeru znaku w napisie). Indeks podajemy w nawiasach kwadratowych (`slowo[i]`). Na przykład dla wartości zmiennej `slowo` równej „mak": zapis `slowo[0]` odpowiada literze „m", `slowo[1]` – literze „a", natomiast `slowo[2]` – literze „k". Indeksowanie zaczyna się zawsze od zera, więc indeks ostatniego znaku napisu jest o jeden mniejszy od długości napisu. Oto kod źródłowy programu dla powyższej specyfikacji:

Kod źródłowy programu zliczającego wystąpienia litery w słowie

```
1.  #include <iostream>
2.  #include <string>
3.
4.  using namespace std;
5.
6.  int main()
7.  {
8.      string slowo;
9.      char litera;
10.     int i, ile=0;
11.     cout<<"Podaj slowo: "; cin>>slowo;
12.     cout<<"Podaj litere: "; cin>>litera;
13.     for (i=0;i<slowo.size();i++)
14.         if (slowo[i]==litera) ile++;
15.     cout<<"Litera "<<litera<<" wystepuje ";
16.     cout<<"w slowie "<<slowo<<" "<<ile<<" razy.";
17.     return 0;
18. }
```

W liniach 8–10 zadeklarowane są zmienne, odpowiednio: zmienna `slowo` typu `string`, zmienna `litera` typu `char`, a także zmienne `ile` oraz `i` typu `int`. Zmiennej `ile` nadana jest od razu wartość początkowa `0`. W liniach 11–12 program wczytuje dane wejściowe poprzedzone odpowiednimi napisami zachęty na ekranie.

Typ `string` jest typem obiektowym (klasą). Nie tylko umożliwia pamiętanie danych, lecz także udostępnia funkcje do ich przetwarzania. Funkcje dostępne w danej klasie nazywamy **metodami**. Gdy korzystamy z danej metody, stosujemy notację z kropką, tzn. `nazwa_zmiennej.nazwa_metody`. Aby było możliwe wykorzystywanie metod zdefiniowanych dla klasy `string`, w linii 2 dodaliśmy dyrektywę `#include <string>`.

Metoda

W zasadniczej części programu (linie 13–14) użyliśmy **metody `size`**, dostępnej w klasie `string`. Wykorzystaliśmy ją w warunku pętli dla zmiennej `slowo`. Wartością tej funkcji jest długość (liczba znaków) napisu. W pętli przeglądane są znaki, z których składa się słowo. Po napotkaniu znaku równego wartości zmiennej `litera` wartość zmiennej `ile` jest powiększana o jeden.

Metoda `size`

Zwróć uwagę na linie 15 i 16, które odpowiadają za wypisanie wyniku. Wielokrotnie jest tu łączone wypisanie stałego napisu na ekranie z wartościami zmiennych. Stałe napisy zapisane są w cudzysłowach. Pojedyncze stałe znakowe ujęlibyśmy w apostrofy (nie występują w tym przykładzie).

Warto wiedzieć

Zamiast metody `size` można użyć metody `length`.

Ćwiczenie 8

Napisz program, który po podaniu z klawiatury słowa oraz litery wyświetli na ekranie liczbę wystąpień litery w podanym słowie (można używać tylko liter alfabetu łacińskiego). Sprawdź działanie programu.

Podjęto próby określenia uniwersalnego sposobu kodowania znaków obejmującego wszystkie alfabety. W efekcie powstał **Unicode**. Uzyskanie binarnego zapisu kodu w Unicode wymaga użycia tzw. schematu kodowania. Przykładem takiego schematu jest **UTF-8**. Stosuje się go np. na stronach internetowych. W UTF-8 znaki są kodowane na 1–4 bajtach pamięci. Schemat ten jest wstecznie kompatybilny z kodem ASCII, tzn. wszystkie znaki standardu ASCII mają w Unicode taki sam kod jak w ASCII i zajmują 1 bajt pamięci. Jeśli chcesz się odwoływać w swoich programach do kodów Unicode, możesz użyć typu `wchar_t`.

Unicode

UTF-8

Warto wiedzieć

Nazwa UTF-8 pochodzi od angielskiej nazwy: *8-bit Unicode Transformation Format*.

Zapamiętaj

Znaki w komputerze reprezentowane są za pomocą kodów liczbowych. Standard ASCII określa znaki o kodach od 0 do 127, przy czym wielkie litery łacińskie mają kody od 65 do 90, małe od 97 do 122, a znaki cyfr od 48 do 57. W rozszerzonej wersji ASCII kody znaków od 128 do 255 są różne w zależności od systemu operacyjnego i zastosowanej strony kodowej. Polskie znaki diakrytyczne mogą mieć różne kody w zależności od użytej strony kodowej, ale zawsze kody te są większe od 127.

2.4. Operacje logiczne, reprezentacja wartości logicznych

Zdanie logiczne

Zdaniem logicznym jest zdanie, któremu możemy przyporządkować jednoznacznie jedną z dwóch wartości logicznych: **`prawda`** lub **`fałsz`**. Przyjmuje się, że **`prawda`** oznaczana jest wartością 1, a **`fałsz`** – wartością 0. Zdania logiczne możemy łączyć w zdania złożone.

Koniunkcja

Koniunkcja jest połączeniem zdań logicznych p, q spójnikiem „i". Żeby zdanie „*p i q*" było prawdziwe, każde ze zdań składowych musi być prawdziwe.

p	q	p i q
0	0	0
0	1	0
1	0	0
1	1	1

Tabela 2.9. Wartości logiczne koniunkcji zdań p, q

Alternatywa

Alternatywa jest połączeniem zdań logicznych p, q spójnikiem „lub". Żeby zdanie „*p lub q*" było prawdziwe, jedno ze zdań składowych musi być prawdziwe.

p	q	p lub q
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 2.10. Wartości logiczne alternatywy zdań p, q

Negacja (zaprzeczenie)

Negacja (zaprzeczenie) to zdanie logiczne, które ma postać „nieprawda, że *p*", gdzie p jest zdaniem logicznym. Zaprzeczenie zdania p zapisuje się często jako $\sim p$.

p	$\sim p$
0	1
1	0

Tabela 2.11. Wartości logiczne negacji zdania p

Warto wiedzieć

W języku C++ wartości zmiennych typu całkowitego mogą być interpretowane jako wartości logiczne. Wówczas 0 jest interpretowane jako **`false`**, a liczby różne od zera jako **`true`**.

W języku C++ dostępne są dwie wartości logiczne: **`true`** (prawda) i **`false`** (fałsz). Wartość **`false`** jest reprezentowana przez liczbę 0, a **`true`** przez 1. Typ logiczny reprezentujący wartości logiczne to **`bool`**.

Spójnik logiczny (operator logiczny)

Spójniki logiczne, nazywane też **operatorami logicznymi**, umożliwiają budowanie wyrażeń logicznych w programach. Tabela 2.12 przedstawia zapis operatorów logicznych w pseudokodzie i języku C++ oraz przykłady ich użycia w języku C++.

Nazwa operatora logicznego	Zapis operatora w pseudokodzie	Zapis operatora w języku C++	Przykład użycia w języku C++	
			Wyrażenie	Wartość logiczna wyrażenia dla przykładowych danych
Koniunkcja	`oraz`	`&&` lub `and`	`x<5 && y>1`	Dla $x = 3$ i $y = 2$ wartością wyrażenia będzie `true`, a dla $x = 6$ i $y = 2$ `false`.
Alternatywa	`lub`	`||` lub `or`	`x<5 || y>1`	Dla $x = 6$ i $y = 2$ wartością wyrażenia będzie `true`, a dla $x = 6$ i $y = 1$ `false`.
Negacja	`nie`	`!` lub `not`	`!(x==4)`	Dla $x = 3$ wartością wyrażenia będzie `true`, dla $x = 4$ `false`.

Tabela 2.12. Zapis operatorów logicznych w pseudokodzie i języku C++ oraz przykłady ich użycia w języku C++

Rozważmy następujący problem. Chcemy sprawdzić, czy z trzech odcinków o określonej długości można zbudować trójkąt.

Specyfikacja

Dane: a, b, c – liczby całkowite dodatnie.

Wynik: słowo „TAK", gdy z odcinków o długościach a, b, c można zbudować trójkąt, słowo „NIE" – w przeciwnym przypadku.

Warunek istnienia trójkąta możesz pamiętać ze szkoły podstawowej: suma długości dwóch krótszych boków musi być większa od długości najdłuższego. Ponieważ nie wiemy, który z boków jest najdłuższy, warunek sprawdzimy dla każdego z nich. Zapis algorytmu w pseudokodzie i jego realizacja w języku C++ mogą wyglądać tak jak poniżej.

```
jeśli a + b > c oraz a + c > b oraz b + c > a to wypisz "TAK"
w przeciwnym przypadku wypisz "NIE"
```

Kod źródłowy programu sprawdzającego, czy z trzech odcinków można zbudować trójkąt

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  int main()
6.  {
7.      int a, b, c;
8.      cout<<"a = "; cin>>a;
9.      cout<<"b = "; cin>>b;
10.     cout<<"c = "; cin>>c;
11.     if (a+b>c && a+c>b && b+c>a) cout<<"TAK";
12.     else cout<<"NIE";
13.     return 0;
14. }
```

Warto wiedzieć

Warunek `!(x==4)` jest równoważny warunkowi `x!=4`.

Warto wiedzieć

Nazwa typu `bool` pochodzi od nazwiska angielskiego matematyka, logika i filozofa George'a Boole'a. Żył on w latach 1815–1864 i wniósł znaczący wkład w rozwój logiki matematycznej.

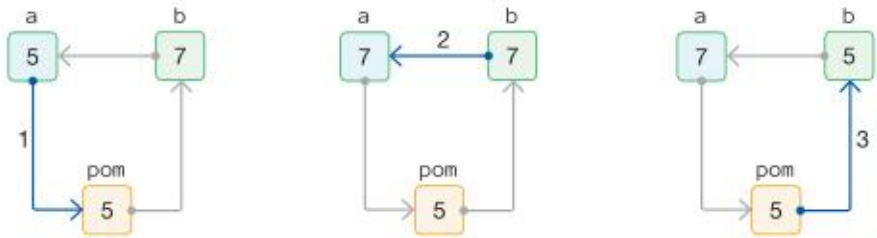
Dobra rada

Pamiętaj, że w języku C++ wyrażenia logiczne w warunkach logicznych ujmujemy w nawiasy okrągłe.

Instrukcja warunkowa w kodzie programu sprawdzającego, czy z danych odcinków można zbudować trójkąt, byłaby prostsza, gdybyśmy znaleźli największą liczbę wśród podanych i zapamiętali ją w konkretnej zmiennej, np. a. Można do takiej sytuacji doprowadzić, jeśli zamienimy odpowiednio wartości zmiennych. Do zamiany wartości dwóch zmiennych można użyć trzeciej zmiennej – pomocniczej.

Zamiana wartości zmiennych

Rysunek 2.3 przedstawia **zamianę wartości zmiennych** a (początkowa wartość 5) i b (początkowa wartość 7). W pierwszym kroku zmienna pomocnicza pom przyjmuje wartość zmiennej a, w drugim kroku zmienna a przyjmuje wartość zmiennej b, a w trzecim kroku zmienna b – wartość przechowaną w zmiennej pomocniczej.



Rys. 2.3. Ilustracja zamiany wartości zmiennych a i b

W pseudokodzie zapis algorytmu sprawdzającego, czy z podanych trzech odcinków można zbudować trójkąt, uwzględniającego znalezienie największej liczby, może wyglądać następująco:

```
jeśli b > a to
    pom ← a; a ← b; b ← pom
jeśli c > a to
    pom ← a; a ← c; c ← pom
jeśli b + c > a to wypisz "TAK"
w przeciwnym przypadku wypisz "NIE"
```

Dobra rada

W języku C++ do zamiany wartości dwóch zmiennych możesz użyć funkcji `swap`.

Ćwiczenie 9

Utwórz program sprawdzający warunek istnienia trójkąta:
a. z wykorzystaniem koniunkcji,
b. według podanego powyżej pseudokodu.
Sprawdź działanie programu dla różnych danych.

Zapamiętaj

Zdaniem logicznym nazywamy zdanie, któremu możemy przyporządkować jednoznacznie jedną z dwóch wartości: **prawda** lub **fałsz**. Zdania logiczne możemy łączyć. W języku C++ dostępne są dwie wartości logiczne: `true` (prawda) i `false` (fałsz). Wartość `false` jest reprezentowana przez liczbę 0, a `true` przez 1. Typ logiczny reprezentujący wartości logiczne to `bool`.

2.5. Reprezentacja obrazów, dźwięków i animacji

Obrazy, dźwięki i animacje, podobnie jak inne informacje, pamiętane są w komputerze w formie cyfrowej – w postaci zer i jedynek.

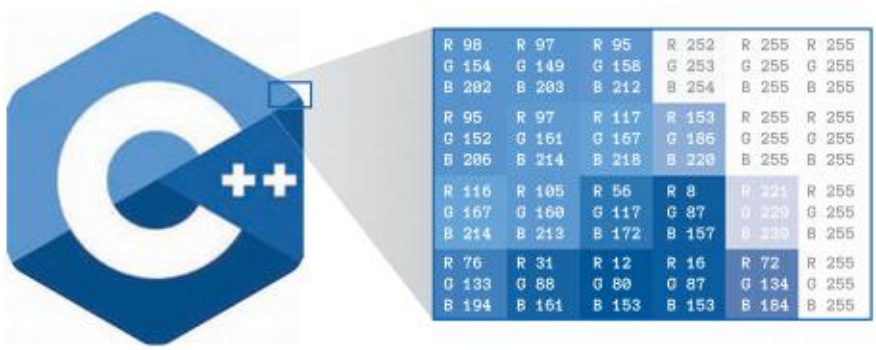
Reprezentacja obrazów

Obraz na ekranie monitora składa się z pojedynczych punktów zwanych pikselami. Do reprezentacji koloru pojedynczego piksela zazwyczaj wykorzystuje się **model RGB** – nasycenie trzema podstawowymi kolorami: czerwonym (ang. *red*), zielonym (ang. *green*) i niebieskim (ang. *blue*). W modelu RGB najczęściej stosuje się 24-bitowy zapis kolorów, w którym na każdą ze składowych przeznacza się 8 bitów. Poszczególne składowe opisuje się za pomocą liczby dziesiętnej od 0 do 255, przy czym pierwsza składowa oznacza poziom jasności koloru czerwonego, druga – zielonego, a trzecia – niebieskiego. W ten sposób można reprezentować ponad 16 mln barw (256 · 256 · 256). Maksymalne natężenie składowych odpowiada za kolor biały, a brak natężenia – za kolor czarny. Tak więc w RGB zapis (255, 255, 255) oznacza biel, a zapis (0, 0, 0) – czerń.

Model RGB

Warto wiedzieć

Podczas tworzenia obrazów przeznaczonych do druku stosuje się model CMYK, który wykorzystuje cztery kolory podstawowe: odcień niebieskiego (cyjan), purpurowy (magenta), żółty i czarny. W tym modelu dodanie czerni pozwala uzyskać lepszy kontrast.



Rys. 2.4. Składowe pikseli fragmentu obrazu w modelu RGB

Często na reprezentację pojedynczego piksela przeznaczony jest jeszcze jeden bajt pamięci, określający przezroczystość punktu, tzw. **kanał alfa**. Taki model nazywamy **modelem RGBA**.

Kanał alfa

Model RGBA

Pliki przechowujące obrazy w modelu RGB lub RGBA bez kompresji zwykle zajmują dużą objętość (np. pliki w formacie BMP). W związku z tym często (szczególnie na stronach internetowych) stosuje się formaty dokonujące kompresji obrazu, zarówno stratnej (np. format JPG), jak i bezstratnej (np. format PNG).

Reprezentacja dźwięków

Zamianę danych z postaci analogowej na postać cyfrową nazywamy **digitalizacją** (**dyskretyzacją**). Digitalizacja dźwięku polega na tzw. **próbkowaniu** (ang. *sampling*), czyli pobieraniu w równych odstępach czasu wartości poziomów sygnału dźwiękowego i zapisywaniu ich w formie cyfrowej. Każdy pomiar wartości nazywany jest **próbką**.

Digitalizacja (dyskretyzacja)

Próbkowanie

Próbka

Częstotliwość próbkowania

Rozdzielczość próbkowania

Kodek

Jakość dźwięku zależy od **częstotliwości próbkowania** (czyli liczby próbek pobieranych w ciągu sekundy) oraz dokładności pomiaru cyfrowego próbki, nazywanej **rozdzielczością próbkowania**. Rozdzielczość to liczba bitów przeznaczonych na zapis jednej próbki. Programy, które zamieniają dźwięk na zapis cyfrowy i odwrotnie (czyli kodują go i odkodowują), nazywamy **kodekami**.

Istnieją różne formaty plików dźwiękowych. Wybrane przedstawione są w tabeli 2.13.

Warto wiedzieć

Ludzkie ucho zwykle odbiera dźwięki w zakresie częstotliwości od ok. 16 Hz do ok. 20 kHz.

Format pliku dźwiękowego	Cechy
WAVE (WAV)	Pozwala na zapis dźwięku w jakości zbliżonej do płyt CD. Wadą tego formatu jest duża objętość pliku, która może np. utrudniać przesyłanie go przez internet.
MP3	W tym formacie zapisywane są tylko dźwięki istotne dla ludzkiego ucha, pozostałe są redukowane, czyli dźwięk jest poddawany kompresji. Przy zastosowaniu średniej kompresji utrata jakości jest niesłyszalna. Zaletą tego typu plików jest ich mała objętość.
MIDI (MID)	Umożliwia zapis cyfrowy partytury (zapisu nutowego). Pliki tego formatu charakteryzują się małą objętością, ale reprezentują tylko wybrany zbiór instrumentów. Nie można w nich zapisać ludzkiego głosu.

Tabela 2.13. Cechy wybranych formatów plików dźwiękowych

Reprezentacja animacji

Klatka

Liczba klatek na sekundę (fps)

Ogólna zasada tworzenia animacji jest bardzo prosta. Żeby powstało wrażenie ruchu, należy wyświetlać z odpowiednią częstotliwością kolejne obrazy, nazywane **klatkami** (ang. *frames*), które zazwyczaj niewiele się różnią. Na jakość obrazu animowanego składają się rozdzielczość pojedynczej klatki oraz **liczba klatek wyświetlanych na sekundę – fps** (ang. *frames per second*).

Warto wiedzieć

Filmy w kinach wyświetla się najczęściej z częstotliwością 24 klatek na sekundę.

Obrazy animowane pochłaniają dużo pamięci. W związku z tym stosuje się różne algorytmy kompresji, aby pliki miały mniejszą objętość, ale utrata jakości nie była zauważalna dla ludzkiego oka. Na przykład, ponieważ klatki zwykle znacznie się nie różnią, w pliku po kompresji zachowuje się tylko klatki kluczowe oraz informacje o różnicach między klatkami.

Podobnie jak w przypadku dźwięków, do zapisu i odczytu plików wideo potrzebne są specjalne programy nazywane kodekami. Popularne formaty zapisu tego typu plików to AVI, MOV, MP4.

Do zapisu animacji powszechnie stosuje się format GIF. Dzięki zastosowaniu kompresji bezstratnej pliki mają małą objętość, dlatego format ten wykorzystuje się np. przy umieszczaniu animacji na stronach internetowych.

Jak dane analogowe są przekształcane na cyfrowe

Podczas nagrywania filmu smartfonem dźwięk i ruchomy obraz są przetwarzane na postać cyfrową. Dzięki temu możemy przechowywać je w pamięci telefonu, a następnie odtwarzać. Jak działają cyfrowe urządzenia rejestrujące dźwięk i obraz? Przeanalizujmy to na przykładzie.

Mikrofon wbudowany w telefon zamienia falę dźwiękową na sygnały elektryczne odpowiadające amplitudzie i częstotliwości dźwięku.

Przetwornik A/C (analogowo-cyfrowy) próbkuje amplitudy sygnałów elektrycznych, czyli mierzy je tysiące razy w ciągu jednej sekundy i zamienia na ciągi liczb binarnych. Aby zapis cyfrowy był zgodny z dźwiękiem, który słyszy ludzkie ucho, próbkowanie musi się odbywać z odpowiednio wysoką częstotliwością i rozdzielczością. Na przykład dla płyt CD standardem są częstotliwość 44,1 kHz i rozdzielczość 16 bitów na próbkę, czyli 2^{16} = 65 536 możliwych wartości amplitudy fali dźwiękowej.

Podsumowanie

- Wszystkie dane w komputerze są pamiętane i przetwarzane w postaci dwóch stanów fizycznych – reprezentowanych umownie za pomocą cyfr 0 i 1.
- Szczególne znaczenie w informatyce mają systemy pozycyjne o podstawach 2 i 16, nazywane odpowiednio systemami dwójkowym (binarnym) i szesnastkowym (heksadecymalnym).
- Każdą liczbę w systemie pozycyjnym można przedstawić jako sumę wielokrotności potęg podstawy systemu.
- Najmniejszą niepodzielną jednostką informacji jest bit (reprezentuje jeden z dwóch stanów: 0 lub 1), a najmniejszą przetwarzaną porcją informacji jest bajt, składający się z 8 bitów.
- Na n bitach można przedstawić 2^n różnych informacji.
- Liczby całkowite ze znakiem zapisywane są w kodzie uzupełnień do 2 (U2). Ułatwia to wykonywanie działań arytmetycznych.
- Typy danych umożliwiające zapisywanie w języku C++ liczb całkowitych ze znakiem to np. `short int`, `int`, `long int`, `long long int`.
- Po dodaniu słowa kluczowego `unsigned` przed nazwą typu służącego do zapisywania liczb całkowitych ze znakiem otrzymujemy nazwę typu danych reprezentującego liczby całkowite bez znaku.
- Znakom przyporządkowane są kody liczbowe. Podstawowy zestaw znaków określa standard ASCII (7-bitowy kod).
- Napis to ciąg znaków.
- W języku C++ znaki możemy reprezentować za pomocą typu `char`. Zmienne tego typu zajmują jeden bajt pamięci.
- Do reprezentacji napisu, czyli ciągu znaków, można wykorzystać typ `string`.
- Do znaków w napisie odwołujemy się za pośrednictwem indeksu, czyli numeru znaku w napisie. Indeksowanie znaków w napisie zawsze zaczyna się od zera.
- Typ `string` jest typem obiektowym (klasą). Nie tylko umożliwia pamiętanie danych, lecz także udostępnia funkcje do ich przetwarzania. Funkcje dostępne dla danej klasy nazywamy metodami.
- Jedną z metod dostępnych w klasie `string` jest metoda `size`, której wynikiem jest liczba znaków napisu.
- Wartości logiczne zapamiętywane są z użyciem jednego bajta pamięci.

Zadania

- 1 Oblicz wartość dziesiętną liczby binarnej 1000100111_2.
- 2 Zapisz liczbę binarną 1000100111010000_2 w systemie szesnastkowym.
- 3 Zapisz liczbę szesnastkową $E0A8_{16}$ w systemie binarnym.
- 4 Oblicz wartość dziesiętną liczby szesnastkowej $89AB_{16}$.
- 5 Oblicz wartość dziesiętną liczby 10001001_{U2} zapisanej w kodzie uzupełnień do 2.

* **6** Zapisz liczbę dziesiętną −116, reprezentowaną przy użyciu jednego bajta pamięci, w kodzie uzupełnień do 2.

* **7** Podaj kilka własności liczb binarnych. Uwzględnij m.in. to, kiedy podana liczba jest parzysta, a kiedy nieparzysta.

* **8** Która z podanych liczb ma inną wartość niż pozostałe?

a. 1105_8
b. 1011000101_2
c. 245_{16}
d. 21011_4

* **9** Z podanych niżej zapisów wybierz ten, który może oznaczać liczbę 80 w systemie czwórkowym. Takie same znaki oznaczają taką samą cyfrę.

a. ●■★◆
a. ■■◆★
c. ●●★★
d. ◆●◆●
e. ■■■■
f. ★◆★★

** **10** Napisz program, który wczyta imię i wiek wyrażony w latach, a następnie wypisze powitanie. Dla osoby pełnoletniej wypisze napis „Dzien dobry" i jej imię, a dla niepełnoletniej napis „Czesc" i jej imię.

** **11** Napisz program, który wczyta trzy liczby całkowite i wypisze je w kolejności od najmniejszej do największej.

** **12** Napisz program, który dla wczytanej liczby całkowitej dodatniej policzy, ile jedynek występuje w jej reprezentacji binarnej.

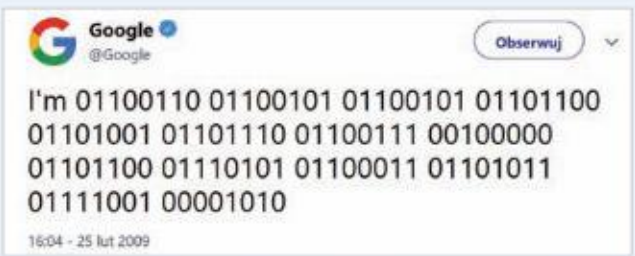
** **13** Napisz program, który wczyta słowo i sprawdzi, czy dwie takie same litery występują w nim obok siebie. Program powinien wypisać słowo „TAK" lub „NIE".

** **14** Napisz program sprawdzający warunek istnienia trójkąta. Zastosuj w nim alternatywę zdań.

*** **15** Napisz program, który wczyta słowo i sprawdzi, czy któraś litera w słowie się powtarza. Program powinien wypisać słowo „TAK" lub „NIE".

*** **16** Napisz program, który pozwoli rozstrzygnąć, czy wśród n liczb całkowitych dodatnich wprowadzonych z klawiatury była taka trójka liczb a, b i c, dla których nie da się zbudować trójkąta o długościach boków a, b i c.

*** **17** Pierwszy tweet firmy Google miał postać binarną i wyglądał następująco:



Za pomocą funkcji dostępnych w arkuszu kalkulacyjnym odczytaj treść tweeta. Możesz skorzystać np. z funkcji DWÓJK.NA.DZIES oraz ZNAK.

3. Algorytmy zamiany reprezentacji liczb między systemami liczbowymi

Wiesz już, że komputer interpretuje i zapisuje różne rodzaje informacji jako ciągi zer i jedynek. Na przykład liczbę wpisaną z klawiatury przekształca na postać binarną. Zamiana reprezentacji liczb między systemami to jedna z podstawowych operacji wykonywanych przez komputer. W tym temacie nauczysz się m.in. pisać programy zamieniające liczbę dziesiętną na jej reprezentację w innym systemie pozycyjnym i odwrotnie.

Cele lekcji

- Napiszesz programy zamieniające liczbę dziesiętną na postać dwójkową oraz liczbę dwójkową na postać dziesiętną.
- Zmodyfikujesz programy tak, aby zamieniały liczbę dziesiętną na liczbę w systemie pozycyjnym o wybranej podstawie i odwrotnie.
- Wykorzystasz reprezentację binarną liczby w algorytmie szybkiego podnoszenia do potęgi.

3.1. Algorytm zamiany reprezentacji liczby z dziesiętnej na dwójkową

System dwójkowy (binarny), s. 28–29

O zamianie liczby dziesiętnej na dwójkową mówiliśmy w temacie 2, w części dotyczącej **systemu dwójkowego** (**binarnego**). W zastosowanym tam algorytmie wyznaczaliśmy cyfry liczby binarnej w kolejności od najmniej znaczącej. Teraz zmodyfikujemy algorytm tak, aby były one zapisane we właściwej kolejności. Kolejno wyznaczane cyfry binarne będziemy łączyć w jeden napis, przy czym aktualnie wyznaczoną cyfrę dołączymy na początku napisu złożonego z otrzymanych wcześniej cyfr. Cyfry binarne będą pamiętane jako znaki. Pojedynczy znak, w odróżnieniu od napisu, czyli ciągu znaków, ujmujemy w apostrofy.

Specyfikacja problemu zamiany liczby dziesiętnej na binarną jest następująca:

Specyfikacja

Dane: d – liczba całkowita zapisana w postaci dziesiętnej, $0 < d \leqslant 2^{31} - 1$.

Wynik: b – napis reprezentujący zapis binarny liczby d.

Zapis algorytmu w pseudokodzie może wyglądać następująco:

```
b ← ""
dopóki d > 0 wykonuj
    jeśli d mod 2 = 0 to b ← '0' + b
    w przeciwnym przypadku b ← '1' + b
    d ← d div 2
wypisz b
```

Dobra rada

Odróżniaj napis złożony z pojedynczej spacji (cudzysłów, spacja, cudzysłów) od napisu pustego (cudzysłów, cudzysłów).

Wartością początkową zmiennej b jest napis pusty. Do łączenia napisów oraz znaków i napisów w jeden napis w języku C++ (a także innych językach) używamy znaku +.

Warto wiedzieć

Łączenie napisów nazywamy konkatenacją.

Tabela 3.1 ilustruje działanie algorytmu dla liczby dziesiętnej 26.

Iteracja pętli	Wartość zmiennej d	Wartość zmiennej b
–	–	""
1	26	'0'+""="0"
2	13	'1'+"0"="10"
3	6	'0'+"10"="010"
4	3	'1'+"010"="1010"
5	1	'1'+"1010"="11010"
–	0	"11010"

Tabela 3.1. Wartości zmiennych d i b przy wyznaczaniu zapisu binarnego liczby 26

Warto wiedzieć

Dodawanie (łączenie) napisów nie jest operacją przemienną.

Oto kod źródłowy programu realizującego omawiany algorytm:

Kod źródłowy programu zamieniającego liczbę dziesiętną na postać binarną

```
1.  #include <iostream>
2.  #include <string>
3.
4.  using namespace std;
5.
6.  int main()
7.  {
8.      int d;
9.      string b="";
10.     cout<<"Podaj liczbe dziesietna: "; cin>>d;
11.     while (d>0)
12.     {
13.         if (d%2==0) b='0'+b;
14.         else b='1'+b;
15.         d=d/2;
16.     }
17.     cout<<"Liczba binarna: "<<b;
18.     return 0;
19. }
```

Dobra rada

Znak ujęty w cudzysłów oznacza napis o długości 1, dlatego znaki 0 i 1 możesz zamiast w apostrofach umieścić w cudzysłowie.

W linii 9 zadeklarowana jest zmienna b typu `string`, w której będzie zapamiętana reprezentacja binarna wczytanej liczby dziesiętnej. Podczas deklaracji nadajemy tej zmiennej wartość początkową (napis pusty). W zależności od tego, czy aktualna wartość zmiennej d jest podzielna przez 2 czy nie, na początku zmiennej b dołączane będzie zero lub jeden (linie 13 i 14).

Ćwiczenie 1

Napisz program, który po podaniu liczby całkowitej dodatniej wyświetli jej reprezentację binarną. Sprawdź działanie programu dla różnych liczb.

3.2. Algorytmy zamiany liczby dwójkowej na dziesiętną

Teraz omówimy dwa algorytmy zamiany liczby dwójkowej na liczbę dziesiętną. Oto specyfikacja problemu:

Specyfikacja

Dane: b – napis reprezentujący liczbę binarną, długość napisu jest mniejsza od 32.

Wynik: d – wartość dziesiętna liczby binarnej b.

System dwójkowy (binarny), s. 28–29

Jeden ze sposobów zamiany liczby binarnej na liczbę dziesiętną przedstawiliśmy przy okazji omawiania **systemu dwójkowego** (**binarnego**). Polegał on na sumowaniu odpowiednich potęg liczby 2. O tym, które potęgi należy dodać, informują cyfry 1 w zapisie binarnym. Kłopot może stanowić wyznaczanie wartości kolejnych potęg. Przeglądając cyfry liczby binarnej od prawej do lewej, należy wymnażać wartość zmiennej pomocniczej (nazwijmy ją potega) przez 2. Jej wartość początkowa będzie wynosiła 1 (odpowiada wartości 2^0). W kolejnych iteracjach pętli wartość zmiennej mnożymy przez 2, a więc obliczamy kolejne potęgi liczby 2. Oto zapis tego algorytmu w pseudokodzie:

Dobra rada

Pamiętaj, że znaki napisu są indeksowane od 0, więc ostatni znak ma indeks o jeden mniejszy od długości napisu.

```
potega ← 1
d ← 0
dla i ← długość liczby binarnej - 1, ..., 0 wykonuj
    jeśli b[i] = '1' to d ← d + potega
    potega ← potega * 2
wypisz d
```

Implementacja tego algorytmu w języku C++ jest następująca. Zwróć uwagę na linię 11 – w pętli `for` wartość zmiennej i jest pomniejszana o jeden (instrukcja i--).

```cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
    int i, d=0, potega=1;
    string b;
    cout<<"Podaj liczbe binarna: "; cin>>b;
    for (i=b.size()-1;i>=0;i--)
    {
        if (b[i]=='1') d=d+potega;
        potega=potega*2;
    }
    cout<<"Liczba dziesietna: "<<d;
    return 0;
}
```

Kod źródłowy programu zamieniającego liczbę binarną na dziesiętną – algorytm 1

Warto wiedzieć

Instrukcję `i--`, oznaczającą pomniejszanie wartości zmiennej `i` o 1, nazywamy dekrementacją. Powiększanie wartości zmiennej o 1 (instrukcja `i++`) to inkrementacja.

Ćwiczenie 2

Napisz program, który będzie zamieniać liczbę binarną na jej postać dziesiętną z wykorzystaniem dodawania odpowiednich potęg liczby 2. Uruchom program i przetestuj go dla różnych liczb.

Aby zamienić liczbę binarną na zapis dziesiętny, możemy też przeglądać cyfry liczby binarnej od lewej do prawej, czyli od cyfry najbardziej znaczącej. Nie trzeba będzie wówczas w oddzielnej zmiennej wyznaczać potęg liczby 2. Rozpatrzmy to na przykładzie liczby 1101_2. Wyciągnijmy wielokrotnie 2 przed nawias.

$$\begin{aligned}
1101_2 &= 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = \\
&= 2 \cdot (1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0) + 1 = \\
&= 2 \cdot (2 \cdot (1 \cdot 2^1 + 1 \cdot 2^0) + 0) + 1 = \\
&= 2 \cdot (2 \cdot (2 \cdot (1 \cdot 2^0) + 1) + 0) + 1 = \\
&= 2 \cdot (2 \cdot (2 \cdot (1) + 1) + 0) + 1
\end{aligned}$$

W każdej iteracji pętli aktualną wartość szukanej liczby należy mnożyć przez 2. Dodatkowo jeśli w zapisie binarnym występuje cyfra 1, to dodajemy ją do wyniku. Pierwszą analizowaną cyfrą binarną jest 1 w najbardziej zagnieżdżonych nawiasach – odpowiada ona najbardziej znaczącej cyfrze. Oto zapis algorytmu w pseudokodzie:

```
d ← 0
dla i ← 0, 1, …, długość liczby binarnej – 1 wykonuj
    d ← d * 2
    jeśli b[i] = '1' to d ← d + 1
wypisz d
```

Tabela 3.2 ilustruje działanie algorytmu dla liczby 1101_2.

Iteracja pętli	Cyfry liczby binarnej w kolejności od najbardziej znaczącej	Obliczana wartość dziesiętna
–	–	0
1	1	$0 \cdot 2 + 1 = 1$
2	1	$1 \cdot 2 + 1 = 3$
3	0	$3 \cdot 2 = 6$
4	1	$6 \cdot 2 + 1 = 13$

Tabela 3.2. Wyznaczanie wartości dziesiętnej liczby 1101_2

Ćwiczenie 3

Zilustruj za pomocą tabeli działanie podanego algorytmu dla liczby 10100101_2.

Oto realizacja omawianego algorytmu w języku C++:

Kod źródłowy programu zamieniającego liczbę binarną na dziesiętną – algorytm 2

```
1.  #include <iostream>
2.  #include <string>
3.
4.  using namespace std;
5.
6.  int main()
7.  {
8.      int i, d=0;
9.      string b;
10.     cout<<"Podaj liczbe binarna: "; cin>>b;
11.     for (i=0;i<b.size();i++)
12.     {
13.         d=d*2;
14.         if (b[i]=='1') d++;
15.     }
16.     cout<<"Liczba dziesietna: "<<d;
17.     return 0;
18. }
```

Ćwiczenie 4

Napisz program zamieniający liczbę binarną na jej postać dziesiętną, realizujący drugi z opisanych algorytmów. Uruchom program i sprawdź jego działanie dla różnych liczb.

3.3. Uogólnienie algorytmów zamiany reprezentacji liczb między systemami pozycyjnymi

Algorytmy umożliwiające zamianę liczby dziesiętnej na jej reprezentację w wybranym systemie pozycyjnym i odwrotnie – z systemu o wybranej podstawie na system dziesiętny – są podobne jak w przypadku zamiany reprezentacji liczby między systemami binarnym i dziesiętnym. Liczbę 2 należy zastąpić wszędzie podstawą systemu – oznaczymy ją przez p. Należy także uogólnić dołączenie aktualnie otrzymanej cyfry do wyniku, ponieważ do zapisu liczby w danym systemie możemy wykorzystać p różnych cyfr.

Zamiana liczby dziesiętnej na liczbę w systemie pozycyjnym o wybranej podstawie

W zapisie liczb mamy do dyspozycji 10 cyfr dziesiętnych oraz 26 wielkich liter alfabetu łacińskiego – w sumie 36 znaków. Dlatego ograniczymy podstawę systemu do 36. Oto specyfikacja problemu:

Specyfikacja

Dane: d – liczba całkowita zapisana w postaci dziesiętnej,
$0 < d \leqslant 2^{31} - 1$,
p – podstawa systemu, $2 \leqslant p \leqslant 36$.

Wynik: s – napis reprezentujący zapis liczby d w systemie o podstawie p.

Problem może stanowić implementacja dołączania cyfry do wyniku. W przypadku systemu binarnego wynikiem d **mod** p jest jedna z dwóch cyfr. Natomiast dla systemów o podstawie większej niż 2 będzie to jedna z wielu cyfr. Liczbę w systemie o podstawie p reprezentujemy jako napis, a do napisu możemy dodawać tylko znaki lub inne napisy. W algorytmie zamiany na system binarny dodawaliśmy znak 0 lub 1. Teraz należy zamienić otrzymaną wartość wyrażenia d **mod** p na znak reprezentujący odpowiednią cyfrę w **tablicy kodów ASCII**. Zapis algorytmu w pseudokodzie może być następujący:

Tablica kodów ASCII, s. 37

```
s ← ""
dopóki d > 0 wykonuj
    s ← cyfra(d mod p) + s
    d ← d div p
wypisz s
```

Ograniczmy na razie podstawę systemu do wartości $2 \leqslant p \leqslant 10$. Dzięki temu uwzględnimy tylko kody ASCII dla cyfr od 0 do 9. Zaczynają się one od 48, tzn. znak 0 ma kod 48, znak 1 kod 49 itd. Wystarczy więc do wartości wyrażenia d **mod** p dodać liczbę 48, aby otrzymać kod ASCII odpowiedniej cyfry. Oto kod źródłowy programu.

Kod źródłowy programu zamieniającego liczbę dziesiętną na liczbę w systemie pozycyjnym o podstawie z zakresu od 2 do 10

```
1.  #include <iostream>
2.  #include <string>
3.
4.  using namespace std;
5.
6.  int main()
7.  {
8.      int d, p;
9.      string s="";
10.     cout<<"Podaj liczbe dziesietna: "; cin>>d;
11.     cout<<"Podaj podstawe systemu: "; cin>>p;
12.     while (d>0)
13.     {
14.         s=char(48+d%p)+s;
15.         d=d/p;
16.     }
17.     cout<<"Liczba w systemie o podstawie "<<p<<": "<<s;
18.     return 0;
19. }
```

Dobra rada

Ponieważ w języku C++ znaki są reprezentowane przez kody ASCII, a typ **char** reprezentuje zarówno znaki, jak i liczby jednobajtowe, w linii 14 kodu zamiast 48+d%p możesz napisać '0'+d%p. Dzięki temu nie musisz pamiętać kodów cyfr z tablicy ASCII.

W linii 14 do kodu ASCII cyfry 0 jest dodana wartość reszty z dzielenia aktualnej wartości d przez podstawę systemu. Potem następuje **konwersja (rzutowanie typu)** otrzymanego w ten sposób kodu ASCII na typ znakowy **char** i połączenie otrzymanego znaku z aktualną wartością zmiennej s. Tabela 3.3 ilustruje działanie programu dla liczby dziesiętnej 212 i podstawy systemu 8.

Konwersja (rzutowanie typu) to zmiana typu danych.

Iteracja pętli	Wartość zmiennej d	Wartość wyrażenia 48+d%p (kod ASCII cyfry)	Wartość zmiennej s
–	–	–	""
1	212	48 + 4 = 52	'4' + "" = "4"
2	26	48 + 2 = 50	'2' + "4" = "24"
3	3	48 + 3 = 51	'3' + "24" = "324"
–	0	–	"324"

Tabela 3.3. Wartości zmiennych d i s przy wyznaczaniu reprezentacji liczby 212 w systemie o podstawie 8

Jeśli podstawa systemu jest większa od 10, oprócz cyfr 0–9 musimy uwzględnić cyfry oznaczane literami. Gdyby kody wielkich liter występowały w tabeli kodowej bezpośrednio po cyfrach, nie trzeba by modyfikować programu. Tak jednak nie jest, więc należy instrukcję z linii 14 programu zastąpić instrukcją warunkową (rozpoznajemy, czy reszta z dzielenia, czyli cyfra w danym systemie pozycyjnym, jest mniejsza od 10 czy nie). Do zapamiętania otrzymanej reszty można użyć dodatkowej zmiennej – nazwiemy ją cyfra.

Zmodyfikowany kod źródłowy programu jest następujący:

Kod źródłowy programu zamieniającego liczbę dziesiętną na liczbę w systemie pozycyjnym o podstawie z zakresu od 2 do 36

```
1.  #include <iostream>
2.  #include <string>
3.
4.  using namespace std;
5.
6.  int main()
7.  {
8.      int d, p, cyfra;
9.      string s="";
10.     cout<<"Podaj podstawe systemu: "; cin>>p;
11.     cout<<"Podaj liczbe dziesietna: "; cin>>d;
12.     while (d>0)
13.     {
14.         cyfra=d%p;
15.         if (cyfra<10) s=char('0'+cyfra)+s;
16.         else s=char('A'-10+cyfra)+s;
17.         d=d/p;
18.     }
19.     cout<<"Liczba w systemie o podstawie "<<p<<": "<<s;
20.     return 0;
21. }
```

W linii 16 od kodu ASCII litery „A" zostało odjęte 10, ponieważ w systemach pozycyjnych cyfry oznaczone literami zaczynają się od 10. Znaki w języku C++ są reprezentowane przez swoje kody ASCII, więc do kodu znaku możemy się odwołać, podając wprost ten znak. Dlatego zapisaliśmy `'A'` zamiast 65 (kodu litery „A").

Ćwiczenie 5

Napisz program, dzięki któremu zamienisz liczbę dziesiętną na jej reprezentację w systemie pozycyjnym o podstawie z zakresu od 2 do 36. Sprawdź działanie programu dla różnych danych wejściowych.

Zamiana liczby w systemie pozycyjnym o wybranej podstawie na liczbę dziesiętną

Specyfikacja problemu zamiany liczby zapisanej w systemie o podstawie p na jej reprezentację w systemie dziesiętnym jest następująca:

Specyfikacja

Dane: s – napis reprezentujący liczbę w systemie o podstawie p, długość napisu jest mniejsza od 6,
p – podstawa systemu, $2 \leqslant p \leqslant 36$.

Wynik: d – wartość dziesiętna liczby s.

Warto wiedzieć

Podany pseudokod jest analogiczny do przedstawionego w tym temacie drugiego algorytmu zamiany liczby binarnej na dziesiętną (bez wyliczania potęg podstawy systemu). Mnożenie przez 2 jest zastąpione mnożeniem przez podstawę systemu, a zamiast 1 dodawana jest odpowiednia cyfra z zapisu liczby.

Ograniczyliśmy długość napisu `s`, żeby wartość dziesiętna liczby mieściła się w zakresie typu **`int`** dla dużych wartości podstawy systemu. Oto zapis algorytmu w pseudokodzie:

```
d ← 0
dla i ← 0, 1, …, długość zapisu liczby - 1 wykonuj
    d ← d * p + wartość(s[i])
wypisz d
```

Zapis `s[i]` oznacza znak z napisu `s` znajdujący się na pozycji `i`. Do wartości zmiennej `d` mnożonej przez podstawę systemu `p` musimy dodać liczbę, więc dokonujemy konwersji znaku na liczbę. Operację tę oznaczyliśmy przez `wartość(s[i])`.

Poniżej znajduje się kod źródłowy programu przy ograniczeniu podstawy systemu do liczb z zakresu od 2 do 10. Zauważ, że w linii 15 nie jest potrzebna konwersja typu. Operacje są wykonywane na kodach ASCII znaków, a więc na liczbach.

Kod źródłowy programu zamieniającego liczbę w systemie pozycyjnym o podstawie z zakresu od 2 do 10 na liczbę dziesiętną

```
1.  #include <iostream>
2.  #include <string>
3.
4.  using namespace std;
5.
6.  int main()
7.  {
8.      int i, p, d=0;
9.      string s;
10.     cout<<"Podaj podstawe systemu: ";
11.     cin>>p;
12.     cout<<"Podaj liczbe w systemie o podstawie "<<p<<": ";
13.     cin>>s;
14.     for (i=0;i<s.size();i++)
15.         d=d*p+s[i]-'0';
16.     cout<<"Liczba dziesietna: "<<d;
17.     return 0;
18. }
```

Warto wiedzieć

Ponieważ w informatyce często wykorzystuje się zapisy ósemkowy i szesnastkowy, w języku C++ na etapie wczytywania lub wypisywania liczby można określić, w jakim systemie ma ona być zapisana lub wyświetlona. Instrukcje z poniższego przykładu realizują wczytanie liczby szesnastkowej i wypisanie jej w systemie ósemkowym.

```
int x;
cin>>hex>>x;
cout<<oct<<x;
```

Ćwiczenie 6

Zmodyfikuj powyższy program tak, aby działał prawidłowo dla podstawy systemu z zakresu od 2 do 36. Przetestuj jego działanie.

Zapamiętaj

Znaki w języku C++ są reprezentowane przez kody ASCII i można na nich wykonywać działania arytmetyczne. Nie trzeba w tym celu przeprowadzać konwersji (rzutowania typu). Żeby uzyskać znak na podstawie kodu ASCII, należy dokonać konwersji na typ **`char`**.

3.4. Algorytm szybkiego podnoszenia do potęgi

Rozwinięcie binarne liczby dziesiętnej wykorzystamy do obliczenia wartości potęgi o wykładniku naturalnym. Oto specyfikacja problemu:

Specyfikacja

Dane: n – liczba całkowita, $n \geqslant 0$,
x – liczba całkowita.

Wynik: $y = x^n$

W **algorytmie naiwnym**, czyli wykorzystującym najbardziej intuicyjną metodę, wykonalibyśmy $n - 1$ operacji mnożenia. Na przykład żeby obliczyć x^{13}, musielibyśmy wykonać mnożenie 12 razy:

Algorytm naiwny

$$x^{13} = \underbrace{x \cdot x \cdot \ldots \cdot x}_{13 \text{ czynników}}$$

Liczbę operacji możemy zmniejszyć, jeśli wykorzystamy fakt, że każdą liczbę całkowitą dodatnią można przedstawić jako sumę różnych potęg liczby 2. Własność ta wynika wprost z reprezentacji binarnej liczby. W naszym przykładzie:

Warto wiedzieć

Algorytmy naiwne są często mało efektywne.

$13 = 8 + 4 + 1$
ponieważ
$13 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1101_2$
Zatem
$x^{13} = x^{1101_2} = x^{8+4+1} = x^8 \cdot x^4 \cdot x^1$

Zauważ, że aby obliczyć wartość x^{13}, wystarczy wykonać pięć operacji mnożenia zamiast dwunastu – trzy mnożenia, które pozwolą wyznaczyć wartości potrzebnych potęg x: x^2 (czyli $x \cdot x$), x^4 ($x^2 \cdot x^2$), x^8 ($x^4 \cdot x^4$), oraz dwa mnożenia wykorzystujące te wartości: $x^8 \cdot x^4 \cdot x^1$.

A to ciekawe

Googol czy Google?

Czy wiesz, skąd pochodzi nazwa wyszukiwarki, z której zapewne najczęściej korzystasz? Okazuje się, że od nazwy liczby googol, czyli 10^{100} (jedynka i sto zer). W 1997 r. Larry Page i Sergey Brin, studenci Uniwersytetu Stanforda w Kalifornii, poszukiwali nazwy dla wyszukiwarki opierającej się na opracowanym przez nich algorytmie wyszukiwania znanym jako PageRank. Wybór padł na nazwę „googol", jednak podczas rejestracji domeny popełniono błąd – zamiast googol.com wpisano znane obecnie wszystkim google.com.

Uogólniając, podczas poszukiwania wartości x^n musimy obliczać wartości potęg liczby x, których wykładniki są kolejnymi potęgami liczby 2. W wyniku uwzględniamy te wartości, dla których w rozwinięciu binarnym wykładnika występuje cyfra 1. Zapiszmy ten algorytm w pseudokodzie.

Warto wiedzieć

Przedstawiony algorytm podnoszenia do potęgi można zastosować także wtedy, gdy podstawa jest liczbą rzeczywistą.

```
y ← 1
tmp ← x
dopóki n > 0 wykonuj
    jeśli n mod 2 = 1 to y ← y * tmp
    n ← n div 2
    jeśli n > 0 to tmp ← tmp * tmp
wypisz y
```

W zmiennej tmp wyliczane są kolejne wartości x^m, gdzie m jest potęgą liczby 2. Jeśli w rozwinięciu binarnym wykładnika występuje cyfra 1, to aktualna wartość zmiennej tmp jest wykorzystywana, czyli wynik (zmienna y) jest przez nią mnożony. Kod źródłowy wygląda następująco:

Kod źródłowy programu obliczającego potęgę liczby

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  int main()
6.  {
7.      int x, n;
8.      long long y, tmp;
9.
10.     cout<<"Podaj podstawe potegi: "; cin>>x;
11.     cout<<"Podaj wykladnik potegi: "; cin>>n;
12.     tmp=x; y=1;
13.     while (n>0)
14.     {
15.         if (n%2==1) y=y*tmp;
16.         n=n/2;
17.         if (n>0) tmp=tmp*tmp;
18.     }
19.     cout<<"Wartosc potegi: "<<y;
20.     return 0;
21. }
```

Zwróć uwagę na deklarację zmiennych w liniach 7 i 8. Wartość potęgi szybko rośnie, dlatego dla zmiennych tmp i y spośród **typów danych umożliwiających zapisywanie liczb całkowitych** został wybrany typ **long long**.

Wybrane typy danych w języku C++ umożliwiające zapisywanie liczb całkowitych, s. 35–36

Ćwiczenie 7

Napisz i przetestuj program szybkiego podnoszenia do potęgi. Dobieraj dane tak, aby nie przekroczyć zakresu wyniku.

Podsumowanie

- Aby zamienić liczbę dziesiętną na binarną, wykorzystujemy dzielenie przez 2 – kolejno otrzymywane reszty z dzielenia to poszukiwane cyfry binarne liczby, począwszy od najmniej znaczącej.
- W ogólnym przypadku zamiany liczby dziesiętnej na jej reprezentację w innym systemie liczbowym wykorzystujemy dzielenie przez podstawę systemu.
- Żeby obliczyć wartość dziesiętną liczby zapisanej w systemie pozycyjnym o podstawie p, należy przeglądać cyfry liczby od najbardziej znaczącej i mnożyć aktualną wartość wyniku przez podstawę systemu, a następnie dodać kolejną cyfrę liczby.
- Każdą liczbę całkowitą dodatnią można przedstawić jako sumę różnych potęg liczby 2.
- Reprezentację binarną wykładnika potęgi można wykorzystać do szybkiego podnoszenia liczby do potęgi.

Zadania

* **1** Napisz program, który wczyta liczbę dziesiętną i wypisze liczbę cyfr potrzebną do zapisania tej liczby w systemie binarnym. Na przykład dla liczby 99 wypisze 7, ponieważ $99 = 1100011_2$.

* **2** Napisz program, który wczyta liczbę zapisaną w systemie czwórkowym i wypisze jej zapis w systemie binarnym.

* **3** Napisz program, który wczyta liczbę zapisaną w systemie ósemkowym i wypisze jej zapis w systemie binarnym.

* **4** Napisz program, który wczyta liczbę zapisaną w systemie szesnastkowym i wypisze jej zapis w systemie binarnym.

* **5** Napisz program, który wczyta liczbę zapisaną w systemie dziewiątkowym i wypisze jej zapis w systemie trójkowym.

** **6** Napisz program, który wczyta liczbę zapisaną w systemie ósemkowym i wypisze jej zapis w systemie szesnastkowym.

** **7** Napisz program, który po podaniu liczby cyfr potrzebnych do zapisania liczby binarnej wypisze liczbę cyfr potrzebną do zapisania tej liczby odpowiednio w systemach czwórkowym, ósemkowym i szesnastkowym. Na przykład jeśli użytkownik poda liczbę 6, program dla systemu czwórkowego wypisze liczbę 3, a dla ósemkowego i szesnastkowego – liczbę 2.

** **8** Napisz program, który po podaniu liczby binarnej wypisze jej zapis odpowiednio w systemach czwórkowym, ósemkowym i szesnastkowym.

** **9** Zaimplementuj w arkuszu kalkulacyjnym algorytm zamiany liczby dwójkowej na postać dziesiętną, uwzględniający przeglądanie cyfr od lewej do prawej.

*** **10** Do zapisu liczb w zrównoważonym systemie trójkowym używa się trzech znaków: 0, +, –. Znak + odpowiada wartości 1, a znak – odpowiada wartości –1. Na przykład liczba –25 zapisana w tym systemie miałaby postać: –0+–. Wynika to z faktu, że:
$-25 = -1 \cdot 3^3 + 0 \cdot 3^2 + 1 \cdot 3^1 - 1 \cdot 3^0 = -27 + 0 + 3 - 1$
Poniższa tabela przedstawia przykładowe wartości dziesiętne liczb zapisanych w tym systemie.

System trójkowy zrównoważony	System dziesiętny
–++	–5
––	–4
–0	–3
–+	–2
–	–1
0	0
+	1
+–	2
+0	3
++	4
+––	5

Napisz program, który po wczytaniu liczby zapisanej w zrównoważonym systemie trójkowym wypisze jej wartość w systemie dziesiętnym.

*** **11** W silniowym systemie pozycyjnym mnożniki poszczególnych pozycji są definiowane nie przez potęgę podstawy systemu, ale przez silnię kolejnych liczb naturalnych. Na przykład:
$2011_! = 2 \cdot 4! + 0 \cdot 3! + 1 \cdot 2! + 1 \cdot 1! = 2 \cdot 24 + 0 + 2 + 1 = 51$
Zapis w silniowym systemie pozycyjnym jest jednoznaczny, tzn. każdą liczbę naturalną można zapisać tylko w jeden sposób i każdy zapis oddaje dokładnie jedną wartość dziesiętną.
Napisz program, który wczyta liczbę dziesiętną x, gdzie $0 \leqslant x \leqslant 3\,628\,799$, i wypisze jej zapis w silniowym systemie pozycyjnym.
Uwaga: $3\,628\,799 = 987654321_!$

4. Czy to jest palindrom?

Gry i zabawy słowne są znane i lubiane od wieków. Być może zdarzyło ci się grać w grę polegającą na szukaniu wyrazu zaczynającego się na ostatnią literę wyrazu podanego przez poprzednią osobę. Znasz też na pewno wiele tzw. łamańców językowych, czyli wyrażeń tworzonych celowo tak, żeby trudno je było wymówić, jak „stół z powyłamywanymi nogami". Inną zabawą, która od starożytności zajmuje ludzi posługujących się różnymi językami, jest szukanie słów, a nawet całych zdań, które brzmią tak samo czytane wspak. Właśnie takimi wyrażeniami zajmiemy się w tym temacie.

Cele lekcji

- Dowiesz się, czym jest palindrom.
- Zbadasz, czy wyraz oraz zdanie są palindromami.
- Wykonasz operacje na zmiennych napisowych.
- Zdefiniujesz własne funkcje w języku C++.
- Poznasz różnicę między parametrem formalnym i aktualnym.
- Wyszukasz w zdaniu wyrazy będące palindromami.

Palindrom to słowo, wyrażenie bądź zdanie, które czytane od lewej do prawej i od prawej do lewej brzmi tak samo. Nazwa wywodzi się od greckiego słowa *palindromos*, które znaczy „biegnący na powrót". Palindromami są np. słowo „sms" czy wyrażenie „oko w oko".

Palindrom

4.1. Sprawdzamy, czy wyraz jest palindromem

Pierwszym zadaniem, które sobie postawimy, będzie sprawdzenie, czy pojedynczy wyraz jest palindromem. Oto specyfikacja problemu:

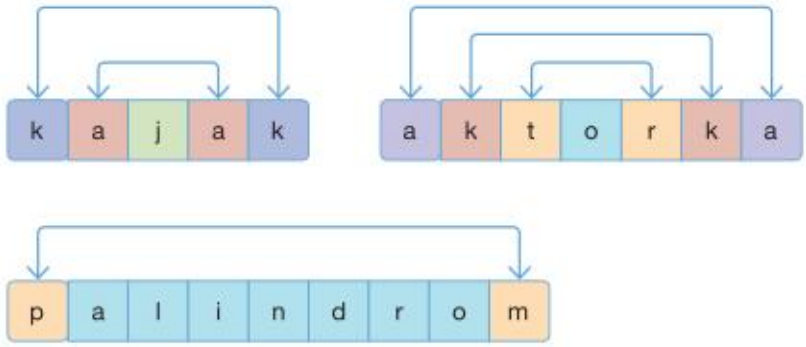
Specyfikacja

Dane: `wyraz` – słowo złożone z małych liter alfabetu łacińskiego.

Wynik: słowo „TAK", gdy wyraz jest palindromem, słowo „NIE" – w przeciwnym przypadku.

Podczas sprawdzania będziemy porównywać kolejne pary liter: pierwszą z ostatnią, drugą z przedostatnią itd. Znalezienie pierwszej pary różnych liter oznacza, że wyraz nie jest palindromem, więc można wtedy zakończyć analizę. Jeśli nie znajdziemy pary różnych liter, to wyraz jest palindromem. Zauważ, że jeśli palindrom składa się z nieparzystej liczby liter, nie musimy porównywać środkowej litery samej ze sobą. Ogólnie, gdy wyraz jest palindromem, wykonamy `n div 2` porównań par liter, gdzie `n` oznacza liczbę liter w wyrazie.

Rysunek 4.1 przedstawia działanie algorytmu dla trzech przykładowych wyrazów. Pierwszy wyraz jest palindromem – tu trzeba wykonać dwa porównania. Kolejne dwa wyrazy nie są palindromami. Aby się o tym przekonać, w przypadku drugiego wyrazu musimy wykonać trzy porównania, a w przypadku trzeciego – jedno porównanie.



Rys. 4.1. Sprawdzenie, czy podane wyrazy są palindromami

Dobra rada

Podany algorytm zadziała dla dowolnego ciągu znaków bez spacji. Za jego pomocą możesz więc np. sprawdzić, czy liczba reprezentowana jako napis jest palindromem.

W pseudokodzie zapis algorytmu sprawdzającego, czy wyraz jest palindromem, może wyglądać następująco:

```
palindrom ← prawda
i ← 0
j ← długość wyrazu - 1
dopóki palindrom oraz (i < j) wykonuj
    jeśli wyraz[i] = wyraz[j] to
        i ← i + 1
        j ← j - 1
    w przeciwnym przypadku palindrom ← fałsz
jeśli palindrom to wypisz "TAK"
w przeciwnym przypadku wypisz "NIE"
```

Na początku zakładamy, że badany wyraz jest palindromem, dlatego zmiennej `palindrom` typu logicznego przypisujemy wartość **prawda**. Dodatkowo przyjmujemy, że słowo puste bądź słowo złożone tylko z jednej litery jest palindromem. Zmienna `i` wskazuje kolejne litery z początku wyrazu (wartość początkowa tej zmiennej jest równa 0, ponieważ znaki napisu indeksowane są od zera), a zmienna `j` – kolejne litery od końca wyrazu (wartość początkowa jest równa indeksowi ostatniej litery wyrazu, czyli długości wyrazu pomniejszonej o 1). Jeśli w aktualnie porównywanej parze są takie same litery, wartość zmiennej `i` jest powiększana o 1, a zmiennej `j` – pomniejszana o 1. W przypadku, gdy litery w badanej parze są różne, zmienna `palindrom` przyjmuje wartość **fałsz**, co powoduje zakończenie wykonywania pętli. Jeśli badany wyraz jest palindromem, instrukcje wykonują się, dopóki `i < j`.

Zapiszemy teraz kod źródłowy programu realizującego podany algorytm. Program zadziała błędnie, gdy wyraz będzie zawierał zarówno małe, jak i wielkie litery, np. litery „A" i „a" odczyta jako różne znaki.

Kod źródłowy programu sprawdzającego, czy słowo jest palindromem

```
1.  #include <iostream>
2.  #include <string>
3.
4.  using namespace std;
5.
6.  int main()
7.  {
8.      string wyraz;
9.      int i=0, j;
10.     bool palindrom=true;
11.     cout<<"Podaj wyraz: "; cin>>wyraz;
12.     j=wyraz.size()-1;
13.     while (palindrom && i<j)
14.         if (wyraz[i]==wyraz[j])
15.         {
16.             i++;
17.             j--;
18.         }
19.         else palindrom=false;
20.     if (palindrom) cout<<"TAK";
21.     else cout<<"NIE";
22.     return 0;
23. }
```

Aby program wypisywał słowo „TAK" niezależnie od tego, czy do zapisu palindromu użyto małych czy wielkich liter, możemy np. zastosować **funkcję** `toupper`. Jej wartością jest kod ASCII wielkiej litery, jeśli **parametr** był małą literą. W przeciwnym przypadku wartością funkcji jest parametr bez zmiany. Na przykład dla parametrów `'A'` i `'a'` wynikiem funkcji będzie ta sama wartość – liczba 65 (kod ASCII litery „A"). W kodzie źródłowym programu wystarczy zmienić warunek w instrukcji warunkowej (linia 14) na:

Funkcja `toupper`

Parametr funkcji, s. 64

```
toupper(wyraz[i])==toupper(wyraz[j])
```

Zamiast funkcji `toupper` moglibyśmy skorzystać z **funkcji** `tolower`. Jej wartością jest kod ASCII małej litery, jeśli parametr był wielką literą. W przeciwnym przypadku wartością funkcji jest parametr bez zmiany, podobnie jak w przypadku funkcji `toupper`.

Funkcja `tolower`

Funkcje `toupper` i `tolower` są zdefiniowane w bibliotece `cctype`.

Ćwiczenie 1

a. Napisz program sprawdzający, czy wyraz złożony z małych liter alfabetu łacińskiego jest palindromem.

b. Zmodyfikuj specyfikację oraz program sprawdzający, czy wyraz jest palindromem, w taki sposób, aby wypisywał słowo „TAK" niezależnie od tego, czy do zapisu wyrazu użyto małych czy wielkich liter.

4.2. Znajdujemy słowa palindromy w zdaniu

Rozwiążemy teraz kolejny problem – sprawdzimy, które wyrazy w zdaniu są palindromami. Program, który napiszemy, wypisze palindromy – niezależnie od tego, czy do zapisu użyto małych czy wielkich liter. Zakładamy, że jedynymi separatorami słów w zdaniu są spacje.

Specyfikacja

Dane: zdanie – napis złożony z liter alfabetu łacińskiego oraz spacji.

Wynik: słowa palindromy z napisu zdanie.

Zauważ, że zwykle w zdaniu spacja występuje po każdym słowie z wyjątkiem ostatniego. Jeśli dodamy spację na końcu napisu, wówczas po każdym wyrazie będzie spacja. Dodany znak będzie wtedy pełnić funkcję **wartownika**, czyli elementu ułatwiającego odnalezienie końca danych – w naszym przypadku końca zdania.

Wartownik

Algorytm znajdujący palindromy w zdaniu będzie działał następująco: wyszukujemy pierwsze wystąpienie znaku spacji w zdaniu, a następnie badamy, czy wyraz złożony z liter znajdujących się przed tą spacją jest palindromem. Po sprawdzeniu usuwamy ten wyraz ze zdania. Wymienione czynności powtarzamy, dopóki napis nie jest pusty.

Do zapamiętywania słów użyjemy zmiennej wyraz. Jeśli w danej iteracji pętli wartość zmiennej będzie palindromem, program wypisze ją na ekranie. Oto zapis algorytmu w pseudokodzie:

Warto wiedzieć

Jeśli pierwszy znak napisu jest spacją, to wykona się tylko instrukcja po instrukcji warunkowej. Spowoduje to usunięcie spacji z początku zdania.

```
zdanie ← zdanie + ' '
dopóki długość zdania > 0 wykonuj
    i ← miejsce wystąpienia pierwszej spacji
    jeśli i > 0 to // jeśli spacja nie jest pierwszym znakiem
        wyraz ← pierwsze i znaków zdania
        jeśli Palindrom(wyraz) to wypisz wyraz
    zdanie ← zdanie bez i + 1 początkowych znaków
    // usuń ze zdania wyraz razem ze spacją
```

W podanym algorytmie założyliśmy, że mamy do dyspozycji operację, która sprawdza, czy wyraz jest palindromem – nazwaliśmy ją Palindrom. Dzięki rozłożeniu problemu algorytmicznego na podproblemy algorytm jest bardziej czytelny.

Funkcje w języku C++

W praktyce rozkładanie problemów na mniejsze oznacza wyodrębnianie fragmentów programów, czyli podprogramów, które w C++ nazywamy **funkcjami**. Często korzystamy z różnych funkcji (dostępnych w bibliotekach), choć nie znamy ich kodu ani algorytmów, które wykonują. Musimy wiedzieć jedynie, jakie dane, nazywane **parametrami**, należy dostarczyć oraz co jest wynikiem danej funkcji.

Funkcja

Parametr funkcji

Budowa funkcji w języku C++ jest analogiczna do budowy znanej ci funkcji `main`. **Definicja funkcji** składa się z nagłówka funkcji i bloku instrukcji. Nagłówek zawiera typ funkcji, jej nazwę oraz listę parametrów, tzn. wskazanie typu i nazwy każdego z nich.

Definicja funkcji

```
typ nazwa_funkcji(typ parametr1, typ parametr2, ...)
{
    // blok instrukcji funkcji
}
```

Parametry określone w definicji funkcji nazywamy **parametrami formalnymi**. Natomiast parametry, dla których funkcja jest wywołana, to **parametry aktualne**. Podczas wywołania funkcji w miejsce parametrów formalnych podstawiane są parametry aktualne. Typy parametrów aktualnych muszą być zgodne z typami parametrów formalnych.

Parametr formalny

Parametr aktualny

Własne funkcje w języku C++ definiujemy przed funkcją `main`. Jeśli funkcja ma zwrócić wartość, w bloku instrukcji musi zostać użyta instrukcja **`return`** – po niej podajemy zwracaną wartość zgodną z typem funkcji określonym w nagłówku. Instrukcja **`return`** jednocześnie kończy działanie funkcji. Istnieją także funkcje niezwracające wartości – są to **funkcje typu `void`**. Poznasz je w temacie 7.

Funkcja typu `void`, s. 113

Dzięki konsekwentnemu stosowaniu funkcji tworzony kod jest bardziej przejrzysty i łatwiejszy do modyfikowania. Unikamy także powtórzeń, ponieważ tę samą funkcję można wywołać wielokrotnie dla różnych wartości parametrów.

Zapamiętaj

Definicja funkcji składa się z dwóch części: nagłówka oraz bloku instrukcji, które funkcja ma wykonać. W nagłówku podaje się typ funkcji, jej nazwę oraz listę parametrów formalnych, zawierającą nazwę i typ każdego z parametrów. Jeśli funkcja ma zwrócić wartość, należy użyć instrukcji `return`, a po niej podać wartość, która będzie wynikiem funkcji.

Definicja funkcji `Palindrom`

Teraz zdefiniujemy funkcję `Palindrom`, którą wykorzystamy do rozwiązania problemu głównego – znajdowania palindromów w zdaniu. Parametrem funkcji jest napis bez spacji (wyraz), a wynikiem – wartość logiczna informująca, czy wyraz jest palindromem czy nie. Funkcja `Palindrom` musi zwrócić wartość logiczną, ponieważ w algorytmie rozwiązującym problem główny używamy tej funkcji w warunku instrukcji warunkowej (szósta linia pseudokodu na poprzedniej stronie). W kodzie źródłowym programu definicja funkcji znajduje się w liniach 7–18.

I część kodu źródłowego programu wypisującego słowa palindromy ze zdania – definicja funkcji `Palindrom`

```
1.  #include <iostream>
2.  #include <string>
3.  #include <cctype>
4.
5.  using namespace std;
6.
7.  bool Palindrom(string wyraz)
8.  {
9.      int i=0, j=wyraz.size()-1;
10.     bool p=true;
11.     while (p && i<j)
12.         if (toupper(wyraz[i])==toupper(wyraz[j]))
13.         {
14.             i++; j--;
15.         }
16.         else p=false;
17.     return p;
18. }
```

W nagłówku funkcji (linia 7) są określone typ wyniku funkcji: **bool** oraz nazwa funkcji: `Palindrom`, a w nawiasie – parametr o nazwie `wyraz` typu **string**. Zwróć uwagę na instrukcję w linii 17. Ponieważ wynikiem funkcji jest wartość typu logicznego, instrukcja **return** zwraca wartość zmiennej `p`, w której przechowywana jest wartość logiczna informująca, czy wyraz jest palindromem.

Definicja funkcji `main` wykorzystującej funkcję `Palindrom`

Definicja funkcji `main` realizującej algorytm zapisany w pseudokodzie na s. 64, wykorzystujący funkcję `Palindrom`, może być następująca:

II część kodu źródłowego programu wypisującego słowa palindromy ze zdania – funkcja `main`

```
19. int main()
20. {
21.     int i;
22.     string wyraz, zdanie;
23.     cout<<"Podaj zdanie: ";
24.     getline(cin,zdanie);
25.     zdanie=zdanie+' ';
26.     while (zdanie.size()>0)
27.     {
28.         i=zdanie.find(' ');
29.         if (i>0)
30.         {
31.             wyraz=zdanie.substr(0,i);
32.             if (Palindrom(wyraz)) cout<<wyraz<<endl;
33.         }
34.         zdanie.erase(0,i+1);
35.     }
36.     return 0;
37. }
```

Aby podane przez użytkownika zdanie wczytać do programu, użyliśmy **funkcji** `getline` (linia 24). Pierwszym parametrem tej funkcji jest miejsce, z którego pobieramy dane – w naszym przypadku jest to klawiatura, dlatego parametrem jest `cin`. Drugim parametrem funkcji jest zmienna, w której zapiszemy dane. Gdybyśmy do wczytania danych użyli instrukcji `cin>>zdanie`, program zapisałby tylko fragment napisu do wystąpienia pierwszej spacji. Funkcja `getline` wczytuje cały wiersz danych.

Funkcja `getline`

Po wczytaniu zdania na końcu dodawana jest spacja – **wartownik**. Pętla dzieląca zdanie na wyrazy (linie 26–35) korzysta z metod operujących na napisach, dostępnych w klasie `string`: `size`, `find`, `substr`, `erase`. O **metodzie** `size` mówiliśmy w temacie 2. Informacje o **metodach** `find`, `substr`, `erase` znajdziesz w tabeli 4.1. Należy pamiętać, że przy stosowaniu metod obowiązuje notacja z kropką, tzn. `nazwa_zmiennej.nazwa_metody`.

Wartownik, s. 64

Metoda `size`, s. 39

Metody `find`, `substr`, `erase`

Metoda	Parametry	Wynik	Przykład zastosowania metody dla zmiennej `s` typu `string` `s="Ala ma kota"`

`find(szukane)`	`szukane` – fraza lub znak	Pozycja pierwszego wystąpienia szukanej frazy w napisie. Jeśli dany fragment nie występuje, wynikiem metody jest liczba spoza zakresu `0..size()-1`.	Instrukcja:
			`cout<<s.find("ma");`
			Wynik:
			`4`

`substr(pozycja,ile)`	`pozycja` – indeks znaku w napisie,
	`ile` – liczba znaków
		Fragment ciągu znaków zapisanych od indeksu `pozycja`. Liczbę znaków określa parametr `ile`.	Instrukcja:
			`cout<<s.substr(4,2);`
			Wynik:
			`ma`

`erase(pozycja,ile)`	`pozycja` – indeks znaku w napisie,
	`ile` – liczba znaków
		Napis bez ciągu znaków o długości `ile`, liczonego od indeksu `pozycja`.	Instrukcje:
			`s.erase(4,3);`
			`cout<<s;`
			Wynik:
			`Ala kota`

Tabela 4.1. Parametry i wyniki metod `find`, `substr`, `erase` oraz przykłady ich zastosowania

W linii 28 poszukiwane jest pierwsze wystąpienie znaku spacji. Jeśli nie jest to pierwszy znak zdania (a więc na lewo od spacji znajdują się litery tworzące wyraz), to ten wyraz jest zapamiętywany w zmiennej `wyraz` (linia 31). Jeśli wyraz jest palindromem, to zostanie wypisany na ekranie (linia 32). Ostatnia instrukcja w pętli (linia 34) odpowiada za usunięcie ze zdania zbadanego wyrazu wraz ze spacją (lub samej spacji, gdy jest ona pierwszym znakiem napisu).

Ćwiczenie 2

Napisz program znajdujący wyrazy palindromy w zdaniu wpisanym z klawiatury (niezależnie od tego, czy w zapisie użyto małych czy wielkich liter). W programie zdefiniuj funkcję sprawdzającą, czy wyraz jest palindromem.

Zapamiętaj

Aby wczytać do programu wiersz danych, w których znajdują się spacje, należy użyć funkcji `getline`. Funkcja ta wymaga podania dwóch parametrów. Pierwszy określa miejsce, z którego pobieramy dane, natomiast drugim jest zmienna, w której zapiszemy dane. Na przykład do zapisania w zmiennej `s` zdania podanego z klawiatury zastosujemy instrukcję `getline(cin,s)`. Instrukcja `cin>>s` wczyta dane do znaku spacji lub znaku końca wiersza.

Modyfikacja funkcji `Palindrom`

W definicji funkcji `Palindrom` nie musimy używać zmiennej pomocniczej `p` typu logicznego. Zamiast niej można zastosować wielokrotnie instrukcję **`return`**.

Zmodyfikowana funkcja `Palindrom`

```
7.  bool Palindrom(string wyraz)
8.  {
9.      int i=0, j=wyraz.size()-1;
10.     while (i<j)
11.         if (toupper(wyraz[i])==toupper(wyraz[j]))
12.         {
13.             i++; j--;
14.         }
15.         else return false;
16.     return true;
17. }
```

Jeśli wyraz jest palindromem, to wynikiem funkcji jest wartość **`true`** (linia 16). Natomiast gdy znaki w porównywanej parze okażą się różne (warunek logiczny w linii 11 przyjmie wartość **`false`**), funkcja zakończy działanie, a jej wynikiem będzie **`false`** (linia 15).

Ćwiczenie 3

Program znajdujący wyrazy palindromy w zdaniu zmodyfikuj tak, aby w funkcji sprawdzającej, czy wyraz jest palindromem, nie była użyta zmienna pomocnicza typu logicznego.

4.3. Sprawdzamy, czy zdanie jest palindromem

Teraz napiszemy program, który sprawdzi, czy całe zdanie jest palindromem. Przykładami tego typu palindromów w języku polskim są zdania „Kobyła ma mały bok" i „A to kanapa pana kota".

Warto wiedzieć

Zdanie „Kobyła ma mały bok" uchodzi obecnie za najbardziej znany polski palindrom. Pochodzi z XIX w. i nie wiadomo, kto jest jego autorem.

Specyfikacja

Dane: `zdanie` – napis złożony z liter alfabetu łacińskiego, spacji oraz znaków przestankowych.

Wynik: słowo „TAK", jeśli `zdanie` jest palindromem, słowo „NIE" – w przeciwnym przypadku.

Żeby zbadać, czy zdanie jest palindromem, usuniemy z napisu wszystkie znaki inne niż litery. Dzięki temu, podobnie jak w programie badającym, czy wyraz jest palindromem, sprawdzimy, czy napis złożony z samych liter jest palindromem. Na przykład dla napisu „Kobylamamalybok" program wypisze słowo „TAK".

W kodzie podanym poniżej w liniach 7–14 zdefiniowano funkcję `TylkoLitery`, której parametrem jest zdanie, a wynikiem – napis złożony z samych liter (funkcja usuwa znaki inne niż litery). W dalszych liniach znajduje się kod źródłowy funkcji `main`, która wypisuje informację, czy zdanie jest palindromem.

Kod źródłowy programu sprawdzającego, czy zdanie jest palindromem

```
1.  #include <iostream>
2.  #include <string>
3.  #include <cctype>
4.
5.  using namespace std;
6.
7.  string TylkoLitery(string s)
8.  {
9.      int i=0;
10.     while (i<s.size())
11.         if (toupper(s[i])>='A' && toupper(s[i])<='Z') i++;
12.         else s.erase(i,1);
13.     return s;
14. }
15.
16. int main()
17. {
18.     string zdanie, wyraz;
19.     int i=0, j;
20.     bool palindrom=true;
21.     cout<<"Podaj zdanie: "; getline(cin,zdanie);
22.     wyraz=TylkoLitery(zdanie);
23.     j=wyraz.size()-1;
```

Dobra rada

Do sprawdzenia, czy znak jest literą, możesz też wykorzystać funkcję `isalpha` z biblioteki `cctype`.

Dobra rada

Pamiętaj, że kompilator języka C++ rozróżnia wielkie i małe litery. Jeśli deklarujesz zmienną lub funkcję zawierającą w nazwie wielkie litery, to w dalszej części kodu musisz konsekwentnie używać ustalonej nazwy.

```
24.     while (palindrom && i<j)
25.         if (toupper(wyraz[i])==toupper(wyraz[j]))
26.         {
27.             i++; j--;
28.         }
29.         else palindrom=false;
30.     if (palindrom) cout<<"TAK";
31.     else cout<<"NIE";
32.     return 0;
33. }
```

W pętli **while** (linie 10–12) w funkcji `TylkoLitery` badane są kolejne znaki napisu. Jeśli aktualny znak jest małą lub wielką literą alfabetu łacińskiego, to powiększany jest indeks `i` (linia 11). W przeciwnym przypadku znak usuwany jest z napisu (linia 12). Wykorzystano do tego **metodę** `erase`, która usuwa z danego napisu określoną liczbę znaków (drugi parametr), zaczynając od konkretnej pozycji (pierwszy parametr). Zdanie do programu wczytuje **funkcja** `getline` (linia 21), podobnie jak w programie znajdującym słowa palindromy w zdaniu.

Metoda `erase`, s. 67

Funkcja `getline`, s. 67

Ćwiczenie 4

Napisz program sprawdzający, czy zdanie jest palindromem. Sprawdź działanie programu dla różnych zdań. Przykładów palindromów możesz poszukać w internecie.

A to ciekawe

Palindromy w kulturze i sztuce

Wymyślanie palindromów traktuje się najczęściej jako zabawę, choć mogą one służyć do przekazania jakiejś ważnej myśli. Zdjęcie obok przedstawia część fontanny z jednym z najstarszych znanych palindromów. Grecki napis oznacza „zmyj grzechy, nie tylko twarz". Za najstarszy polski palindrom uważa się pochodzące z XVII w. zdanie autorstwa księdza Wojciecha Waśniowskiego: „Co w onei? ow pokoy y okop, woien owoc". Palindromami zajmowali się także polscy poeci, m.in. Julian Tuwim, który w książce *Pegaz dęba* zebrał wiele różnych gier słownych, w tym palindromów. Tadeusz Morawski, profesor Politechniki Warszawskiej, jest czołowym polskim twórcą palindromów. Wydał kilkanaście książek je zawierających, założył też Muzeum Palindromów w Nowej Wsi koło Serocka.

Podsumowanie

- Palindrom to słowo, wyrażenie lub zdanie, które brzmi tak samo czytane od lewej do prawej i od prawej do lewej.
- Definicja funkcji w języku C++ składa się z dwóch głównych części: nagłówka funkcji oraz bloku instrukcji. W nagłówku podajemy typ funkcji, jej nazwę oraz listę parametrów formalnych (lista zawiera typ i nazwę każdego z parametrów).
- Parametry formalne to parametry podane w definicji funkcji.
- Parametry aktualne to parametry, dla których funkcja jest wywołana.
- Typy parametrów aktualnych muszą być zgodne z typami parametrów formalnych.
- Zapisanie podproblemu w postaci funkcji poprawia czytelność programu i umożliwia obliczanie wartości funkcji dla różnych wartości parametrów.
- Wynikiem funkcji `toupper` dla parametru będącego literą jest kod ASCII wielkiej litery, natomiast wynikiem funkcji `tolower` dla parametru będącego literą jest kod ASCII małej litery.
- Specjalny element ułatwiający odnalezienie końca danych nazywamy wartownikiem.
- Aby wczytać wiersz danych ze spacjami, należy użyć funkcji `getline`.
- Dla zmiennych typu `string` dostępnych jest wiele użytecznych metod umożliwiających pracę z napisami, np. `size` – badanie długości napisu, `substr` – znajdowanie fragmentu napisu, `erase` – usuwanie fragmentu napisu, `find` – znajdowanie pierwszego wystąpienia fragmentu napisu.

Zadania

* **1** Napisz program, który wczyta napis i sprawdzi, czy składa się on wyłącznie z małych liter alfabetu łacińskiego. Program powinien wypisać słowo „TAK" lub „NIE".

* **2** Napisz program wczytujący zdanie złożone z liter alfabetu łacińskiego oraz spacji i wypisujący liczbę wyrazów w zdaniu.

* **3** Napisz program, który po wczytaniu zdania złożonego z liter alfabetu łacińskiego oraz spacji wypisze zdanie, w którym pierwsze litery każdego z wyrazów będą wielkie, a pozostałe litery w wyrazach – małe.

** **4** Napisz program, który po wczytaniu zdania złożonego z liter alfabetu łacińskiego oraz spacji sprawdzi, czy wszystkie wyrazy w tym zdaniu składają się z tej samej liczby liter oraz czy wszystkie są palindromami. Jeśli oba warunki będą spełnione, program wypisze słowo „TAK", w przeciwnym przypadku – słowo „NIE".

** **5** Napisz program, który wypisze wszystkie trzycyfrowe liczby palindromiczne.
Uwaga: Liczba palindromiczna to liczba całkowita nieujemna, która czytana od lewej do prawej i od prawej do lewej ma taką samą wartość.

** **6** Napisz program wypisujący wszystkie czterocyfrowe liczby palindromiczne.
Uwaga: Liczba palindromiczna to liczba całkowita nieujemna, która czytana od lewej do prawej i od prawej do lewej ma taką samą wartość.

** **7** Napisz funkcję `string UpString(string s)`, która dla napisu złożonego z małych i wielkich liter alfabetu łacińskiego zwróci napis zapisany wielkimi literami, np. dla parametru aLa zwróci wartość ALA. Przetestuj działanie funkcji w programie sprawdzającym, czy wyraz jest palindromem.

*** **8** Napisz program, który wczyta słowo złożone z wielkich liter alfabetu łacińskiego i wypisze najdłuższy ciąg kolejnych liter zawarty w tym słowie, który brzmi tak samo czytany od lewej do prawej i od prawej do lewej, a więc ma cechę palindromu (ale nie musi być wyrazem). Jeśli takie ciągi będą dwa – program wypisze ten, który występuje wcześniej. Na przykład dla słowa RABATA program powinien wypisać ciąg ABA.

*** **9** Napisz program wczytujący liczby całkowite a i b, gdzie $0 \leqslant a \leqslant b < 10^{10}$, i wypisujący, ile liczb palindromicznych jest w przedziale $[a; b]$.
Uwaga: Liczba palindromiczna to liczba całkowita nieujemna, która czytana od lewej do prawej i od prawej do lewej ma taką samą wartość.

5. Czy ta liczba jest pierwsza?

Liczby pierwsze fascynowały ludzi już w czasach starożytnych. Wtedy to udowodniono, że takich liczb jest nieskończenie wiele, oraz podano sposób ich wyznaczania. Przez kolejne lata formułowano nowe własności i twierdzenia dotyczące liczb pierwszych, jednak często uchodziły one za piękne, ale bezużyteczne. Obecnie liczby te mają bardzo konkretne zastosowania, np. w kryptografii. Z ich własności korzysta się m.in. przy tworzeniu zabezpieczeń do kont bankowych i w celu zapewnienia poufności komunikacji w sieci. W tym temacie poznasz podstawowe algorytmy związane z liczbami pierwszymi.

Cele lekcji

- Poznasz podstawowe własności liczb pierwszych.
- Sprawdzisz, czy dana liczba jest pierwsza.
- Rozłożysz liczbę na czynniki pierwsze.
- Dowiesz się, czym są liczby bliźniacze i jak je wyznaczyć.

5.1. Liczby pierwsze i złożone

Liczba pierwsza to liczba całkowita dodatnia, która ma dokładnie dwa różne dzielniki: liczbę 1 i samą siebie. Zgodnie z tą definicją liczba 1 nie jest liczbą pierwszą. Liczbę większą od 1, która nie jest pierwsza, nazywamy **liczbą złożoną**.

Liczba pierwsza

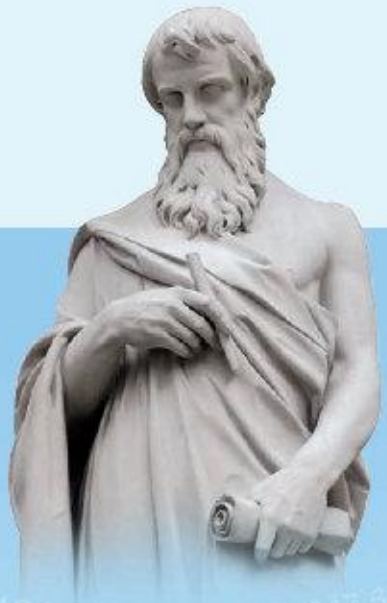
Liczba złożona

Wśród liczb mniejszych od 100 jest dwadzieścia pięć liczb pierwszych: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97. Na rysunku 5.1 oznaczono je niebieskim tłem.

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	

Rys. 5.1. Liczby pierwsze (zaznaczone na niebiesko) i liczby złożone mniejsze od 100

Co warto wiedzieć o liczbach pierwszych?



Ile jest liczb pierwszych?

Liczby pierwsze odkryto i badano już w starożytności. W IV w. p.n.e. Euklides udowodnił, że jest ich nieskończenie wiele. Dotąd nie odkryto jednak wzoru, według którego można byłoby dla konkretnej liczby n wyznaczyć n-tą liczbę pierwszą.

2 to najmniejsza i jedyna parzysta liczba pierwsza

Liczby postaci 2^n-1, gdzie n jest liczbą naturalną, nazywamy liczbami Mersenne'a. Wiele z nich jest liczbami pierwszymi

Liczby pierwsze większe od 5 kończą się cyframi: 1, 3, 7, 9

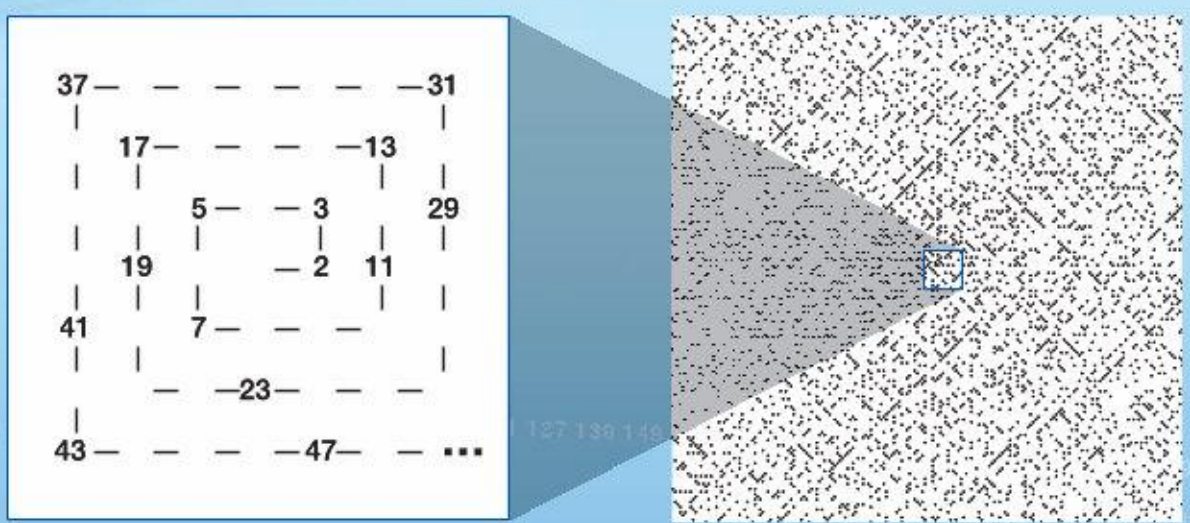
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Liczby czworacze to cztery liczby pierwsze mające postać: $p, p+2, p+6, p+8$

Liczby bliźniacze to dwie liczby pierwsze, których różnica wynosi 2

Spirala Ulama

W 1963 r. polski matematyk Stanisław Ulam przedstawił graficznie rozmieszczenie liczb pierwszych. Na kwadratowej tablicy spiralnie wypisał kolejne liczby naturalne. Po naniesieniu bardzo wielu liczb okazało się, że na niektórych liniach liczby pierwsze grupują się częściej niż na innych. Do tej pory nie wyjaśniono, dlaczego tak się dzieje.



5.2. Sprawdzamy, czy liczba jest pierwsza

Żeby sprawdzić, czy liczba jest pierwsza, będziemy poszukiwać jakiegoś jej dzielnika różnego od 1 i od niej samej. Spróbujemy więc wykazać, że liczba jest złożona. Jeśli nie znajdziemy odpowiedniego dzielnika, badana liczba jest liczbą pierwszą. Oto specyfikacja problemu:

Specyfikacja

Dane: n – liczba całkowita, $0 < n \leqslant 2^{31} - 1$.

Wynik: słowo „TAK", gdy n jest liczbą pierwszą, słowo „NIE" – w przeciwnym przypadku.

Sprawdzanie, czy liczba jest pierwsza – algorytm 1

Dzielników liczby n większych od 1 i mniejszych od n poszukiwaliśmy podczas rozwiązywania **problemu podziału na równoliczne grupy**. Zastosowane tam algorytmy można zmodyfikować tak, aby pozwalały określić, czy dana liczba jest pierwsza. Wtedy znajdowaliśmy wszystkie dzielniki właściwe większe od 1, teraz wystarczy znaleźć tylko jeden taki dzielnik. W pierwszym z algorytmów poszukiwaliśmy dzielników wśród liczb całkowitych z zakresu od 2 do połowy n. Drugi algorytm przeglądał liczby, zaczynając od 2, a kończąc, gdy $d \cdot d \leqslant n$, czyli $d \leqslant \sqrt{n}$. W tym przypadku maksymalna liczba sprawdzeń jest mniejsza, dlatego skorzystamy z tego rozwiązania. Zapis algorytmu w pseudokodzie może wyglądać następująco:

Problem 4. Podział na równoliczne grupy, s. 13–14

```
jeśli n > 1 to pierwsza ← prawda
w przeciwnym przypadku pierwsza ← fałsz
d ← 2
dopóki pierwsza oraz (d * d ≤ n) wykonuj
    jeśli n mod d = 0 to pierwsza ← fałsz
    w przeciwnym przypadku d ← d + 1
jeśli pierwsza to wypisz "TAK"
w przeciwnym przypadku wypisz "NIE"
```

Dobra rada

Instrukcję warunkową zawartą w dwóch pierwszych liniach pseudokodu możesz zapisać krócej:
`pierwsza ← (n > 1)`

Na początku zakładamy, że jeśli liczba n jest większa od 1, to jest liczbą pierwszą – zmienna `pierwsza` typu logicznego przyjmuje wtedy wartość **prawda**. Dla n równego 1 zmienna `pierwsza` przyjmie wartość **fałsz**. W pętli poszukujemy dzielnika `d`. Wartość początkowa tej zmiennej jest równa 2. Zwróć uwagę na warunek powtarzania pętli. Zmienna `pierwsza` musi mieć wartość **prawda** (tzn. jeszcze nie został znaleziony dzielnik) i jednocześnie wartość zmiennej `d` nie może przekraczać wartości $\sqrt{n}$. Warunek $d \leqslant \sqrt{n}$ zapisaliśmy jako `d * d ≤ n`. Unikamy w ten sposób obliczania pierwiastka kwadratowego i wykonywania operacji na liczbach rzeczywistych. Jeśli zostanie znaleziony dzielnik `d`, zmiennej `pierwsza` zostanie przypisana wartość logiczna **fałsz**, co spowoduje zakończenie wykonywania pętli.

Jeśli zmienna pierwsza zachowa wartość **prawda** do momentu, gdy pętla zakończy działanie po sprawdzeniu wszystkich potencjalnych dzielników, to badana liczba jest pierwsza.

Kod źródłowy programu może wyglądać następująco:

Kod źródłowy programu sprawdzającego, czy liczba jest pierwsza – algorytm 1

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  int main()
6.  {
7.      int n, d=2;
8.      bool pierwsza;
9.      cout<<"n = "; cin>>n;
10.     if (n>1) pierwsza=true;
11.     else pierwsza=false;
12.     while (pierwsza && d*d<=n)
13.         if (n%d==0) pierwsza=false;
14.         else d++;
15.     if (pierwsza) cout<<"TAK";
16.     else cout<<"NIE";
17.     return 0;
18. }
```

Sprawdzanie, czy liczba jest pierwsza – algorytm 2

Zmodyfikujemy podany wcześniej algorytm tak, aby pętla wykonała mniej iteracji. Zauważ, że jedyną parzystą liczbą pierwszą jest liczba 2. Każda kolejna liczba podzielna przez 2 jest liczbą złożoną. Zatem przed pętlą możemy sprawdzić, czy liczba $n > 2$ jest liczbą parzystą. Jeśli nie będzie parzysta, w pętli poszukamy dzielnika wśród liczb nieparzystych z przedziału $[3; \sqrt{n}]$. Poniżej znajduje się zmodyfikowany algorytm zapisany w pseudokodzie. Pierwsza linia jest skrótową formą zapisu dwóch pierwszych linii z poprzedniego pseudokodu.

```
pierwsza ← (n > 1)
jeśli n > 2 oraz n mod 2 = 0 to pierwsza ← fałsz
d ← 3
dopóki pierwsza oraz (d * d ≤ n) wykonuj
    jeśli n mod d = 0 to pierwsza ← fałsz
    w przeciwnym przypadku d ← d + 2
jeśli pierwsza to wypisz "TAK"
w przeciwnym przypadku wypisz "NIE"
```

Ćwiczenie 1

Utwórz program sprawdzający, czy liczba jest pierwsza. Uwzględnij w nim poszukiwanie dzielnika wśród liczb nieparzystych.

Sprawdzanie, czy liczba jest pierwsza – algorytm 3

Teraz usprawnimy algorytm 2. W tym celu przed pętlą oprócz podzielności przez 2 rozpatrzymy także podzielność przez 3. Ponadto chcielibyśmy zmniejszyć liczbę iteracji w pętli. Aby to zrobić, wystarczyłoby szukać dzielników wśród liczb pierwszych. Ponieważ jednak wyznaczenie wszystkich tych liczb byłoby kłopotliwe, możemy ograniczyć sprawdzane dzielniki do liczb, które potencjalnie mogą być liczbami pierwszymi. Zwróć uwagę, że co druga liczba podzielna przez 3 jest też podzielna przez 2. Najmniejszą liczbą podzielną jednocześnie przez 2 i 3 jest liczba 6. Kolejne sześć liczb większych od 5 można zapisać w postaci $6i$, $6i + 1$, $6i + 2$, $6i + 3$, $6i + 4$, $6i + 5$, gdzie i jest liczbą całkowitą dodatnią. Przyjrzyj się tabeli 5.1.

Wzór	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$
$6i$	6	12	18	24	30
$6i + 1$	7	13	19	25	31
$6i + 2$	8	14	20	26	32
$6i + 3$	9	15	21	27	33
$6i + 4$	10	16	22	28	34
$6i + 5$	11	17	23	29	35

Tabela 5.1. Liczby całkowite większe od 5 opisane wzorami

Liczbami pierwszymi na pewno nie są liczby o postaci $6i$, $6i + 2$, $6i + 3$, $6i + 4$. Liczby o postaci $6i$ są podzielne przez 2 i 3, liczby $6i + 2$ oraz $6i + 4$ są podzielne przez 2, a liczby $6i + 3$ – przez 3. Zatem dzielników większych od 5 należy szukać wśród liczb, które można przedstawić jako $6i + 1$ lub $6i + 5$. Wzór $6i + 5$ jest równoważny zapisowi $6i - 1$. Liczbę pierwszą większą od 3 można więc przedstawić w postaci $6i + 1$ lub $6i - 1$. Oto zmodyfikowany algorytm zapisany w pseudokodzie oraz kod źródłowy programu. Dzielników o postaci $6i - 1$ oraz $6i + 1$ będziemy szukać w przedziale $[5; \sqrt{n}]$.

```
pierwsza ← (n > 1)
jeśli n > 2 oraz n mod 2 = 0 to pierwsza ← fałsz
jeśli n > 3 oraz n mod 3 = 0 to pierwsza ← fałsz
d ← 5
dopóki pierwsza oraz (d * d ≤ n) wykonuj
    jeśli n mod d = 0 to pierwsza ← fałsz
    w przeciwnym przypadku
        jeśli n mod (d + 2) = 0 to pierwsza ← fałsz
        w przeciwnym przypadku d ← d + 6
jeśli pierwsza to wypisz "TAK"
w przeciwnym przypadku wypisz "NIE"
```

Kod źródłowy programu sprawdzającego, czy liczba jest pierwsza – algorytm 3

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  int main()
6.  {
7.      int n, d=5;
8.      bool pierwsza;
9.      cout<<"n = "; cin>>n;
10.     pierwsza=(n>1);
11.     if (n>2 && n%2==0) pierwsza=false;
12.     if (n>3 && n%3==0) pierwsza=false;
13.     while (pierwsza && (d*d)<=n)
14.         if (n%d==0) pierwsza=false;
15.         else if (n%(d+2)==0) pierwsza=false;
16.              else d=d+6;
17.     if (pierwsza) cout<<"TAK";
18.     else cout<<"NIE";
19.     return 0;
20. }
```

W jednej iteracji pętli sprawdzane są dwa dzielniki (linie 14–15). Aktualnej wartości zmiennej d odpowiada dzielnik o postaci $6i - 1$, a wartości d+2 – dzielnik o postaci $6i + 1$.

Ćwiczenie 2

Napisz i przetestuj program sprawdzający, czy podana liczba jest pierwsza. Wykorzystaj w nim poszukiwanie dzielników wśród liczb o postaci $6i - 1$ oraz $6i + 1$.

Dobra rada

Podczas rozwiązywania problemu staraj się dobrać taki algorytm, który będzie wymagał wykonania jak najmniejszej liczby operacji.

W omówionych algorytmach 1, 2 i 3 najczęściej wykonywaną operacją jest wyznaczanie reszty z dzielenia. Tabela 5.2 przedstawia liczbę tych operacji (łącznie z operacjami przed pętlą) dla przykładowych liczb pierwszych w zależności od wersji algorytmu. Wybrane liczby to odpowiednio najmniejsza liczba pierwsza większa od: 10 000, 1 000 000 i 100 000 000. W tabeli jako „Algorytm 0" uwzględniono rozwiązanie naiwne, czyli sprawdzające wszystkie dzielniki z przedziału $[2; n/2]$.

Liczba pierwsza	Algorytm 0	Algorytm 1	Algorytm 2	Algorytm 3
10 007	5 002	99	50	34
1 000 003	500 000	999	500	334
100 000 007	50 000 002	9 999	5 000	3 334

Tabela 5.2. Liczba operacji wyznaczania reszty z dzielenia dla różnych algorytmów sprawdzających pierwszość liczby

Ćwiczenie 3

Dla każdego z algorytmów uwzględnionych w tabeli 5.2 podaj wzór, który pozwoli określić liczbę (rząd wielkości) operacji dzielenia z resztą wykonywanych dla liczby pierwszej n.

Zapamiętaj

Jeśli dana liczba jest liczbą pierwszą większą od 3, to da się ją przedstawić w postaci $6i - 1$ lub $6i + 1$, gdzie i jest liczbą całkowitą dodatnią.

5.3. Rozkładamy liczbę na czynniki pierwsze

Każdą liczbę złożoną można przedstawić w postaci iloczynu liczb pierwszych. Iloczyn ten nazywamy **rozkładem liczby na czynniki pierwsze** lub **faktoryzacją** liczby. Algorytm rozkładu liczby na czynniki pierwsze znasz z lekcji matematyki. Spójrz na przykłady.

Rozkład liczby na czynniki pierwsze (faktoryzacja liczby)

70	2
35	5
7	7
1	

$70 = 2 \cdot 5 \cdot 7$

198	2
99	3
33	3
11	11
1	

$198 = 2 \cdot 3 \cdot 3 \cdot 11$

Aby rozłożyć liczbę na czynniki pierwsze, badamy resztę z dzielenia liczby przez kolejne liczby pierwsze, zaczynając od najmniejszej. Jeśli reszta z dzielenia wynosi zero, to znaleźliśmy czynnik pierwszy, a liczbę dzielimy przez ten dzielnik. Ponieważ w rozkładzie ten czynnik może występować wielokrotnie, nie zmieniamy dzielnika. Kolejnego kandydata na czynnik pierwszy bierzemy pod uwagę dopiero wtedy, gdy reszta z dzielenia jest różna od zera. Proces kończymy, gdy liczba osiągnie wartość 1.

Oto specyfikacja problemu faktoryzacji:

Specyfikacja

Dane: n – liczba całkowita, $0 < n \leqslant 2^{31} - 1$.

Wynik: rozkład liczby n na czynniki pierwsze, $n = p_1 \cdot p_2 \cdot \ldots \cdot p_m$, gdzie p_i dla $i \in \{1, 2, \ldots, m\}$ jest liczbą pierwszą oraz $p_1 \leqslant p_2 \leqslant \ldots \leqslant p_m$.

Ponieważ nie znamy wzoru, z którego możemy wyznaczać liczby pierwsze, w algorytmie rozwiązującym problem będziemy sprawdzać podzielność przez kolejne liczby, zaczynając od 2. Zauważ, że gdy dzielnikiem będzie liczba złożona, reszta z dzielenia będzie różna od zera. Wynika to stąd, że przez czynniki pierwsze tej liczby były już wykonane próby dzielenia. Oto zapis algorytmu w pseudokodzie.

Warto wiedzieć

Podstawą niektórych współczesnych metod szyfrowania jest fakt, że nie znamy szybkich algorytmów faktoryzacji bardzo dużych liczb. W części trzeciej podręcznika omówimy jeden z takich algorytmów szyfrowania.

Warto wiedzieć

Rozkład dowolnej liczby na czynniki pierwsze jest jednoznaczny (np. dla 30 jest to wyłącznie rozkład $2 \cdot 3 \cdot 5$).

```
d ← 2
dopóki n > 1 wykonuj
    jeśli n mod d = 0 to
        wypisz d
        n ← n div d
    w przeciwnym przypadku d ← d + 1
```

Faktoryzację można zakończyć wcześniej. Jeśli wartość zmiennej d podniesiona do kwadratu będzie większa od aktualnej wartości zmiennej n, to liczba n nie będzie miała już dzielników, więc jest liczbą pierwszą. Po pętli należy wypisać aktualną wartość zmiennej n jako ostatni czynnik pierwszy. Oto zmodyfikowany pseudokod i kod źródłowy programu:

```
d ← 2
dopóki d * d ≤ n wykonuj
    jeśli n mod d = 0 to
        wypisz d
        n ← n div d
    w przeciwnym przypadku d ← d + 1
wypisz n
```

Kod źródłowy programu wypisującego czynniki pierwsze liczby

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  int main()
6.  {
7.      int n, d=2;
8.      cout<<"n = "; cin>>n;
9.      cout<<n<<" = ";
10.     while (d*d<=n)
11.         if (n%d==0)
12.         {
13.             cout<<d<<" * ";
14.             n=n/d;
15.         }
16.         else d++;
17.     cout<<n;
18.     return 0;
19. }
```

Rysunek 5.2 przedstawia przykładowe wywołanie programu.

```
n = 456
456 = 2 * 2 * 2 * 3 * 19
```

Rys. 5.2. Przykład wywołania programu rozkładającego liczbę na czynniki pierwsze

Ćwiczenie 4

Program wypisujący czynniki pierwsze podanej liczby naturalnej zmodyfikuj tak, aby ewentualny czynnik pierwszy 2 rozpatrzyć w oddzielnej pętli, a w pętli głównej powiększać wartość zmiennej d o 2.

5.4. Liczby bliźniacze

Liczby bliźniacze to dwie liczby pierwsze, których różnica wynosi 2. Przykładem takiej pary są liczby 11 i 13. Teraz zajmiemy się wyznaczaniem podanej liczby par liczb bliźniaczych. Oto specyfikacja problemu:

Liczby bliźniacze

Specyfikacja

Dane: n – liczba całkowita, $0 < n \leqslant 100$.

Wynik: n początkowych par liczb bliźniaczych.

Warto wiedzieć

Liczby 3, 5, 7 są jedyną trójką kolejnych liczb pierwszych, w której są dwie pary liczb bliźniaczych.

Algorytm rozwiązujący ten problem może sprawdzać pierwszość kolejnych liczb nieparzystych i wybierać spośród nich te, które różnią się o dwa. Do sprawdzenia, czy liczba jest pierwsza, wykorzystamy funkcję `Pierwsza`, której parametrem będzie liczba całkowita dodatnia, a wynikiem – wartość logiczna określająca, czy parametr jest liczbą pierwszą czy nie. Oto zapis tego algorytmu w pseudokodzie:

```
licznik ← 0
x ← 3
dopóki licznik < n wykonuj
    jeśli Pierwsza(x) oraz Pierwsza(x + 2) to
        wypisz x i x + 2
        licznik ← licznik + 1
    x ← x + 2
```

Wiesz już, że jeśli liczba większa od 3 jest liczbą pierwszą, to można ją przedstawić w postaci $6i - 1$ lub $6i + 1$. Zatem para liczb bliźniaczych, z wyjątkiem pary 3 i 5, będzie mieć postać $6i - 1$ oraz $6i + 1$. Oto algorytm wykorzystujący tę własność, zapisany w pseudokodzie:

```
licznik ← 1
wypisz 3 i 5
x ← 5
dopóki licznik < n wykonuj
    jeśli Pierwsza(x) oraz Pierwsza(x + 2) to
        wypisz x i x + 2
        licznik ← licznik + 1
    x ← x + 6
```

Warto wiedzieć

W latach 90. XX w. pewien producent procesorów był zmuszony wymienić jeden z modeli, ponieważ wykonywał on operacje dzielenia z mniejszą dokładnością, niż zakładano. Wynosiła ona 9 zamiast 19 cyfr po przecinku. Wada ta ujawniła się przy szacowaniu tzw. stałej Bruna, która dotyczy liczb bliźniaczych.

Funkcja `Pierwsza` może np. realizować trzeci algorytm badający pierwszość liczby, omówiony na s. 77. Zapiszmy kod źródłowy programu zawierający definicję tej funkcji. Zmienną logiczną przechowującą informację o tym, czy liczba jest pierwsza, nazwijmy `p`.

I część kodu źródłowego programu wypisującego liczby bliźniacze – definicja funkcji `Pierwsza` (sposób 1)

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  bool Pierwsza(int n)
6.  {
7.      int d=5;
8.      bool p=(n>1);
9.      if (n>2 && n%2==0) p=false;
10.     if (n>3 && n%3==0) p=false;
11.     while (p && d*d<=n)
12.         if (n%d==0) p=false;
13.         else if (n%(d+2)==0) p=false;
14.              else d=d+6;
15.     return p;
16. }
17.
```

Ponieważ możemy wielokrotnie stosować instrukcję **`return`**, funkcję `Pierwsza` można też zdefiniować bez użycia zmiennej pomocniczej `p` typu logicznego.

I część kodu źródłowego programu wypisującego liczby bliźniacze – definicja funkcji `Pierwsza` (sposób 2)

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  bool Pierwsza(int n)
6.  {
7.      int d=5;
8.      if (n==1) return false;
9.      if (n>2 && n%2==0) return false;
10.     if (n>3 && n%3==0) return false;
11.     while (d*d<=n)
12.         if (n%d==0) return false;
13.         else if (n%(d+2)==0) return false;
14.              else d=d+6;
15.     return true;
16. }
17.
```

Jeśli parametr nie jest liczbą pierwszą (ma wartość 1 lub został znaleziony dzielnik), wynikiem funkcji jest wartość **`false`** i funkcja kończy działanie (instrukcje **`return false`** w liniach 8–13). Jeśli pętla zakończy się naturalnie, czyli warunek pętli `d*d<=n` przyjmie wartość **`false`**, to wynikiem funkcji jest wartość **`true`** (linia 15). Oznacza to, że liczba `n` jest liczbą pierwszą.

Poniżej znajduje się kod funkcji `main` wypisującej podaną liczbę par liczb bliźniaczych, wykorzystujący funkcję `Pierwsza`. Zwróć uwagę na linię 25. Dwukrotnie jest w niej wywołana funkcja `Pierwsza`. Przy pierwszym wywołaniu w miejsce **parametru formalnego** podstawiona jest bieżąca wartość zmiennej `x`, przy drugim wywołaniu – wartość `x+2`. Jeżeli funkcja `Pierwsza` dla parametru `x` zwróci wartość **`false`**, nie jest już obliczana wartość drugiego składnika koniunkcji (`Pierwsza(x+2)`), ponieważ **koniunkcja** dwóch zdań, z których jedno jest fałszywe, ma wartość **`false`**.

Parametr formalny, s. 65

Koniunkcja, s. 40

```
18. int main()
19. {
20.     int licznik=1, x=5, n;
21.     cout<<"Podaj liczbe par: "; cin>>n;
22.     cout<<3<<" "<<5<<endl;
23.     while (licznik<n)
24.     {
25.         if (Pierwsza(x) && Pierwsza(x+2))
26.         {
27.             cout<<x<<" "<<x+2<<endl;
28.             licznik++;
29.         }
30.         x=x+6;
31.     }
32.     return 0;
33. }
```

II część kodu źródłowego programu wypisującego liczby bliźniacze – definicja funkcji `main`

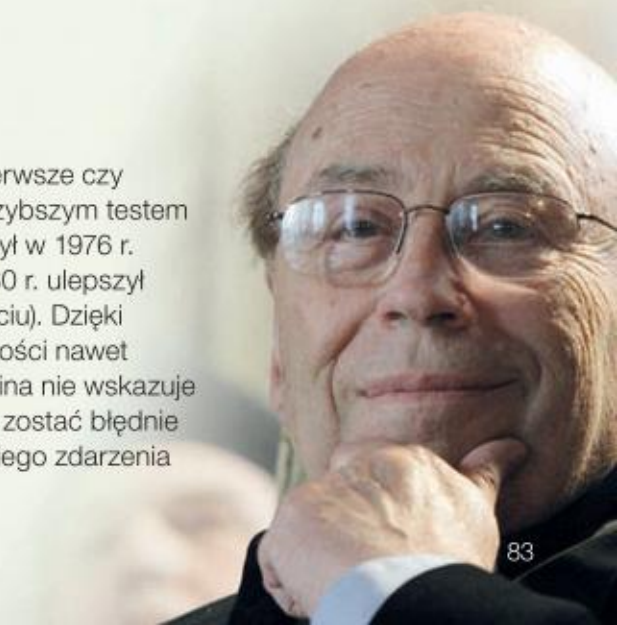
Ćwiczenie 5

Napisz program, który – wykorzystując funkcję `Pierwsza` – będzie znajdował pary liczb bliźniaczych.

A to ciekawe

Szybki test pierwszości

Dla bardzo dużych liczb trudno ustalić, czy są one pierwsze czy złożone. Powszechnie stosowanym i w praktyce najszybszym testem pierwszości jest test Millera–Rabina. Algorytm stworzył w 1976 r. amerykański profesor informatyki Gary Miller, a w 1980 r. ulepszył go urodzony we Wrocławiu Michael O. Rabin (na zdjęciu). Dzięki temu algorytmowi można przeprowadzić test pierwszości nawet dla ogromnych liczb, rzędu 150 cyfr. Test Millera–Rabina nie wskazuje żadnych dzielników testowanej liczby, więc może ona zostać błędnie rozpoznana jako pierwsza. Prawdopodobieństwo takiego zdarzenia jest jednak znikome – wynosi ok. 1 do 10^{12}.



Podsumowanie

- Liczba pierwsza to liczba całkowita dodatnia, która ma dokładnie dwa różne dzielniki: liczbę 1 i samą siebie.
- Liczbę całkowitą większą od 1, która nie jest pierwsza, nazywamy liczbą złożoną.
- Liczba 1 nie jest ani pierwsza, ani złożona.
- Każdą liczbę złożoną można przedstawić w postaci iloczynu $n = d_1 \cdot d_2$, gdzie $d_1 \leqslant \sqrt{n}$ i $d_2 \geqslant \sqrt{n}$.
- Każdą liczbę złożoną można przedstawić w postaci iloczynu czynników pierwszych. Iloczyn ten nazywamy rozkładem na czynniki pierwsze lub faktoryzacją liczby.
- Liczby pierwsze większe od 3 można przedstawić w postaci $6i - 1$ lub $6i + 1$, gdzie i jest liczbą całkowitą dodatnią.
- Żeby sprawdzić, czy liczba n jest liczbą pierwszą, wystarczy szukać dzielników wśród liczb pierwszych mniejszych lub równych $\sqrt{n}$.
- Liczby bliźniacze to dwie liczby pierwsze różniące się o 2.

Zadania

* **1** Napisz program, który wypisze liczby pierwsze mniejsze niż 100.

* **2** Napisz program, który wypisze 30 początkowych liczb pierwszych.

* **3** Napisz program, który wypisze liczbę czynników pierwszych podanej liczby całkowitej dodatniej.

* **4** Napisz program, który sprawdzi, czy podana liczba całkowita dodatnia jest liczbą półpierwszą. W zależności od otrzymanego wyniku program powinien wypisać słowo „TAK" lub „NIE".
Uwaga: Liczba półpierwsza to liczba będąca iloczynem dwóch liczb pierwszych. Liczbami półpierwszymi są np.: 4, 6, 9, 10, 14, 15.

** **5** Napisz program, który wypisze liczbę różnych czynników pierwszych podanej liczby całkowitej dodatniej, np. dla liczby 60 wypisze 3.

** **6** Napisz program, który wypisze najmniejszą liczbę pierwszą większą od podanej liczby całkowitej dodatniej.

** **7** Napisz program znajdujący cztery początkowe liczby doskonałe.
Uwaga: Liczba doskonała to liczba równa sumie wszystkich swoich dzielników właściwych. Liczbą doskonałą jest np. 28, bo $28 = 1 + 2 + 4 + 7 + 14$.

*** **8** Napisz program, który rozłoży wczytaną liczbę całkowitą dodatnią na czynniki pierwsze. W algorytmie uwzględnij to, że liczbę pierwszą można przedstawić w postaci $6i - 1$ lub $6i + 1$.

*** **9** Napisz program znajdujący trzy początkowe pary liczb zaprzyjaźnionych. Utwórz i wykorzystaj funkcję obliczającą sumę dzielników właściwych podanego parametru.
Uwaga: Liczby zaprzyjaźnione to takie dwie liczby całkowite dodatnie, z których każda jest równa sumie dzielników właściwych drugiej liczby (np. 220 i 284).

*** **10** Poszukaj w internecie informacji dotyczących związku liczb Mersenne'a z liczbami doskonałymi. Napisz program znajdujący pięć początkowych liczb doskonałych.
Uwaga: Liczba doskonała to liczba równa sumie wszystkich swoich dzielników właściwych. Liczbą doskonałą jest np. 28, bo $28 = 1 + 2 + 4 + 7 + 14$.

*** **11** Napisz program sprawdzający hipotezę Goldbacha dla podanej parzystej liczby całkowitej z zakresu od 2 do 100.
Uwaga: Hipoteza Goldbacha mówi, że każdą liczbę parzystą większą od 2 można przedstawić jako sumę dwóch liczb pierwszych.

6. Działania na liczbach w systemach innych niż dziesiętny

Czy wiesz, od jak dawna ludzkość potrafi wykonywać działania arytmetyczne? Do najstarszych znanych przedmiotów, które mogą świadczyć o wykonywaniu przez człowieka prostych obliczeń, należą kości z charakterystycznymi nacięciami. Przykładami takimi są znalezione w II połowie XX w. w Afryce kość z Ishango (sprzed ok. 20 tys. lat) i kość z Lebombo (sprzed ok. 37 tys. lat). Współcześnie do obliczeń często wykorzystuje się komputery. W tym temacie nauczysz się wykonywać działania arytmetyczne w systemach o różnych podstawach i napiszesz programy pozwalające wykonać wybrane działania. Najwięcej uwagi poświęcimy działaniom na liczbach binarnych.

Cele lekcji

- Przypomnisz sobie, jak dodać pisemnie dwie liczby binarne.
- Wykonasz działania arytmetyczne na liczbach zapisanych w różnych systemach pozycyjnych, innych niż dziesiętny.
- Napiszesz programy umożliwiające wykonanie wybranych działań arytmetycznych w różnych systemach pozycyjnych.

6.1. Dodawanie liczb w różnych systemach pozycyjnych

Reguły rządzące działaniami arytmetycznymi w różnych systemach liczbowych są takie same jak w znanym ci systemie dziesiętnym.

Dodawanie liczb w systemie dwójkowym

Kod uzupełnień do 2 (U2), s. 33–35

O tym, jak dodać pisemnie liczby całkowite zapisane binarnie, mówiliśmy w temacie 2 w części dotyczącej **kodu uzupełnień do 2 (U2)**. Teraz napiszemy program dodający dwie liczby binarne reprezentowane jako napisy złożone z zer i jedynek. Oto specyfikacja problemu:

Specyfikacja

Dane: a, b – napisy reprezentujące liczby całkowite nieujemne zapisane binarnie.

Wynik: c – napis reprezentujący liczbę binarną będącą sumą liczb a i b.

Aby łatwiej można było dodać liczby jednocyfrowe znajdujące się na tych samych pozycjach w liczbach a i b, liczby te powinny

być tej samej długości. Jeśli a i b składają się z różnej liczby cyfr, na początku krótszej z tych liczb dopiszemy zera nieznaczące. Poniżej znajduje się zapis algorytmu w pseudokodzie. Komentarze zawierają wyjaśnienie, jakie operacje są wykonywane w poszczególnych fragmentach. Zwróć uwagę na dwie instrukcje odpowiedzialne za dopisanie zer nieznaczących (zaczynają się od słowa **dopóki**). Dla liczb składających się z tej samej liczby cyfr żadna z pętli się nie wykona – warunki powtarzania pętli będą miały wartość **fałsz**. W przypadku liczb składających się z różnej liczby cyfr wykona się jedna pętla. Pod pseudokodem znajduje się kod programu realizującego ten algorytm.

```
przn ← 0 // nadanie przeniesieniu wartości 0
c ← ""   // określenie wartości wyniku - napis pusty
// dla liczb składających się z różnej liczby cyfr dopisanie
// na początku krótszej liczby zer nieznaczących
dopóki długość a < długość b wykonuj a ← '0' + a
dopóki długość b < długość a wykonuj b ← '0' + b
// dodanie liczb a i b
dla i ← długość a - 1, ..., 0 wykonuj
    suma ← przn + a[i] + b[i]
    jeśli suma mod 2 = 1 to c ← '1' + c
    w przeciwnym przypadku c ← '0' + c
    przn ← suma div 2
jeśli przn = 1 to c ← '1' + c
```

Kod źródłowy programu dodającego dwie liczby binarne

```
1.  #include <iostream>
2.  #include <string>
3.
4.  using namespace std;
5.
6.  int main()
7.  {
8.      int i, przn=0, suma;
9.      string a, b, c="";
10.     cout<<"Podaj 1 liczbe binarna: "; cin>>a;
11.     cout<<"Podaj 2 liczbe binarna: "; cin>>b;
12.     while (a.size()<b.size()) a='0'+a;
13.     while (b.size()<a.size()) b='0'+b;
14.     for (i=a.size()-1;i>=0;i--)
15.     {
16.         suma=przn+a[i]-'0'+b[i]-'0';
17.         if (suma%2==1) c='1'+c;
18.         else c='0'+c;
19.         przn=suma/2;
20.     }
21.     if (przn==1) c='1'+c;
22.     cout<<"Suma = "<<c;
23.     return 0;
24. }
```

W liniach 8–9 zadeklarowane są zmienne używane w programie oraz nadane są wartości początkowe przeniesienia i wyniku dodawania. Wczytanie danych realizują instrukcje w liniach 10–11. Jedna z pętli zapisanych w liniach 12–13 dopisze na początku krótszej liczby zera nieznaczące. Linie 14–20 realizują zasadniczą część algorytmu. W linii 16 wyznaczana jest suma odpowiednich cyfr liczb `a` i `b` oraz przeniesienia z poprzedniej pozycji. Ponieważ cyfry są znakami, są reprezentowane przez kody ASCII. Kod ASCII cyfry 0 jest równy 48, więc od kodu badanej cyfry należy odjąć `'0'` lub 48, aby otrzymać liczbę jednocyfrową. Na przykład jeśli na pozycji `i` liczby `a` znajduje się cyfra 1, to wartość `a[i]-'0'` jest równa 1. W liniach 17–18 do wyniku dołączana jest kolejna cyfra, a w linii 19 wyznaczana jest nowa wartość przeniesienia.

Dobra rada

Cyfrę 0 lub 1 do wyniku możesz też dołączyć bez użycia instrukcji warunkowej. Aby to zrobić, możesz zastosować instrukcję:
`c=char(suma%2+'0')+c;`

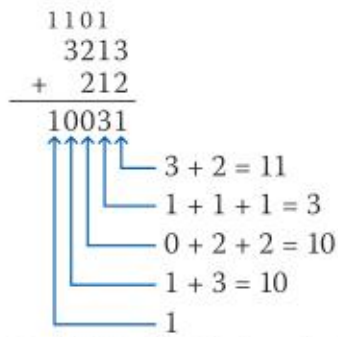
Dodawanie liczb w dowolnym systemie pozycyjnym

Aby szybciej wykonać dodawanie pisemne w dowolnym systemie pozycyjnym, można stworzyć tabelę dodawania liczb jednocyfrowych w tym systemie. Oto przykład takiej tabeli dla systemu czwórkowego:

+	0	1	2	3
0	0	1	2	3
1	1	2	3	10
2	2	3	10	11
3	3	10	11	12

Tabela 6.1. Tabela dodawania w systemie czwórkowym

Korzystając z tej tabeli, możemy szybko dodać np. liczby 3213_4 i 212_4. Działanie będzie wyglądało tak jak na rysunku 6.1. Suma liczb 3 i 2 w systemie czwórkowym wynosi 11, a więc w wyniku zapisujemy 1 i otrzymujemy przeniesienie 1. Suma trzech jedynek to 3 bez przeniesienia, suma dwóch dwójek wynosi 10, czyli 0 i przeniesienie 1 itd.



Rys. 6.1. Przykład dodawania liczb w systemie czwórkowym

Ćwiczenie 1

Przygotuj tabelę dodawania w systemie pozycyjnym o podstawie 6, a następnie dodaj pisemnie liczby 5324_6 i 444_6.

Program dodający dwie liczby całkowite nieujemne zapisane w systemie binarnym przekształcimy teraz tak, aby dodawał liczby w systemie pozycyjnym o podstawie z zakresu od 2 do 36. Ograniczenie podstawy

wynika z tego, że do zapisu liczb używamy maksymalnie 36 znaków (10 cyfr i 26 wielkich liter alfabetu). Oto specyfikacja problemu:

Specyfikacja

Dane: `podstawa` – podstawa systemu, $2 \leqslant$ `podstawa` $\leqslant 36$,
`a`, `b` – napisy reprezentujące liczby całkowite nieujemne w systemie pozycyjnym o podstawie `podstawa`.

Wynik: `c` – napis reprezentujący sumę liczb `a` i `b` w systemie pozycyjnym o podstawie `podstawa`.

W części algorytmu, w której dodajemy liczby `a` i `b`, zamiast dzielić przez 2, będziemy dzielić przez podstawę systemu. Ponadto w instrukcji odpowiadającej za dołączanie aktualnie otrzymanej cyfry do wyniku `c` musimy dokonać konwersji kodu ASCII na znak (cyfrę). Podobnie postępowaliśmy przy algorytmie **zamiany liczby dziesiętnej na liczbę w systemie pozycyjnym o wybranej podstawie**. Oto pętla algorytmu realizująca dodawanie liczb `a` i `b` zapisana w pseudokodzie:

Zamiana liczby dziesiętnej na liczbę w systemie pozycyjnym o wybranej podstawie, s. 53–55

```
dla i ← długość a - 1, ..., 0 wykonuj
    suma ← przn + a[i] + b[i]
    c ← cyfra(suma mod podstawa) + c
    przn ← suma div podstawa
```

Oto fragment kodu źródłowego programu odpowiadający powyższej pętli przy ograniczeniu podstawy systemu do zakresu od 2 do 10:

```
1. for (i=a.size()-1;i>=0;i--)
2. {
3.     suma=przn+a[i]-'0'+b[i]-'0';
4.     c=char(suma%podstawa+'0')+c;
5.     przn=suma/podstawa;
6. }
```

Fragment kodu źródłowego programu dodającego liczby w systemie pozycyjnym o podstawie z zakresu od 2 do 10 – pętla odpowiadająca za dodanie liczb

Ćwiczenie 2

Napisz program, który wczyta podstawę systemu pozycyjnego z zakresu od 2 do 10 oraz dwa napisy reprezentujące liczby całkowite nieujemne zapisane w tym systemie, a następnie wypisze sumę tych liczb.

Zmodyfikujemy teraz program tak, aby dodawał liczby całkowite nieujemne w systemie pozycyjnym o podstawie z zakresu od 2 do 36. Musimy w tym wypadku oddzielnie rozpatrywać dwie sytuacje: gdy cyfra liczby jest z zakresu od 0 do 9 oraz gdy jest reprezentowana przez literę. Dlatego wprowadzimy dwie pomocnicze zmienne: `cyfra1` i `cyfra2`, przechowujące aktualne cyfry jako liczby jednocyfrowe.

Oto zmodyfikowany fragment kodu źródłowego programu dodającego liczby. W liniach 3–6 wyznaczane są wartości zmiennych `cyfra1` i `cyfra2`. W liniach 10–11 następuje konwersja wartości zmiennej `suma` na odpowiedni znak oraz dołączenie znaku do wyniku.

Fragment kodu źródłowego programu dodającego liczby w systemie pozycyjnym o podstawie z zakresu od 2 do 36 – pętla odpowiadająca za dodanie liczb

```
1.  for (i=a.size()-1;i>=0;i--)
2.  {
3.      if (a[i]<='9') cyfra1=a[i]-'0';
4.      else cyfra1=a[i]-'A'+10;
5.      if (b[i]<='9') cyfra2=b[i]-'0';
6.      else cyfra2=b[i]-'A'+10;
7.      suma=przn+cyfra1+cyfra2;
8.      przn=suma/podstawa;
9.      suma=suma%podstawa;
10.     if (suma<10) c=char(suma+'0')+c;
11.     else c=char(suma+'A'-10)+c;
12. }
```

Ćwiczenie 3

Napisz program umożliwiający dodanie dwóch liczb zapisanych w systemie pozycyjnym o podstawie z zakresu od 2 do 36.

6.2. Odejmowanie liczb w różnych systemach pozycyjnych

Podobnie jak przy dodawaniu, najpierw zajmiemy się odejmowaniem liczb binarnych. Skorzystamy tu z własności mówiącej, że odejmowanie liczb można zapisać w postaci dodawania, np. 4 – 3 = 4 + (–3).

Odejmowanie liczb w systemie dwójkowym

Kod uzupełnień do 2 (U2), s. 33–35

Liczby całkowite komputer pamięta w **kodzie uzupełnień do 2 (U2)**. Odejmowanie wykonywane jest więc poprzez dodawanie liczb zapisanych w kodzie uzupełnień do 2 (U2). Aby odjąć od siebie dwie liczby całkowite nieujemne zapisane w postaci binarnej, wystarczy znaleźć liczbę przeciwną do odjemnika zapisaną w kodzie U2, a następnie dodać ją do odjemnej. Należy pamiętać, że obydwie liczby muszą się składać z tej samej liczby bitów, a przeniesienie z najstarszego bitu pomijamy. Jako przykład rozpatrzymy odejmowanie dwóch liczb binarnych zapisanych na 8 bitach, zgodnie z poniższą specyfikacją.

Specyfikacja

Dane: `a`, `b` – napisy reprezentujące liczby całkowite nieujemne z zakresu od 0 do 127 zapisane w systemie binarnym.

Wynik: `c` – napis reprezentujący liczbę zapisaną w kodzie U2 będącą różnicą liczb binarnych `a` i `b`.

Zapis binarny liczb a i b w określonej liczbie bitów jest taki sam jak zapis tych liczb w kodzie U2 – dla liczb nieujemnych odpowiada on **naturalnemu zapisowi binarnemu**.

Naturalny zapis binarny, s. 33

Aby napisać program odejmujący liczby a i b, rozwiążemy dwa problemy:

1. znajdziemy liczbę przeciwną do liczby b zapisanej w kodzie U2 – aby wykonać to zadanie, zdefiniujemy funkcję `PrzeciwnaU2`;
2. dodamy do liczby a liczbę przeciwną do liczby b zapisanej w kodzie U2 – zdefiniujemy w tym celu funkcję `DodajU2`.

Kod źródłowy funkcji `main` wykorzystujący funkcje `PrzeciwnaU2` oraz `DodajU2` może wyglądać następująco:

Fragment kodu źródłowego programu odejmującego 8-bitowe liczby binarne – funkcja `main`

```
1.  int main()
2.  {
3.
4.      string a, b, c;
5.      cout<<"Liczba1: "; cin>>a;
6.      cout<<"Liczba2: "; cin>>b;
7.      while (a.size()<8) a='0'+a;
8.      while (b.size()<8) b='0'+b;
9.      b=PrzeciwnaU2(b);
10.     c=DodajU2(a,b);
11.     cout<<"Roznica = "<<c;
12.     return 0;
13. }
```

Aby bez pośrednictwa systemu dziesiętnego wyznaczyć liczbę przeciwną do liczby zapisanej na 8 bitach w kodzie U2, wystarczy skorzystać z podanego niżej algorytmu. Otrzymany wynik będzie reprezentacją liczby w kodzie U2.

Warto wiedzieć

Do liczby –128 zapisanej w kodzie U2 na 8 bitach nie ma liczby przeciwnej w kodzie U2 (wymagałoby to użycia 9 bitów), a liczbą przeciwną do 0 jest 0.

1. Zamień wszystkie bity liczby na przeciwne (0 na 1, a 1 na 0).
2. Do wyniku otrzymanego w kroku 1 dodaj liczbę 1.

Sprawdźmy działanie algorytmu dla liczby 01111000_{U2}, odpowiadającej liczbie dziesiętnej 120. Po zamianie bitów na przeciwne otrzymujemy 10000111, a po dodaniu liczby 1: 10001000_{U2}, czyli w systemie dziesiętnym –120. Aby dodać liczbę 1 w naszym przykładzie, wystarczy zamieniać – zaczynając od prawej strony – cyfry 1 na 0 do momentu, aż napotkamy pierwsze 0, które zamienimy na 1.

Ćwiczenie 4

Korzystając z powyższego algorytmu, znajdź liczbę przeciwną w kodzie U2 do liczby 11110100_{U2}. Podaj wartość dziesiętną obu liczb.

Kod funkcji `PrzeciwnaU2`, realizującej podany algorytm, może wyglądać następująco:

Fragment kodu źródłowego programu odejmującego 8-bitowe liczby binarne – funkcja znajdująca liczbę przeciwną do danej liczby (obie w kodzie U2)

```
1.  string PrzeciwnaU2(string s)
2.  {
3.      int i;
4.      for (i=0;i<8;i++)
5.          if (s[i]=='0') s[i]='1';
6.          else s[i]='0';
7.      s='0'+s; i=8;
8.      while (s[i]=='1')
9.      {
10.         s[i]='0';
11.         i--;
12.     }
13.     s[i]='1';
14.     return s.substr(1,8);
15. }
```

W liniach 4–6 wartości bitów zamieniane są na przeciwne (krok 1 algorytmu). Pętla w liniach 8–12 zamienia – zaczynając od prawej strony – cyfry 1 na 0 do momentu napotkania pierwszego 0, które jest zamieniane na 1 (linia 13). W linii 7 na początku napisu dołączane jest 0 pełniące funkcję **wartownika**, aby uniknąć sytuacji, w której w zapisie liczby będą same cyfry 1 (dotyczy to zapisu liczby 0, w którym po zamianie cyfr na przeciwne będą występować same cyfry 1). Wynikiem funkcji jest napis z pominięciem wartownika (linia 14).

Wartownik, s. 64

Dobra rada

Zamiast wprowadzać wartownika, możesz na początku funkcji sprawdzić, czy w zapisie liczby znajduje się co najmniej jedna cyfra 1 (np. korzystając z metody `find`).

Ćwiczenie 5

Napisz program wczytujący dwie liczby całkowite z zakresu od 0 do 127 zapisane w systemie binarnym i wypisujący wynik odejmowania tych liczb w kodzie U2. Zdefiniuj funkcję `DodajU2`, dodającą dwie liczby w kodzie U2.

Zapamiętaj

Aby odjąć dwie liczby całkowite nieujemne zapisane binarnie, należy znaleźć liczbę przeciwną do odjemnika zapisaną w kodzie U2, a następnie dodać ją do odjemnej.

Odejmowanie liczb w dowolnym systemie pozycyjnym

Zapiszemy teraz algorytm odejmowania dwóch liczb całkowitych nieujemnych w systemie o podanej podstawie. Zakładamy, że pierwsza z liczb jest większa od drugiej. Podstawę systemu ograniczymy do wartości z zakresu od 2 do 10, aby zapis algorytmu był czytelniejszy.

Oto specyfikacja problemu:

Specyfikacja

Dane: podstawa – podstawa systemu, $2 \leqslant$ podstawa $\leqslant 10$,
a, b – napisy reprezentujące liczby całkowite nieujemne w systemie pozycyjnym o podstawie podstawa, a > b.

Wynik: c – napis reprezentujący różnicę liczb a i b w systemie pozycyjnym o podstawie podstawa.

Pisemne odejmowanie liczb w dowolnym systemie pozycyjnym jest analogiczne do pisemnego odejmowania w systemie dziesiętnym. Jeśli cyfra odjemnej jest mniejsza od znajdującej się na tej samej pozycji cyfry odjemnika, to z poprzedniej pozycji pożyczamy wartość podstawy systemu. Na przykład dla liczb 3213_4 i 3031_4 odejmowanie będzie wyglądało tak jak na rysunku 6.2. Zera nieznaczące w wyniku pomijamy, więc $3213_4 - 3031_4 = 122_4$.

Warto wiedzieć

W programowaniu do podstawowych operacji arytmetycznych na liczbach całkowitych zalicza się: dodawanie, odejmowanie, mnożenie, wyznaczanie części całkowitej z dzielenia oraz wyznaczanie reszty z dzielenia.

```
  15
 3ZZ3
-3031
 0122
```

Rys. 6.2. Przykład odejmowania liczb w systemie czwórkowym

Algorytm zapiszemy w postaci listy kroków. Implementację algorytmu w języku C++ pomijamy ze względu na długość kodu.

1. Ustal wartość początkową wyniku c na napis pusty.
2. Ustal wartość początkową zmiennej pozyczka na **fałsz**.
3. Uzupełnij odjemnik zerami nieznaczącymi tak, aby liczba cyfr w odjemniku była taka sama jak liczba cyfr w odjemnej.
4. Dla każdej z cyfr odjemnej i odjemnika, traktowanych jako liczby jednocyfrowe, w kolejności od prawej do lewej wykonaj kroki 5-12.
   5. Jeśli zmienna pozyczka ma wartość **prawda**, wykonaj krok 6.
      6. Odejmij od cyfry odjemnej 1.
   7. Ustal wartość zmiennej pozyczka na **fałsz**.
   8. Oblicz różnicę cyfr odjemnej i odjemnika.
   9. Jeśli różnica jest ujemna, to wykonaj kroki 10-11.
      10. Ustal wartość zmiennej pozyczka na **prawda**.
      11. Do różnicy dodaj podstawę systemu.
   12. Dołącz cyfrę różnicy do wyniku.
13. Usuń z wyniku zera nieznaczące.

6.3. Mnożenie liczb w różnych systemach pozycyjnych

W pierwszym temacie wprowadzającym w zagadnienia dotyczące algorytmiki i programowania jednym z przykładów problemów algorytmicznych było **mnożenie sposobem pisemnym liczb** dziesiętnych. Odpowiednio zmodyfikowany algorytm można zastosować do mnożenia liczb zapisanych w systemach o podstawie innej niż 10.

Problem 2. Mnożenie liczb sposobem pisemnym, s. 11–12

Mnożenie liczb w systemie dwójkowym

Mnożenie liczb całkowitych nieujemnych zapisanych w systemie binarnym jest dość proste, ponieważ w systemie tym używa się tylko dwóch cyfr. Mnożenie przez 0 daje w wyniku 0, a przez 1 nie zmienia mnożonej liczby. Dlatego mnożąc górną liczbę przez kolejne cyfry liczby na dole, w iloczynach cząstkowych otrzymujemy albo zera, albo górną liczbę. Otrzymane iloczyny cząstkowe, od drugiego włącznie, można uzupełnić z prawej strony zerami – w przykładzie przedstawionym na rysunku 6.3 zera te są zaznaczone na czerwono.

Dobra rada

Podobnie jak w przypadku dodawania, aby ułatwić sobie mnożenie w dowolnym systemie pozycyjnym, możesz stworzyć tabelę mnożenia liczb jednocyfrowych w tym systemie.

```
   100
 · 101
 -----
   100
  0000
+ 10000
 -----
  10100
```

Rys. 6.3. Przykład mnożenia liczb w systemie dwójkowym

Oto specyfikacja problemu mnożenia liczb całkowitych nieujemnych zapisanych w systemie binarnym:

Specyfikacja

Dane: a, b – napisy reprezentujące liczby całkowite nieujemne w systemie binarnym.

Wynik: c – napis reprezentujący liczbę binarną będącą iloczynem liczb a i b.

Zapis algorytmu mnożącego liczby binarne zgodnie z podaną specyfikacją może wyglądać tak jak poniżej. Wykorzystujemy w nim funkcję `Dodaj`, dodającą dwie liczby binarne, realizującą algorytm zapisany w pseudokodzie na s. 87.

```
c ← "0"
dla i ← długość b - 1, ..., 0 wykonuj
    jeśli b[i]='1' to c ← Dodaj(c,a)
    a ← a + '0'
```

Kod źródłowy funkcji `main` programu mnożącego liczby całkowite nieujemne zapisane w systemie binarnym może wyglądać następująco.

```
1.  int main()
2.  {
3.      string a, b, c="0";
4.      int i, j;
5.      cout<<"Liczba1: "; cin>>a;
6.      cout<<"Liczba2: "; cin>>b;
7.      for (i=b.size()-1;i>=0;i--)
8.      {
9.          if (b[i]=='1') c=Dodaj(c,a);
10.         a=a+'0';
11.     }
12.     cout<<"Iloczyn = "<<c;
13.     return 0;
14. }
```

Fragment kodu źródłowego programu mnożącego dwie liczby binarne – funkcja `main`

Ćwiczenie 6

Napisz program wczytujący dwie liczby całkowite nieujemne zapisane w systemie dwójkowym, reprezentowane jako napisy, a następnie wypisujący iloczyn tych liczb w systemie dwójkowym. Zdefiniuj i wykorzystaj funkcję `Dodaj`, dodającą dwie liczby binarne.

Przedstawiony algorytm mnożenia liczb binarnych można zastosować dla liczb zapisanych w kodzie U2, a więc także ujemnych. W końcowym wyniku należy jedynie pominąć wszystkie bity przekraczające liczbę bitów, których używamy do zapisu liczby w kodzie U2. Na przykład dla liczb –2 i –3 zapisanych na czterech bitach ($-2 = 1110_{U2}$, $-3 = 1101_{U2}$) otrzymujemy liczbę 6 (0110_{U2}).

```
    1110
  · 1101
  ------
    1110
   00000
  111000
+ 1110000
  ~~1011~~0110
```

Rys. 6.4. Przykład mnożenia liczb 4-bitowych zapisanych w kodzie U2

Mnożenie liczb w dowolnym systemie pozycyjnym

Zapiszmy specyfikację problemu mnożenia liczb całkowitych nieujemnych w systemie o podanej podstawie oraz algorytm rozwiązujący ten problem. Podstawę systemu ograniczymy do wartości z zakresu od 2 do 10, aby kod programu, który utworzymy, był czytelniejszy. Zapis algorytmu w pseudokodzie jest poprawny także dla większych wartości podstawy.

Specyfikacja

Dane: `podst` – podstawa systemu, $2 \leqslant$ `podst` $\leqslant 10$,
`a`, `b` – napisy reprezentujące liczby całkowite nieujemne w systemie pozycyjnym o podstawie `podst`.

Wynik: `c` – napis reprezentujący iloczyn liczb `a` i `b` w systemie pozycyjnym o podstawie `podst`.

```
c ← "0"
d ← długość liczby b - 1
dla i ← d, d - 1, ..., 1, 0 wykonuj
    pom ← MnozPrzezCfr(b[i],a,podst)
    dla j ← 1, 2, ..., d - i wykonuj pom ← pom + '0'
    c ← Dodaj(c,pom,podst)
```

W powyższym algorytmie założyliśmy, że umiemy wykonać dwie operacje: mnożenie liczby przez liczbę jednocyfrową oraz dodawanie dwóch liczb. Rozwiązania obu tych problemów zapiszemy w postaci funkcji. Najpierw zdefiniujemy funkcję `MnozPrzezCfr`, mnożącą liczbę przez liczbę jednocyfrową. Parametrami funkcji będą: `cfr` – liczba jednocyfrowa, `liczba` – liczba, którą mnożymy przez `cfr`, `podst` – podstawa systemu. W pseudokodzie zapis algorytmu, który realizuje ta funkcja, może wyglądać następująco:

```
funkcja MnozPrzezCfr(cfr,liczba,podst)
    przn ← 0
    wynik ← ""
    dla i ← długość liczba - 1, ..., 0 wykonuj
        iloczyn ← cfr * liczba[i] + przn
        wynik ← cyfra(iloczyn mod podst) + wynik
        przn ← iloczyn div podst
    jeśli przn > 0 to wynik ← cyfra(przn) + wynik
    zwróć wynik i zakończ
```

Algorytm mnożenia liczby przez liczbę jednocyfrową jest analogiczny do algorytmu dodającego dwie liczby, zapisanego w pseudokodzie na s. 87. Dodawanie zastąpiliśmy iloczynem. W przypadku dodawania ewentualne przeniesienie może być równe tylko 1, przy mnożeniu może być większe, ale jest reprezentowane przez liczbę jednocyfrową w danym systemie.

Oto kod funkcji mnożącej liczbę reprezentowaną jako napis przez liczbę jednocyfrową w danym systemie pozycyjnym:

Fragment kodu źródłowego programu mnożącego dwie liczby w systemie pozycyjnym o podstawie z zakresu od 2 do 10 – definicja funkcji umożliwiającej mnożenie przez liczbę jednocyfrową

```
1.  string MnozPrzezCfr(int cfr, string liczba, int podst)
2.  {
3.      int i, przn=0, iloczyn;
4.      string wynik="";
5.      for (i=liczba.size()-1;i>=0;i--)
6.      {
7.          iloczyn=cfr*(liczba[i]-'0')+przn;
8.          przn=iloczyn/podst;
9.          wynik=char(iloczyn%podst+'0')+wynik;
10.     }
11.     if (przn>0) wynik=char(przn+'0')+wynik;
12.     return wynik;
13. }
```

Kod źródłowy funkcji `main` algorytmu realizującego mnożenie dwóch liczb, wykorzystujący funkcje `MnozPrzezCfr` oraz `Dodaj`, może wyglądać następująco:

```
1.  int main()
2.  {
3.      int i, j, d, podstawa;
4.      string a, b, c="0", pom;
5.      cout<<"Podstawa systemu: "; cin>>podstawa;
6.      cout<<"Liczba1: "; cin>>a;
7.      cout<<"Liczba2: "; cin>>b;
8.      d=b.size()-1;
9.      for (i=d;i>=0;i--)
10.     {
11.         pom=MnozPrzezCfr(b[i]-'0',a,podstawa);
12.         for (j=1;j<=d-i;j++) pom=pom+'0';
13.         c=Dodaj(c,pom,podstawa);
14.     }
15.     cout<<"Iloczyn = "<<c;
16.     return 0;
17. }
```

Fragment kodu źródłowego programu mnożącego dwie liczby w systemie pozycyjnym o podstawie z zakresu od 2 do 10 – funkcja `main`

Zwróć uwagę na nagłówek definicji funkcji `MnozPrzezCfr`:

```
string MnozPrzezCfr(int cfr, string liczba, int podst)
```

oraz wywołanie funkcji `MnozPrzezCfr` w funkcji `main`:

```
MnozPrzezCfr(b[i]-'0',a,podstawa)
```

Przy wywołaniu funkcji `MnozPrzezCfr` w miejsce **parametru formalnego** `cfr` typu całkowitego podstawiany jest **parametr aktualny** – wartość `i`-tej cyfry liczby `b`: kod ASCII cyfry znajdującej się na `i`-tej pozycji minus kod ASCII znaku `'0'` (`b[i]-'0'`).

Parametr formalny, parametr aktualny, s. 65

A to ciekawe

Gdzie komputer wykonuje obliczenia?

Mózgiem komputera jest procesor. Za wykonywanie podstawowych działań arytmetycznych oraz logicznych odpowiada jedna z jego głównych części: ALU, czyli jednostka arytmetyczno-logiczna (ang. *arithmetic logic unit*). ALU można zaprojektować tak, aby wykonywała dowolne operacje, jednak im większe będzie mieć możliwości, tym wyższa będzie cena procesora. Architekturę komputera uwzględniającą jednostkę arytmetyczno-logiczną zaproponowano w 1945 r. Od nazwiska jednego z jej twórców nazywa się ją architekturą von Neumanna.

Ćwiczenie 7

Napisz program wykonujący mnożenie dwóch liczb w systemie pozycyjnym o podstawie p z zakresu od 2 do 10. Zdefiniuj i wykorzystaj dwie funkcje: dodającą dwie liczby oraz mnożącą liczbę przez liczbę jednocyfrową.

6.4. Dzielenie liczb w różnych systemach pozycyjnych

Rozważymy teraz problem dzielenia dwóch liczb w systemie pozycyjnym o wybranej podstawie. Ponieważ operujemy liczbami całkowitymi dodatnimi, będziemy znajdować część całkowitą oraz resztę z dzielenia.

Dzielenie liczb w systemie dwójkowym

Wykonamy teraz dzielenie pisemne przykładowych liczb binarnych: $1111_2 : 101_2$ i $100010_2 : 100_2$ (rys. 6.5). Zwróć uwagę, że w przypadku systemu binarnego wynikiem cząstkowego dzielenia może być tylko 0 lub 1, więc tak naprawdę zamiast dzielenia wykonujemy odejmowanie. Ostatnia otrzymana różnica jest resztą z dzielenia.

```
   11                 110
 1111:101           100010:101
-101                -101
  101                 111
 -101                -101
    0                  100
```

Rys. 6.5. Dzielenie przykładowych liczb binarnych

Zapiszmy algorytm znajdowania części całkowitej i reszty z dzielenia dwóch liczb binarnych. Oto specyfikacja problemu:

Specyfikacja

Dane: a, b – napisy reprezentujące liczby całkowite dodatnie zapisane w systemie binarnym, a > b.

Wynik: `ilorazc` – napis reprezentujący liczbę binarną będącą częścią całkowitą z dzielenia a przez b,
`reszta` – napis reprezentujący liczbę binarną będącą resztą z dzielenia a przez b.

Ponieważ zakładamy, że dzielna jest większa od dzielnika, wartością początkową wyniku dzielenia (pierwszą cyfrą wyniku) będzie 1. Aktualny wynik dzielenia będziemy przechowywać w zmiennej `ilorazc`. Następnie z lewej strony dzielnej a należy wybrać cyfry tworzące najmniejszą liczbę nie mniejszą od dzielnika b. Niech będzie to wartość

początkowa reszty z dzielenia (`reszta`). Następnie odejmujemy od reszty dzielnik. Dla kolejnych cyfr dzielnej:

1. dopisujemy cyfrę do reszty,
2. jeśli reszta nie jest mniejsza od dzielnika, to do wyniku (`ilorazc`) dopisujemy 1 i odejmujemy od reszty dzielnik. W przeciwnym przypadku do wyniku dopisujemy 0.

Zapis algorytmu w pseudokodzie może wyglądać następująco:

```
ilorazc ← "1"
d ← długość b
// zapisanie w zmiennej reszta d pierwszych znaków z napisu a
reszta ← a[0..d-1]
jeśli reszta < b to
    reszta ← reszta + a[d]
    d ← d + 1
reszta ← Odejmij(reszta,b)
dla i ← d, d + 1, ..., długość a - 1 wykonuj
    reszta ← reszta + a[i]
    jeśli reszta < b to ilorazc ← ilorazc + '0'
    w przeciwnym przypadku
        ilorazc ← ilorazc + '1'
        reszta ← Odejmij(reszta,b)
```

W algorytmie odwołujemy się do problemu odejmowania dwóch liczb binarnych omówionego w tym temacie w zagadnieniu 6.2. Implementację algorytmu dzielącego dwie liczby binarne w języku C++ pomijamy ze względu na długość kodu.

Warto wiedzieć

Niektóre kalkulatory systemowe (np. kalkulator w systemie Windows po ustawieniu widoku programisty) umożliwiają wykonywanie działań w systemach o podstawach 2, 8, 10 i 16.

Dzielenie liczb w dowolnym systemie pozycyjnym

Dzielenie jest zwykle najbardziej pracochłonnym działaniem arytmetycznym zarówno dla człowieka, jak i dla komputera. Tutaj ograniczymy się do pokazania przykładu dzielenia liczb w systemie o podstawie innej niż 2 i 10.

Zasady dzielenia pisemnego są takie same niezależnie od podstawy systemu. Rysunek 6.6 pokazuje dzielenie liczb w systemie o podstawie 4: 3213_4 przez 113_4. Liczba 113_4 mieści się dwa razy w 321_4, a także dwa razy w 233_4.

```
    22
  3213 : 113
 -232
   233
  -232
     1   reszta z dzielenia
```

Rys. 6.6. Dzielenie przykładowych liczb w systemie czwórkowym

Zapamiętaj

Pisemne działania arytmetyczne w systemach pozycyjnych o podstawie innej niż 10 wykonujemy podobnie jak w systemie dziesiętnym. Trzeba tylko pamiętać, że wyznaczając przeniesienie oraz cyfrę wyniku, należy obliczać część całkowitą i resztę z dzielenia przez podstawę systemu. Pożyczana wartość to podstawa systemu.

Podsumowanie

- Działania arytmetyczne w systemach pozycyjnych o podstawach innych niż dziesięć wykonujemy podobnie jak w systemie dziesiętnym.
- Gdy reprezentujemy liczby jako napisy, nie jesteśmy ograniczeni zakresem liczb – możemy wykonywać działania arytmetyczne na dowolnie dużych liczbach.
- Wykonując operacje arytmetyczne na liczbach w systemie o podstawie większej niż 10 reprezentowanych jako napisy, należy pamiętać, że inaczej dokonujemy konwersji liczb jednocyfrowych większych od 9, a inaczej – mniejszych lub równych 9.
- Kiedy dodajemy dwie liczby, wygodnie jest uzupełnić krótszą z nich zerami nieznaczącymi, aby obydwie liczby składały się z tej samej liczby cyfr.
- Żeby odjąć liczbę binarną, wystarczy dodać liczbę przeciwną do tej liczby zapisaną w kodzie U2.
- W dzieleniu pisemnym liczb binarnych można wykorzystać odejmowanie.

Zadania

* **1** Dodaj pisemnie liczby 6543_7 i 432_7.

* **2** Pomnóż pisemnie liczby 6543_7 i 432_7.

* **3** Oblicz część całkowitą oraz resztę z dzielenia liczby 10000000_2 przez 11_2.

* **4** Napisz program dodający do siebie trzy liczby reprezentowane jako napisy, zapisane w systemie o podstawie z zakresu od 2 do 10. Program powinien zwrócić napis reprezentujący wynik dodawania zapisany w systemie o tej samej podstawie.

** **5** Adresy komórek pamięci często wyrażamy w systemie szesnastkowym. Chcemy wyznaczyć adres przesunięty o określoną liczbę komórek (wyrażoną w systemie dziesiętnym). Napisz program, który wczyta napis reprezentujący liczbę szesnastkową oraz liczbę całkowitą dodatnią w systemie dziesiętnym i wyznaczy wartość szesnastkową sumy wczytanych wartości.

** **6** Napisz program, który doda do siebie dwie liczby binarne zapisane w kodzie uzupełnień do 2 (U2) na 16 bitach. Program powinien sygnalizować przekroczenie zakresu.

** **7** Napisz program implementujący algorytm pisemnego odejmowania liczb w systemie o podstawie z zakresu od 2 do 10 omówiony w tym temacie.

** **8** Omówiony w tym temacie program mnożący dwie liczby zapisane w systemie o podstawie p zmodyfikuj tak, aby podstawa systemu mogła być większa od 10.

** **9** Napisz program implementujący algorytm pisemnego dzielenia liczb binarnych omówiony w tym temacie.

** **10** Napisz program dzielący liczbę reprezentowaną jako napis w systemie o podstawie z zakresu od 2 do 10 przez liczbę jednocyfrową zapisaną w tym samym systemie.

*** **11** Poszukaj informacji na temat zapisu liczb w zrównoważonym systemie trójkowym. Następnie napisz program dodający dwie liczby w tym systemie.

*** **12** Napisz program dodający dwie liczby w silniowym systemie pozycyjnym.

Uwaga: W silniowym systemie pozycyjnym mnożniki poszczególnych pozycji są zdefiniowane nie przez potęgę podstawy systemu, ale przez silnię kolejnych liczb naturalnych. Na przykład $2011_! = 2 \cdot 4! + 0 \cdot 3! + 1 \cdot 2! + 1 \cdot 1! = 2 \cdot 24 + 0 + 2 + 1 = 51$. Zapis w silniowym systemie pozycyjnym jest jednoznaczny, tzn. każdą liczbę naturalną można zapisać tylko w jeden sposób i każdy zapis oddaje dokładnie jedną wartość dziesiętną.

7. Algorytm Euklidesa i działania na ułamkach

Algorytm Euklidesa, czyli metoda znajdowania największego wspólnego dzielnika dwóch liczb, jest uważany za najstarszy znany algorytm. Znalazł on zastosowanie w rozwiązywaniu różnych problemów, m.in. w kryptografii. W tym temacie poznasz różne wersje algorytmu Euklidesa oraz użyjesz go do wykonywania działań na liczbach wymiernych.

Cele lekcji

- Poznasz interpretację geometryczną algorytmu Euklidesa z wykorzystaniem odejmowania.
- Zapiszesz iteracyjnie algorytm Euklidesa w dwóch wersjach: z wykorzystaniem odejmowania i dzielenia.
- Dowiesz się, czym jest struktura w języku C++.
- Zastosujesz strukturę do reprezentacji liczb wymiernych.
- Wykorzystasz algorytm Euklidesa w działaniach na liczbach wymiernych.
- Nauczysz się używać funkcji typu `void`.
- Dowiesz się, czym są zmienne lokalne i globalne.
- Zrozumiesz, na czym polega przekazanie parametrów przez wartość.

Największy wspólny dzielnik (NWD) dwóch liczb całkowitych dodatnich to największa liczba całkowita, która dzieli obie te liczby bez reszty.

Największy wspólny dzielnik (NWD) dwóch liczb całkowitych dodatnich znajdowaliście już w szkole podstawowej na lekcjach matematyki, np. skracając ułamki. Aby wyznaczyć NWD, wystarczy rozłożyć obie liczby na czynniki pierwsze i obliczyć iloczyn czynników powtarzających się w obu rozkładach. Jest to jednak metoda mało wygodna do implementacji, a w przypadku dużych liczb – czasochłonna. Dlatego warto skorzystać z algorytmu opisanego przez Euklidesa ok. 300 r. p.n.e.

7.1. Algorytm Euklidesa z wykorzystaniem odejmowania

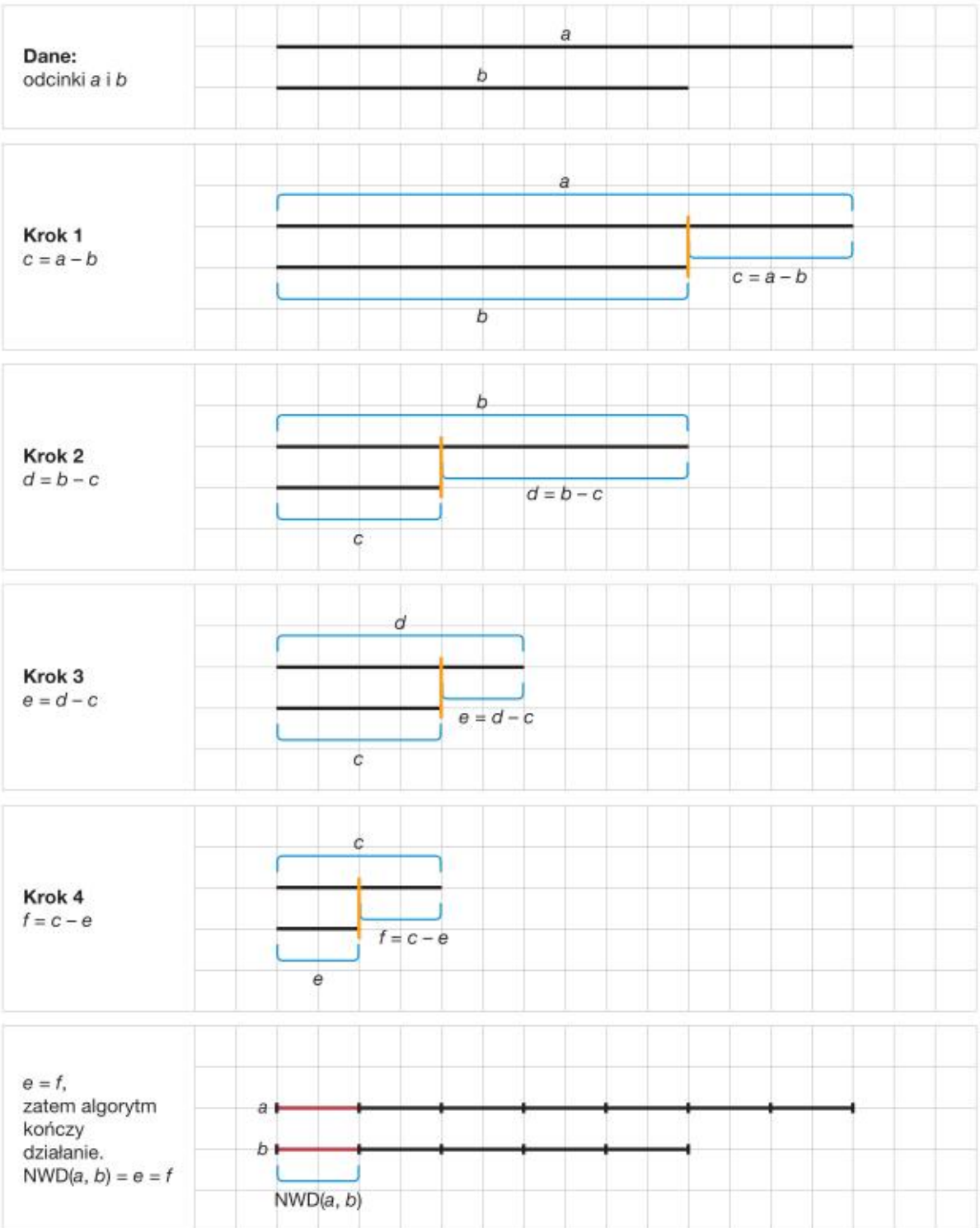
Algorytm Euklidesa

Euklides swój algorytm, zwany obecnie **algorytmem Euklidesa**, opisał pierwotnie dla odcinków, a nie dla liczb. Znajdował najdłuższy odcinek jednostkowy, który wyznaczał miarę dwóch odcinków, czyli odkładał się w każdym z odcinków całkowitą liczbę razy.

Interpretacja geometryczna algorytmu Euklidesa

Załóżmy, że mamy dwa odcinki a i b różnej długości. Najpierw od dłuższego odcinka odejmujemy krótszy. Następnie wykonujemy podobne odejmowanie, przy czym bierzemy pod uwagę krótszy z odcinków z poprzedniego kroku oraz różnicę odcinków z poprzedniego kroku. Odejmowanie odpowiednich odcinków powtarzamy tak długo, aż otrzymamy dwa odcinki tej samej długości.

Rysunek 7.1 przedstawia przykładową ilustrację działania algorytmu. Zauważ, że jeśli wyjściowe odcinki byłyby tej samej długości, nie musielibyśmy nic robić – poszukiwany odcinek miałby taką samą długość jak każdy z danych odcinków.



Rys. 7.1. Graficzne przedstawienie algorytmu Euklidesa z wykorzystaniem odejmowania

Algorytm Euklidesa, który znajduje największy wspólny dzielnik z wykorzystaniem odejmowania, opiera się na następującej własności: NWD dwóch liczb całkowitych dodatnich jest taki sam jak NWD mniejszej z tych liczb oraz różnicy większej i mniejszej liczby. Na przykład dla dwóch liczb całkowitch dodatnich a i b, gdzie $a > b$:

$$\mathrm{NWD}(a, b) = \mathrm{NWD}(a - b, b)$$

Można łatwo wykazać, że podana zależność jest prawdziwa.

Jeśli $k = \mathrm{NWD}(a, b)$, to $a = n \cdot k$ i $b = m \cdot k$, gdzie n i m to liczby całkowite dodatnie. Wówczas:
$a - b = n \cdot k - m \cdot k = (n - m) \cdot k$, czyli k jest dzielnikiem różnicy $a - b$.

Wiemy już, że k jest dzielnikiem liczb $a - b$ i b. Jednak nie wykazaliśmy jeszcze, że jest największym wspólnym dzielnikiem tych liczb. Przypuśćmy, że istnieje liczba s, taka że $s = \mathrm{NWD}(a - b, b)$. Liczba s musi dzielić się przez k. Ponieważ jest dzielnikiem liczb $a - b$ i b, jest także dzielnikiem $(a - b) + b = a$. Zatem s jest dzielnikiem k. Stąd $s = k$, czyli $k = \mathrm{NWD}(a - b, b)$.

Tabela 7.1 ilustruje działanie algorytmu Euklidesa z wykorzystaniem odejmowania dla liczb 14 i 10 (liczby te odpowiadają długościom odcinków z przykładu na rysunku 7.1 ze s. 103).

Krok algorytmu	Liczba a	Liczba b
–	14	10
1	14 – 10 = 4	10
2	4	10 – 4 = 6
3	4	6 – 4 = 2
4	4 – 2 = 2	2

Tabela 7.1. Działanie algorytmu Euklidesa z wykorzystaniem odejmowania na przykładzie liczb 14 i 10

Oto specyfikacja problemu oraz algorytm Euklidesa z wykorzystaniem odejmowania zapisany w pseudokodzie:

Specyfikacja

Dane: a, b – liczby całkowite, $0 < a \leqslant 2^{31} - 1$, $0 < b \leqslant 2^{31} - 1$.

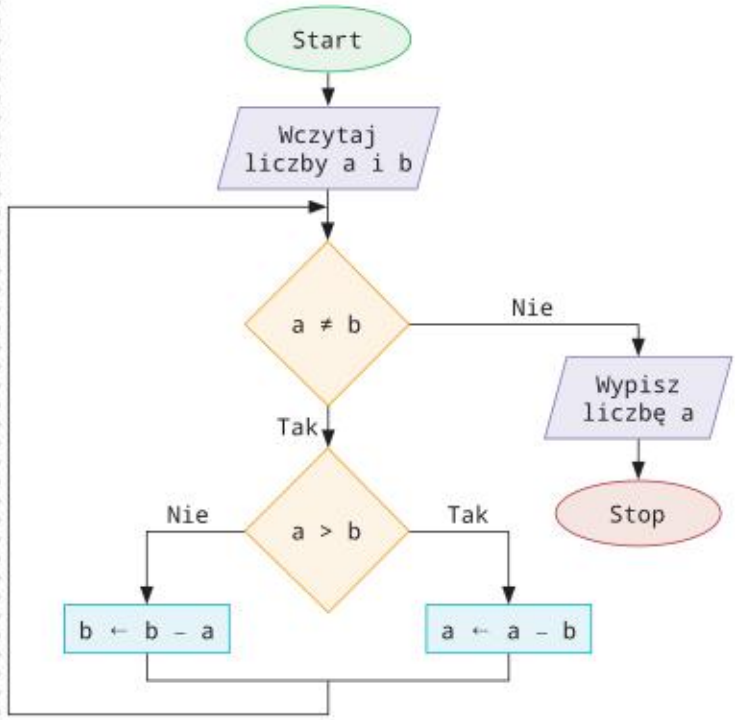
Wynik: $\mathrm{NWD}(a, b)$

```
dopóki a ≠ b wykonuj
    jeśli a > b to a ← a - b
    w przeciwnym przypadku b ← b - a
wypisz a
```

Zwróć uwagę, że w ostatniej linii pseudokodu zamiast wartości zmiennej a moglibyśmy wypisać wartość zmiennej b – po wykonaniu instrukcji w ostatnim kroku pętli wartości tych zmiennych będą równe.

Jeśli chcielibyśmy przedstawić algorytm poszukiwania NWD dwóch liczb całkowitych dodatnich a i b w postaci **schematu blokowego**, wyglądałby on np. tak jak poniżej.

Schemat blokowy, s. 15



Rys. 7.2. Schemat blokowy algorytmu Euklidesa z wykorzystaniem odejmowania

Ćwiczenie 1

Napisz program wyznaczający największy wspólny dzielnik dwóch wczytanych liczb zgodnie z podanymi wcześniej specyfikacją i algorytmem.

A to ciekawe

Matematyczny bestseller wszechczasów

Elementy Euklidesa to traktat matematyczny, który opisuje osiągnięcia starożytnej geometrii. Jest on uznawany za dzieło wyjątkowe, ponieważ zastosowano w nim podejście dedukcyjne (od ogółu do szczegółu), precyzję definicji i dowodów twierdzeń. W traktacie znalazł się również algorytm, który jest wykorzystywany we współczesnej informatyce. Mimo że dzieło Euklidesa powstało ok. 300 r. p.n.e., używano go w kolejnych przekładach jako podręcznika do geometrii przez niemal 2000 lat. Uznaje się, że *Elementy* są drugą po Biblii książką najczęściej drukowaną i tłumaczoną w historii świata.

7.2. Algorytm Euklidesa z wykorzystaniem dzielenia

Zwróć uwagę, że jeśli w algorytmie Euklidesa wykorzystującym odejmowanie jedna liczba jest dużo większa od drugiej, to od większej liczby trzeba wiele razy odjąć tę samą (mniejszą) liczbę. Taki przypadek ilustruje tabela 7.2.

Iteracja pętli	Liczba *a*	Liczba *b*
–	196	42
1	196 – 42 = 154	42
2	154 – 42 = 112	42
3	112 – 42 = 70	42
4	70 – 42 = 28	42
5	28	42 – 28 = 14
6	28 – 14 = 14	14

Tabela 7.2. Działanie algorytmu Euklidesa z wykorzystaniem odejmowania na przykładzie liczb 196 i 42

Wielokrotne odejmowanie możemy zastąpić dzieleniem, a dokładniej resztą z dzielenia większej liczby przez mniejszą. Taki sposób postępowania uwzględnia druga wersja algorytmu Euklidesa. Tabela 7.3 ilustruje działanie tego algorytmu dla tych samych liczb co tabela 7.2.

Iteracja pętli	Liczba *a*	Liczba *b*
–	196	42
1	196 **mod** 42 = 28	42
2	28	42 **mod** 28 = 14
3	28 **mod** 14 = 0	14

Tabela 7.3. Działanie algorytmu Euklidesa z wykorzystaniem dzielenia na przykładzie liczb 196 i 42

Warto wiedzieć

Algorytm Euklidesa z wykorzystaniem dzielenia oblicza NWD także dla liczb całkowitych ujemnych. Algorytm z wykorzystaniem odejmowania działa prawidłowo tylko dla liczb całkowitych dodatnich.

Wynikiem dzielenia 196 przez 42 jest 4 i reszta 28. Część całkowita określa liczbę operacji odejmowania, a reszta z dzielenia – wynik ostatniego odejmowania. W tym przypadku cztery działania zastąpiliśmy jednym.

Zwróć uwagę na ostatni wiersz tabeli 7.3. Warunkiem zakończenia działania algorytmu nie jest tym razem otrzymanie dwóch takich samych wartości. Algorytm zakończy działanie, jeśli jedna ze zmiennych przyjmie wartość 0 (przed ostatnim dzieleniem większa liczba jest wielokrotnością mniejszej, a więc ta mniejsza jest największym wspólnym dzielnikiem). Oto zapis algorytmu w pseudokodzie:

```
dopóki a ≠ 0 oraz b ≠ 0 wykonuj
    jeśli a > b to a ← a mod b
    w przeciwnym przypadku b ← b mod a
jeśli a ≠ 0 to wypisz a
w przeciwnym przypadku wypisz b
```

Podany pseudokod można uprościć. Warunek powtarzania pętli zamienimy na warunek `a * b ≠ 0` (ponieważ jeśli `a ≠ 0` **oraz** `b ≠ 0`, to `a * b ≠ 0`). Ponadto instrukcję warunkową:

```
jeśli a ≠ 0 to wypisz a
w przeciwnym przypadku wypisz b
```

można zastąpić wypisaniem sumy wartości zmiennych, ponieważ jedna ze zmiennych ma wartość zero. Oto zapis algorytmu po modyfikacji:

```
dopóki a * b ≠ 0 wykonuj
    jeśli a > b to a ← a mod b
    w przeciwnym przypadku b ← b mod a
wypisz a + b
```

Ćwiczenie 2

Napisz program wyznaczający największy wspólny dzielnik dwóch liczb z wykorzystaniem dzielenia.

Jeśli założymy, że wartość zmiennej `a` jest zawsze większa od wartości zmiennej `b`, algorytm można zapisać prościej, bez użycia instrukcji warunkowej. Zmienna `pom` w poniższym pseudokodzie jest zmienną pomocniczą, służącą do tymczasowego przechowania mniejszej wartości. Po wykonaniu dzielenia wartość tę przyjmuje zmienna `a`.

```
dopóki b ≠ 0 wykonuj
    pom ← b
    b ← a mod b
    a ← pom
wypisz a
```

Jeśli na początku wartość zmiennej `a` będzie mniejsza od wartości zmiennej `b`, to pierwsza iteracja pętli zamieni wartości zmiennych `a` i `b`. Działanie algorytmu w takim przypadku ilustruje tabela 7.4.

Warto wiedzieć

Algorytm Euklidesa z wykorzystaniem dzielenia wykonuje nie więcej operacji dzielenia niż liczba cyfr mniejszej liczby pomnożona przez 5. Wykazał to francuski matematyk i inżynier Gabriel Lamé. Najwięcej operacji jest wykonywanych dla liczb, które są dwiema kolejnymi liczbami Fibonacciego. Liczby te omówimy w temacie 17.

Iteracja pętli	Wartość zmiennej a	Wartość zmiennej b	Wartość zmiennej pom
1	42	196	–
2	196	42 **mod** 196 = 42	196
3	42	196 **mod** 42 = 28	42
4	28	42 **mod** 28 = 14	28
5	14	28 **mod** 14 = 0	14

Tabela 7.4. Działanie zmodyfikowanego algorytmu Euklidesa z wykorzystaniem dzielenia na przykładzie liczb 42 i 196

Ćwiczenie 3

Przedstaw w postaci schematu blokowego działanie algorytmu Euklidesa z wykorzystaniem dzielenia w wersji ze zmienną pomocniczą.

Poniżej znajduje się kod źródłowy programu wyznaczającego największy wspólny dzielnik dwóch liczb całkowitych dodatnich z wykorzystaniem dzielenia. Zwróć uwagę na warunek pętli w linii 8: zapis `b!=0` oznacza „wartość zmiennej `b` jest różna od zera".

Kod źródłowy programu znajdującego NWD dwóch liczb całkowitych dodatnich z wykorzystaniem dzielenia

```
1.  #include <iostream>
2.
3.  using namespace std;
4.
5.  int NWD(int a, int b)
6.  {
7.      int pom;
8.      while (b!=0)
9.      {
10.         pom=b;
11.         b=a%b;
12.         a=pom;
13.     }
14.     return a;
15. }
16.
17. int main()
18. {
19.     int a, b;
20.     cout<<"a = "; cin>>a;
21.     cout<<"b = "; cin>>b;
22.     cout<<"NWD("<<a<<","<<b<<") = "<<NWD(a,b);
23.     return 0;
24. }
```

Zmienna lokalna

Zmienna globalna

Do obliczeń funkcja `NWD` wykorzystuje zmienną pomocniczą zadeklarowaną w obrębie funkcji. Takie zmienne nazywamy **zmiennymi lokalnymi**. Zmienne deklarowane poza definicją funkcji i dostępne dla różnych funkcji nazywamy **zmiennymi globalnymi**.

Przekazanie parametru przez wartość

Istnieją różne sposoby przekazania parametrów do funkcji. Ten, który stosowaliśmy w dotychczasowych przykładach, nazywa się **przekazaniem parametru przez wartość**. Parametry są w nim traktowane jak zmienne lokalne, czyli deklarowane w funkcji. Podczas wywołania funkcji tworzona jest kopia parametru i nadawana jest mu wartość początkowa **parametru aktualnego**. Oznacza to, że wartości parametrów są chronione, a parametrem aktualnym może być nie tylko wartość zmiennej, lecz także stała lub wartość wyrażenia. Nawet jeśli zmieniamy wartość parametru podczas działania funkcji (w funkcji `NWD` zmieniają się wartości parametrów `a` i `b`), to te zmiany są lokalne. Po zakończeniu wykonywania funkcji parametry `a` i `b` mają wartości takie same jak przed wywołaniem funkcji.

Parametr aktualny, s. 65

Zapamiętaj

Algorytm Euklidesa pozwala na znalezienie największego wspólnego dzielnika dwóch liczb całkowitych dodatnich. Odejmowanie od większej liczby mniejszej można zastąpić resztą z dzielenia większej liczby przez mniejszą.

7.3. Reprezentacja liczb wymiernych

W pamięci komputera **liczby wymierne** nie mają żadnej specjalnej reprezentacji – podobnie jest w językach programowania. Dlatego liczby wymierne należy przedstawiać jako pary liczb całkowitych. W języku C++ można to zrobić na kilka sposobów.

Liczba wymierna to liczba, którą można przedstawić w postaci ilorazu dwóch liczb całkowitych, czyli ułamka zwykłego.

Jednym z nich jest zapisanie licznika i mianownika jako powiązanych ze sobą zmiennych typu całkowitego. Dzięki temu będzie wiadomo, że reprezentują daną liczbę wymierną, a nie dwie niezależne liczby całkowite. Można w tym celu wykorzystać **strukturę**. Struktura jest złożonym typem danych, który pozwala zgrupować wiele zmiennych różnych typów pod jedną nazwą. Jednocześnie każda ze zmiennych ma także własną nazwę. Oto przykład definicji struktury o nazwie `wymierna`, grupującej dwie zmienne typu całkowitego o nazwach `licz` i `mian` (od słów „licznik" i „mianownik").

Struktura

```
struct wymierna
{
    int licz;
    int mian;
};
```

Dobra rada

Przy definiowaniu struktury pamiętaj o dodaniu średnika po nawiasie zamykającym. Po ciągu instrukcji ujętych w nawiasy klamrowe, np. w definicjach funkcji lub pętlach, nie trzeba tego robić.

Po zdefiniowaniu w programie typu danych o nazwie `wymierna` będziemy mogli deklarować zmienne tego typu. Poniżej znajduje się przykład deklaracji zmiennych `a` i `b` typu `wymierna`, czyli ułamków zwykłych `a` i `b`.

```
wymierna a, b;
```

Aby odwołać się do wartości przechowywanych w strukturze, korzystamy z notacji z kropką, np. `a.licz`, co w naszym przypadku oznacza licznik ułamka `a`.

Problemem, którym się teraz zajmiemy, będzie skrócenie ułamka właściwego tak, aby w liczniku i mianowniku otrzymać **liczby względnie pierwsze**. Oto specyfikacja problemu:

Liczby względnie pierwsze to liczby całkowite, których największym wspólnym dzielnikiem jest liczba 1.

Specyfikacja

Dane: a – liczba wymierna postaci $\frac{l}{m}$, gdzie l i m to liczby całkowite dodatnie, $l < m$.

Wynik: skrócony ułamek a, $a = \frac{p}{q}$, gdzie p i q to liczby względnie pierwsze.

Żeby skrócić ułamek zwykły, należy obliczyć największy wspólny dzielnik licznika i mianownika, a następnie podzielić licznik i mianownik przez ten dzielnik. Poniżej znajduje się fragment kodu źródłowego programu wykorzystującego funkcję NWD. Liczbę wymierną zdefiniowano za pomocą struktury.

Fragment kodu źródłowego programu skracającego ułamek właściwy – definicja struktury `wymierna` i funkcja `main`

```
1.  struct wymierna
2.  {
3.      int licz;
4.      int mian;
5.  };
6.
7.  int main()
8.  {
9.      wymierna a;
10.     int n;
11.     cout<<"Licznik = "; cin>>a.licz;
12.     cout<<"Mianownik = "; cin>>a.mian;
13.     n=NWD(a.licz,a.mian);
14.     a.licz=a.licz/n;
15.     a.mian=a.mian/n;
16.     cout<<"Ulamek po skroceniu: ";
17.     cout<<a.licz<<"/"<<a.mian;
18.     return 0;
19. }
```

Ćwiczenie 4

Zdefiniuj funkcję `Skroc` typu `wymierna`, skracającą ułamek właściwy podany jako parametr. Nagłówek funkcji powinien wyglądać następująco: `wymierna Skroc(wymierna a)`.
Następnie napisz program skracający ułamek właściwy z wykorzystaniem zdefiniowanej funkcji.

7.4. Dodawanie i mnożenie liczb wymiernych

W szkole podstawowej wykonywaliście podstawowe działania arytmetyczne na ułamkach zwykłych. Tu skupimy się na dodawaniu i mnożeniu. Odejmować i dzielić liczby wymierne można, wykorzystując odpowiednio dodawanie (wystarczy zmienić znak drugiego ułamka na przeciwny) oraz mnożenie (wystarczy zamienić miejscami licznik i mianownik drugiego ułamka).

Dodawanie liczb wymiernych

Aby dodać dwa ułamki, można wymnożyć ich mianowniki, dodać liczniki mnożone odpowiednio przez mianowniki, a następnie (jeśli to możliwe) skrócić otrzymany ułamek.

Dodając ułamki w ten sposób, korzystalibyśmy ze wzoru:

$$\frac{l_1}{m_1} + \frac{l_2}{m_2} = \frac{l_1 \cdot m_2 + l_2 \cdot m_1}{m_1 \cdot m_2}$$

Podana metoda nie jest jednak najlepsza, ponieważ w wyniku jej zastosowania możemy otrzymać duże liczby w liczniku i mianowniku. Dlatego na lekcjach matematyki sprowadzaliście ułamki do wspólnego mianownika, którym była **najmniejsza wspólna wielokrotność (NWW)** obu mianowników. Podobnie postąpimy, pisząc program dodający dwa ułamki. Dodatkowo skorzystamy z **zależności między NWD i NWW**, która mówi, że dla liczb całkowitych dodatnich a i b:

$$\mathrm{NWW}(a,b) = \frac{a \cdot b}{\mathrm{NWD}(a,b)}$$

Najmniejsza wspólna wielokrotność (NWW) dwóch liczb całkowitych dodatnich to najmniejsza liczba całkowita dodatnia, która dzieli się bez reszty przez obie te liczby.

Zależność między NWD i NWW

Ćwiczenie 5

Napisz funkcję `NWW`, obliczającą najmniejszą wspólną wielokrotność dwóch liczb całkowitych dodatnich. Możesz wykorzystać funkcję `NWD`. Postaraj się zapisać funkcję `NWW` tak, aby wykonywać operacje arytmetyczne na jak najmniejszych liczbach. Napisz program sprawdzający, czy dla poprawnych danych funkcja `NWW` zwróci poprawną wartość.

Aby dodać dwa ułamki, gdy w każdym z nich licznik i mianownik są liczbami względnie pierwszymi, można wykorzystać następujący wzór:

$$\frac{l_1}{m_1} + \frac{l_2}{m_2} = \frac{l_1 \cdot \frac{m_2}{\mathrm{NWD}(m_1,m_2)} + l_2 \cdot \frac{m_1}{\mathrm{NWD}(m_1,m_2)}}{\mathrm{NWW}(m_1,m_2)}$$

Ćwiczenie 6

Dodaj ułamki $\frac{5}{12}$ i $\frac{3}{20}$, stosując powyższy wzór.

Żeby nie skracać wyniku, można (po obliczeniu licznika) od razu wyznaczyć liczbę, przez którą należy podzielić licznik i mianownik. Wprowadźmy następujące oznaczenia:

$$nwd_1 = \mathrm{NWD}(m_1, m_2)$$

$$l = l_1 \cdot \frac{m_2}{nwd_1} + l_2 \cdot \frac{m_1}{nwd_1}$$

$$nwd_2 = \mathrm{NWD}(l, nwd_1)$$

Wówczas:

$$\frac{l_1}{m_1} + \frac{l_2}{m_2} = \frac{\frac{l}{nwd_2}}{\frac{m_1}{nwd_1} \cdot \frac{m_2}{nwd_2}}$$

Kiedy zastosujemy podane oznaczenia dla ułamków $\frac{4}{15}$ i $\frac{9}{10}$, otrzymamy:

$nwd_1 = \text{NWD}(15, 10) = 5$

$l = 4 \cdot \frac{10}{5} + 9 \cdot \frac{15}{5} = 35$

$nwd_2 = \text{NWD}(35, 5) = 5$

$$\frac{4}{15} + \frac{9}{10} = \frac{\frac{35}{5}}{\frac{15}{5} \cdot \frac{10}{5}} = \frac{7}{6}$$

Teraz napiszemy program dodający dwie liczby wymierne. Przyjmiemy założenie, że znak ułamka będzie pamiętany w liczniku. Licznik może więc być dowolną liczbą całkowitą, a mianownik – tylko liczbą całkowitą dodatnią. Oto specyfikacja problemu dodawania dwóch ułamków oraz zapisany w pseudokodzie algorytm wykorzystujący wzór ze s. 111:

Specyfikacja

Dane: a, b – liczby wymierne postaci $\frac{l}{m}$, gdzie l i m to liczby całkowite, $m > 0$.

Wynik: c – liczba wymierna, $c = a + b$.

```
nwd1 ← NWD(a.mian,b.mian)
c.licz ← a.licz*(b.mian div nwd1) +
         b.licz*(a.mian div nwd1)
nwd2 ← NWD(|c.licz|,nwd1)
c.licz ← c.licz div nwd2
c.mian ← (a.mian div nwd1) * (b.mian div nwd2)
```

Licznik może być liczbą ujemną, a wartość wyniku powinna być dodatnia, dlatego przy obliczaniu wartości nwd2 parametrem aktualnym jest wartość bezwzględna licznika. Oto fragment kodu źródłowego programu z implementacją funkcji dodającej dwa ułamki zgodnie z powyższymi specyfikacją i algorytmem:

Fragment kodu źródłowego programu dodającego dwie liczby wymierne – definicja funkcji Dodaj

```
1.  wymierna Dodaj(wymierna a, wymierna b)
2.  {
3.      wymierna c;
4.      int nwd1=NWD(a.mian,b.mian);
5.      c.licz=a.licz*(b.mian/nwd1)+b.licz*(a.mian/nwd1);
6.      int nwd2=NWD(abs(c.licz),nwd1);
7.      c.licz=c.licz/nwd2;
8.      c.mian=(a.mian/nwd1)*(b.mian/nwd2);
9.      return c;
10. }
```

W linii 6 parametrem funkcji NWD jest wartość bezwzględna licznika. Do obliczenia wartości bezwzględnej użyliśmy **funkcji** abs z biblioteki cmath. Aby móc skorzystać z tej funkcji, trzeba na początku kodu programu dołączyć dyrektywę `#include <cmath>`.

Funkcja abs

Fragment kodu źródłowego programu, który wykorzystuje funkcję dodającą liczby wymierne oraz umożliwia sprawdzenie, czy funkcja działa poprawnie, może być następujący:

Fragment kodu źródłowego programu dodającego dwie liczby wymierne – definicja funkcji PiszUlamek **oraz funkcja** main

```
1.  void PiszUlamek(wymierna a)
2.  {
3.      if (a.licz<0) cout<<"(";
4.      cout<<a.licz;
5.      if (a.mian>1) cout<<"/"<<a.mian;
6.      if (a.licz<0) cout<<")";
7.  }
8.
9.  int main()
10. {
11.     wymierna a, b, c;
12.     cout<<"Licznik1 = "; cin>>a.licz;
13.     cout<<"Mianownik1 = "; cin>>a.mian;
14.     a=Skroc(a);
15.     cout<<"Licznik2 = "; cin>>b.licz;
16.     cout<<"Mianownik2 = "; cin>>b.mian;
17.     b=Skroc(b);
18.     c=Dodaj(a,b);
19.     PiszUlamek(a);
20.     cout<<" + ";
21.     PiszUlamek(b);
22.     cout<<" = ";
23.     PiszUlamek(c);
24.     return 0;
25. }
```

Funkcja main najpierw wypisuje komunikaty z prośbą o podanie dwóch ułamków, a następnie skraca te ułamki (linie 12–17). W liniach 14 i 17 wywoływana jest funkcja Skroc, skracająca ułamek będący parametrem, czyli dzieląca licznik i mianownik ułamka przez ich największy wspólny dzielnik. W linii 18 wywoływana jest funkcja dodająca ułamki, a wynik jej działania zapamiętywany jest w zmiennej c. Kolejne linie kodu odpowiadają za wypisywanie pełnego działania: pierwszy ułamek, znak plus, drugi ułamek, znak równości i wynik. Do wypisania ułamka została zdefiniowana funkcja PiszUlamek (linie 1–7). Jest to **funkcja typu void**, czyli bez typu – nie zwraca ona żadnej wartości, wykonuje tylko instrukcje, w naszym przypadku wypisanie liczby wymiernej określonej parametrem.

Funkcja typu void

Liczba ujemna na ekranie jest ujęta w nawiasy okrągłe (linie 3 i 6). Kreska ułamkowa (/) i mianownik są wypisywane tylko wtedy, gdy liczba nie jest całkowita, czyli mianownik jest większy od 1 (linia 5). Oto przykład wykonania programu:

```
licznik1 = 5
mianownik1 = 12
licznik2 = -3
mianownik2 = 20
5/12 + (-3/20) = 4/15
```

Rys. 7.3. Przykład wykonania programu dodającego dwa ułamki

Mnożenie liczb wymiernych

Warto wiedzieć

Liczbami wymiernymi posługiwali się już Egipcjanie w czasach starożytnych. Zapisywali oni te liczby w postaci sumy ułamków, których liczniki są równe jeden (ułamek o liczniku 1 nazywany jest ułamkiem egipskim). Na przykład: 7/8 = 1/2 + 1/4 + 1/8. Taki zapis ułatwiał m.in. porównywanie liczb.

Jeśli chcemy obliczyć iloczyn dwóch ułamków zwykłych, możemy pomnożyć przez siebie ich liczniki i mianowniki, a następnie skrócić otrzymany wynik. Żeby wykonywać operacje na mniejszych liczbach, skracamy najpierw ułamki „na krzyż" zgodnie z poniższym wzorem (tak samo postępujecie na lekcjach matematyki).

$$\frac{l_1}{m_1} \cdot \frac{l_2}{m_2} = \frac{\frac{l_1}{\mathrm{NWD}(l_1, m_2)} \cdot \frac{l_2}{\mathrm{NWD}(l_2, m_1)}}{\frac{m_1}{\mathrm{NWD}(l_2, m_1)} \cdot \frac{m_2}{\mathrm{NWD}(l_1, m_2)}}$$

W algorytmie mnożenia ułamków można także wykorzystać skracanie ułamka. W tym celu należy zamienić miejscami mianowniki lub liczniki ułamków. Na przykład:

$$\frac{3}{5} \cdot \frac{10}{21} = \frac{3}{21} \cdot \frac{10}{5} = \frac{1}{7} \cdot \frac{2}{1} = \frac{2}{7}$$

Oto funkcja realizująca ten algorytm zapisana w pseudokodzie:

```
funkcja Mnoz(a,b)
    pom ← a.mian
    a.mian ← b.mian
    b.mian ← pom
    Skroc(a)
    Skroc(b)
    c.licz ← a.licz * b.licz
    c.mian ← a.mian * b.mian
    zwróć c i zakończ
```

Ćwiczenie 7

Zapisz funkcję `wymierna Mnoz(wymierna a, wymierna b)`, realizującą algorytm opisany za pomocą powyższego pseudokodu. Napisz program testujący mnożenie ułamków, analogiczny do programu dodającego ułamki.

Podsumowanie

- Algorytm Euklidesa znajdujący największy wspólny dzielnik (NWD) dwóch liczb całkowitych dodatnich z wykorzystaniem odejmowania polega na zastępowaniu większej liczby różnicą większej i mniejszej liczby tak długo, aż otrzymamy dwa razy tę samą liczbę – będzie ona równa szukanemu NWD.
- Algorytm Euklidesa z wykorzystaniem dzielenia polega na zastępowaniu większej liczby resztą z dzielenia tej liczby przez mniejszą liczbę tak długo, aż reszta z dzielenia będzie równa zero. Ostatni dzielnik jest poszukiwanym NWD.
- Struktura w języku C++ pozwala na zapamiętanie pod jedną nazwą wielu zmiennych różnych typów.
- Liczbę wymierną można reprezentować w programie jako strukturę składającą się z dwóch zmiennych całkowitych, odpowiadających licznikowi i mianownikowi ułamka.
- W działaniach na liczbach wymiernych wykorzystuje się algorytm Euklidesa do skracania ułamka oraz sprowadzania ułamków do wspólnego mianownika.
- Zmienne lokalne to zmienne deklarowane wewnątrz funkcji.
- Zmienne globalne to zmienne deklarowane poza definicją funkcji i dostępne dla różnych funkcji.
- Funkcja powinna się odwoływać tylko do swoich parametrów i zmiennych lokalnych. Jest wtedy uniwersalna – można ją zastosować też w innych programach.
- Przekazanie parametru przez wartość oznacza, że parametr jest traktowany tak samo jak zmienna lokalna. Tworzona jest jego kopia (zmienna lokalna), na której są wykonywane wszystkie operacje. Po zakończeniu działania funkcji wartość parametru się nie zmienia.
- Funkcja typu `void` nie zwraca żadnego wyniku, wykonuje tylko określone czynności.

Zadania

* **1** Zapisz w postaci listy kroków algorytm znajdowania największego wspólnego dzielnika dwóch liczb całkowitych dodatnich z wykorzystaniem rozkładu liczb na czynniki pierwsze.

* **2** Napisz program, który wczyta n liczb całkowitych dodatnich i wyznaczy ich największy wspólny dzielnik.

* **3** Napisz program, który dla dwóch dodatnich liczb całkowitych a i b, będących długościami boków prostokąta, wyznaczy najmniejszą liczbę identycznych kwadratów nienachodzących na siebie i całkowicie wypełniających prostokąt.

* **4** Omówiony w tym temacie program skracający ułamek zmodyfikuj tak, aby wyznaczał dla ułamka niewłaściwego część całkowitą.

* **5** Zapisz specyfikację oraz algorytm odejmowania dwóch ułamków zwykłych. Napisz funkcję wykonującą odejmowanie oraz program testujący jej działanie.

** **6** Wykaż, że dla dwóch liczb całkowitych dodatnich a i b, gdzie $a > b$, $NWD(a, b) = NWD(a \bmod b, b)$.

** **7** Napisz program, który dla danych dodatnich liczb całkowitych a i b, będących długościami boków prostokąta, wyznaczy najmniejszą liczbę kwadratów nienachodzących na siebie i całkowicie wypełniających prostokąt. Długości boków kwadratów mogą być różne.

** **8** Pudełko z dużymi bombkami choinkowymi zawiera n bombek, a pudełko z małymi – m bombek. Napisz program, który obliczy, ile minimum pudełek z dużymi oraz małymi bombkami potrzeba, aby powiesić na choince tyle samo bombek obydwu wielkości i wykorzystać wszystkie bombki z tych pudełek.

** **9** Zapisz specyfikację oraz algorytm dzielenia dwóch ułamków zwykłych. Napisz funkcję wykonującą dzielenie oraz program testujący jej działanie.

*** **10** Wyobraź sobie, że chcesz zaczerpnąć n litrów wody ze studni. Masz do dyspozycji dwa wiadra o objętości odpowiednio m i k litrów, gdzie $m \leqslant k$ i $n \leqslant k$. Możesz wykonywać następujące czynności: nalać pełne wiadro wody ze studni, opróżnić wiadro, wylewając wodę do studni, przelać wodę z jednego wiadra do drugiego, tak aby jedno było pełne lub drugie puste. Ustal, jaki warunek muszą spełniać dane, aby można było odmierzyć n litrów wody. Podaj sposób wyznaczania sekwencji czynności wlewania i wylewania wody.

8. Szyfr Cezara i inne szyfry podstawieniowe

Ludzie od wieków przekazywali sobie tajne wiadomości, a szyfrowanie było znane już w czasach starożytnych. Korzystali z niego władcy, wojskowi, szpiedzy, a także kochankowie. Dostęp do informacji często oznaczał władzę, zwycięstwo czy pieniądze, dlatego wymyślano szyfry coraz trudniejsze do złamania. Mimo że dziś, w epoce komputerów, szyfry używane przez naszych przodków nie stanowią już żadnego zabezpieczenia, warto się dowiedzieć, na czym polegały. W tym temacie napiszesz programy wykorzystujące szyfr Cezara i poznasz kilka innych szyfrów.

Cele lekcji

- Dowiesz się, na czym polegają szyfry podstawieniowe, oraz poznasz przykłady takich szyfrów.
- Zaszyfrujesz informację szyfrem Cezara.
- Nauczysz się wczytywać do programu dane z pliku tekstowego i zapisywać wyniki w pliku tekstowym.
- Wykorzystasz w szyfrowaniu alfabet uwzględniający litery z polskimi znakami diakrytycznymi.

Podstawy kryptografii, podręcznik *Informatyka na czasie 1. Zakres rozszerzony*, s. 214–221

O **podstawach kryptografii**, czyli zagadnieniach związanych z szyfrowaniem informacji, mówiliśmy w części pierwszej podręcznika. Przypomnijmy kilka pojęć, które wykorzystamy w tym temacie. Podczas szyfrowania przy użyciu **klucza** informacja jawna przekształcana jest w szyfrogram (kryptogram). Aby odbiorca mógł poznać informację jawną, czyli odszyfrować kryptogram, musi znać klucz, który mu to umożliwi. Klucze szyfrujący i deszyfrujący mogą się różnić.

Klucz to dane umożliwiające zaszyfrowanie lub odszyfrowanie informacji. Kluczem może być np. liczba lub tekst.

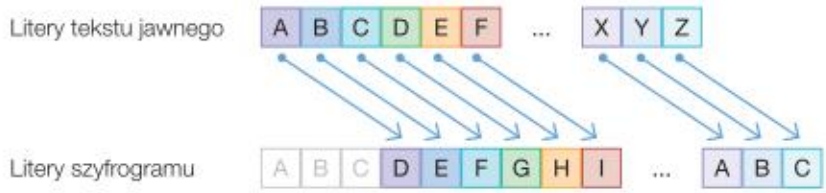
8.1. Szyfrujemy napis szyfrem Cezara

Szyfry podstawieniowe to szyfry, w których informacje utajnia się, zamieniając każdy znak tekstu jawnego na inny znak lub grupę znaków. Przykładem szyfru podstawieniowego jest sposób wymyślony w starożytności przez Juliusza Cezara, nazwany potem **szyfrem Cezara**.

Szyfr podstawieniowy

Szyfr Cezara

Utajnianie informacji tą metodą polega na zastąpieniu każdej litery tekstu jawnego literą oddaloną od niej o stałą liczbę pozycji w alfabecie. Kierunek zamiany wszystkich liter musi być ten sam. Przesunięcie odbywa się cyklicznie, tzn. po ostatniej literze alfabetu następuje pierwsza. Kluczem dla szyfru Cezara jest liczba określająca przesunięcie w alfabecie. Juliusz Cezar stosował klucz 3, tzn. literę A zastępował literą D, literę B – literą E itd. Przedstawia to rysunek 8.1 na s. 118.



Rys. 8.1. Szyfrowanie liter alfabetu łacińskiego szyfrem Cezara z kluczem 3

Warto wiedzieć

Kilkukrotne zastosowanie szyfru Cezara nie zwiększa poziomu utajnienia informacji. Na przykład użycie klucza 5, a potem klucza 8 daje taki sam wynik jak szyfrowanie z kluczem 13.

Na przykład słowo INFORMATYKA zaszyfrowane z kluczem 3 to LQIRUPDWBND.

Ćwiczenie 1

a. Zaszyfruj wyraz KRYPTOLOGIA szyfrem Cezara z kluczem 25.
b. Odszyfruj kryptogram QVBMZVMB, zaszyfrowany szyfrem Cezara z kluczem 8.

Szyfrujemy napis złożony z wielkich liter

Teraz napiszemy program, który zakoduje szyfrem Cezara napis złożony wyłącznie z wielkich liter alfabetu łacińskiego. Ponieważ ten alfabet liczy 26 liter, ograniczymy wartość klucza do liczb większych od 0 i mniejszych niż 26. Oto specyfikacja problemu:

Specyfikacja

Dane: `t_jawny` – napis złożony wyłącznie z wielkich liter alfabetu łacińskiego,
`klucz` – liczba całkowita, $0 < \texttt{klucz} < 26$.

Wynik: `szyfrogram` – napis `t_jawny` zaszyfrowany szyfrem Cezara z wykorzystaniem klucza `klucz`.

Warto wiedzieć

Celem kryptografii jest ukrycie znaczenia informacji, a nie faktu jej istnienia. Steganografia zaś obejmuje ogół metod służących do ukrywania istnienia komunikatu, bez konieczności szyfrowania. Sposoby takie były znane już w starożytności. W czasie II wojny światowej tajne wiadomości fotografowano, a następnie pomniejszano i naklejano na kropkę w tekście jawnym. Obecnie informacje ukrywa się np. w plikach graficznych.

Zapis algorytmu prowadzącego do rozwiązania problemu może wyglądać następująco:

```
szyfrogram ← ""
dla i ← 0, 1, ..., długość t_jawny - 1 wykonuj
    kod ← kod_ASCII(t_jawny[i]) + klucz
    jeśli kod > kod_ASCII('Z') to kod ← kod - 26
    szyfrogram ← szyfrogram + znak_ASCII(kod)
```

Kod ASCII litery szyfrogramu zapamiętamy w pomocniczej zmiennej kod typu `int`. Aby wyznaczyć kod ASCII litery szyfrogramu, do kodu litery tekstu jawnego dodajemy przesunięcie, czyli wartość klucza. Jeśli przekroczymy alfabet, czyli otrzymana wartość będzie większa od kodu litery Z (liczby 90), musimy odjąć 26 (liczbę liter alfabetu), aby otrzymać kod litery z początku alfabetu.

Na przykład podczas szyfrowania litery Y w słowie INFORMATYKA z kluczem 3 otrzymamy literę B. Kod ASCII litery Y to 89. Po dodaniu 3 uzyskujemy 92, a więc wartość większą od kodu litery Z. Odejmujemy 26 i otrzymujemy 66, czyli kod litery B.

Oto kod źródłowy programu realizującego algorytm zapisany wcześniej w pseudokodzie:

Kod źródłowy programu szyfrującego wielkie litery alfabetu łacińskiego szyfrem Cezara z kluczem większym od 0 i mniejszym od 26

```
1.  #include <iostream>
2.  #include <string>
3.
4.  using namespace std;
5.
6.  int main()
7.  {
8.      string t_jawny, szyfrogram="";
9.      int i, klucz;
10.     int kod;
11.     cout<<"Tekst jawny: "; cin>>t_jawny;
12.     cout<<"Klucz: "; cin>>klucz;
13.     for (i=0;i<t_jawny.size();i++)
14.     {
15.         kod=t_jawny[i]+klucz;
16.         if (kod>'Z') kod=kod-26;
17.         szyfrogram=szyfrogram+char(kod);
18.     }
19.     cout<<"Szyfrogram: "<<szyfrogram;
20.     return 0;
21. }
```

W pętli **for** (linie 13–18) tekst jest szyfrowany zgodnie z algorytmem przedstawionym za pomocą pseudokodu. W linii 15 obliczamy kod ASCII pojedynczego znaku szyfrogramu (jednej litery) i zapisujemy go w zmiennej pomocniczej `kod`. Jeśli wartość zmiennej `kod` jest większa od 90, odejmujemy 26 (linia 16). W linii 17 do wartości zmiennej `szyfrogram` dołączamy otrzymaną literę szyfrogramu. Zauważ, że w linii 15 nie potrzebujemy konwersji znaku na kod, ponieważ w języku C++ znaki są reprezentowane przez wartości liczbowe (kody ASCII).

W programie szyfrującym tekst, kiedy wyznaczamy kod ASCII znaku szyfrogramu, możemy zrezygnować z instrukcji warunkowej i zamiast tego skorzystać z operacji **mod**. Wówczas instrukcja prowadząca do wyznaczenia kodu ASCII znaku szyfrogramu wyglądałaby następująco:

```
kod ← (kod_ASCII(t_jawny[i]) + klucz -
       kod_ASCII('A')) mod 26 + kod_ASCII('A')
```

Dobra rada

Kiedy szyfrujesz szyfrem Cezara tekst składający się z liter 26-znakowego alfabetu, a klucz jest większy od 26, możesz zastosować operację `klucz mod 26`. Na przykład szyfrowanie z kluczem 32 daje taki sam wynik jak przy użyciu klucza 6.

Aby można było zastosować operację **mod**, kody wielkich liter powinny się zaczynać od zera. Dlatego resztę z dzielenia przez 26 obliczamy dla wyrażenia kod_ASCII(t_jawny[i]) + klucz pomniejszonego o kod ASCII litery A. Po wyznaczeniu reszty z dzielenia dodajemy kod litery A. Na przykład dla litery Y i klucza 3 otrzymamy (89 + 3 – 65) **mod** 26 + 65 = 66, czyli kod ASCII litery B.

Ćwiczenie 2

Program szyfrujący napis złożony z wielkich liter alfabetu zmodyfikuj tak, aby przy wyznaczaniu litery szyfrogramu zastosować operację **mod**.

Szyfrujemy napis złożony z liter i innych znaków

Warunek, że tekst jawny musi się składać tylko z wielkich liter alfabetu łacińskiego, jest dużym ograniczeniem. Dlatego program zmodyfikujemy tak, aby tekst jawny mógł zawierać małe i wielkie litery alfabetu łacińskiego oraz inne znaki, np. znaki przestankowe (przecinki, kropki itp.) oraz spacje. Szyfrować będziemy tylko litery, pozostałe znaki pozostaną w szyfrogramie niezmienione. Oto specyfikacja problemu:

Specyfikacja

Dane: t_jawny – napis złożony z liter alfabetu łacińskiego oraz innych znaków,
klucz – liczba całkowita, 0 < klucz < 26.

Wynik: szyfrogram – litery tekstu jawnego zaszyfrowane szyfrem Cezara z kluczem klucz, pozostałe znaki bez zmiany.

Ponieważ napis może zawierać spacje, do wczytania wiersza tekstu trzeba użyć **funkcji** getline. Pętla przeglądająca znaki napisu w zmodyfikowanym programie może wyglądać następująco:

Funkcja getline, s. 67

Fragment kodu źródłowego programu szyfrującego szyfrem Cezara z kluczem większym od 0 i mniejszym od 26 małe i wielkie litery alfabetu łacińskiego – pętla przeglądająca znaki napisu

```
1.  for (i=0;i<t_jawny.size();i++)
2.      if (toupper(t_jawny[i])>='A' && toupper(t_jawny[i])<='Z')
3.      {
4.          kod=t_jawny[i]+klucz;
5.          if ((t_jawny[i]<='Z' && kod>'Z') ||
6.              (t_jawny[i]>='a' && kod>'z')) kod=kod-26;
7.          szyfrogram=szyfrogram+char(kod);
8.      }
9.      else szyfrogram=szyfrogram+t_jawny[i];
```

W pętli dodaliśmy instrukcję warunkową (linia 2), która bada, czy znak jest literą. Jeśli badany znak jest literą, to wykonywane są takie

same instrukcje jak w poprzednim programie. W przeciwnym przypadku do szyfrogramu dołączany jest znak bez zmiany. W warunku logicznym użyliśmy **funkcji** `toupper`. Aby z niej skorzystać, należy dołączyć bibliotekę `cctype`.

Funkcja `toupper`, s. 63

Ćwiczenie 3

Napisz program, który zaszyfruje szyfrem Cezara linię tekstu złożonego z liter alfabetu łacińskiego (małych i wielkich) oraz znaków przestankowych i spacji. Szyfrowane mają być tylko litery, pozostałe znaki powinny pozostać w szyfrogramie niezmienione.

Zapamiętaj

Utajnienie tekstu przy użyciu szyfru podstawieniowego polega na zamianie każdego znaku tekstu jawnego na inny znak lub grupę znaków. Przykładem szyfru podstawieniowego jest szyfr Cezara. Szyfrowanie tą metodą polega na zastąpieniu każdej litery tekstu jawnego literą oddaloną od niej o stałą liczbę pozycji w alfabecie.

8.2. Szyfrujemy tekst wczytany z pliku tekstowego

Omówione wcześniej programy szyfrowały szyfrem Cezara tekst wpisany z klawiatury. Jest to wygodne, gdy tekst jawny nie jest długi. Jeśli jednak chcielibyśmy zaszyfrować dłuższy tekst, np. zapisany w pliku, przepisywanie go mogłoby być czasochłonne. Dlatego napiszemy program, w którym tekst jawny wczytamy z pliku tekstowego. Otrzymany szyfrogram także zostanie zapisany w pliku. Z klawiatury program wczyta jedynie klucz, za pomocą którego zaszyfruje tekst.

Dobra rada

Umiejętności wczytywania do programu danych z pliku oraz zapisywania wyników do pliku przydadzą ci się podczas rozwiązywania wielu zadań.

Kolejne wiersze tekstu z pliku tekstowego program będzie przetwarzał sekwencyjnie, jeden po drugim, tzn. odczyta pierwszy wiersz z pliku wejściowego, zaszyfruje go i zapisze szyfrogram do pliku wynikowego. Następnie pobierze drugi wiersz tekstu, zaszyfruje go, wynik zapisze w pliku itd., aż do końca pliku wejściowego. Zapis algorytmu w pseudokodzie może być następujący:

```
dopóki nie koniec pliku wejściowego wykonuj
    s ← linia tekstu z pliku wejściowego
    s ← Cezar(s,klucz)
    // zaszyfrowanie linii tekstu
    zapisz s do pliku wynikowego
```

W pseudokodzie wykorzystujemy funkcję `Cezar`, której parametrami są linia tekstu `s` oraz klucz, a wynikiem jest zaszyfrowana linia tekstu. Funkcja ta wykonuje takie same zadania jak program omówiony w poprzednim zagadnieniu. Zapiszmy jej kod źródłowy.

Kod źródłowy funkcji szyfrującej wielkie i małe litery alfabetu łacińskiego szyfrem Cezara z kluczem większym od 0 i mniejszym od 26

```
string Cezar(string s, int klucz)
{
    string szyfrogram="";
    int kod;
    for (int i=0;i<s.size();i++)
        if (toupper(s[i])>='A' && toupper(s[i])<='Z')
        {
            kod=s[i]+klucz;
            if ((t_jawny[i]<='Z' && kod>'Z') ||
                (t_jawny[i]>='a' && kod>'z')) kod=kod-26;
            szyfrogram=szyfrogram+char(kod);
        }
        else szyfrogram=szyfrogram+s[i];
    return szyfrogram;
}
```

Ćwiczenie 4

Napisz program szyfrujący szyfrem Cezara linię tekstu. Klucz powinien być większy od 0 i mniejszy od 26. Wykorzystaj funkcję Cezar.

Obsługa plików tekstowych

Obsługa plików tekstowych, s. 336–338

Zmienna plikowa

Korzystanie z plików wymaga dodania w kodzie programu dyrektywy `#include <fstream>` – powoduje ona dołączenie biblioteki służącej do **obsługi plików**. Komunikacja z plikami z poziomu programu tworzonego w języku C++ odbywa się za pośrednictwem **zmiennych plikowych**, np. zmiennych typu `ifstream` oraz `ofstream`. Typ `ifstream` służy do obsługi plików wejściowych (do odczytu), a typ `ofstream` – do obsługi plików wyjściowych (do zapisu).

W pracy z plikiem tekstowym można wyodrębnić kilka etapów.

- **Otwarcie pliku z danymi do odczytu**

  Plik tekstowy możemy otworzyć już podczas deklaracji zmiennej plikowej. Na przykład dzięki poniższej instrukcji otworzymy do odczytu plik *dane.txt*. Będzie on reprezentowany w programie przez zmienną plikową `wejscie` typu `ifstream`. Plik otwierany w ten sposób musi istnieć. Powinien też być zapisany w folderze, w którym jest zapisany wykorzystujący go program.

  ```
  ifstream wejscie("dane.txt");
  ```

Warto wiedzieć

Nazwa `ifstream` pochodzi od angielskiego wyrażenia *input file stream*.

- **Utworzenie pliku przeznaczonego do zapisu danych**

  Podana niżej instrukcja utworzy plik *wyniki.txt*. W programie będzie reprezentowany przez zmienną plikową `wyjscie` typu `ofstream`. Plik zostanie utworzony w folderze, z którego uruchomiony jest program. Jeśli plik o danej nazwie już istnieje, to zostanie skasowany.

  ```
  ofstream wyjscie("wyniki.txt");
  ```

Warto wiedzieć

Nazwa `ofstream` pochodzi od angielskiego wyrażenia *output file stream*.

- **Wczytanie danych z pliku oraz zapisanie danych w pliku**
  Wczytanie i wypisanie danych odbywa się podobnie jak przy korzystaniu z instrukcji `cin` i `cout`. Używamy tylko innych nazw – zamiast polecenia `cin` lub `cout` podajemy nazwę zmiennej plikowej reprezentującej odpowiednio plik wejściowy lub wyjściowy.

```
wejscie>>s; // wczytanie do zmiennej s danych z pliku
            // reprezentowanego przez zmienną wejscie
getline(wejscie,s); // wczytanie wiersza danych z pliku
                    // reprezentowanego przez zmienną
                    // wejscie i zapisanie go w zmiennej s
wyjscie<<s; // zapisanie wartości zmiennej s w pliku
            // reprezentowanym przez zmienną wyjscie
```

- **Zamknięcie pliku**
  Plik zamykamy za pomocą **metody** `close`. Na przykład pierwsza z poniższych instrukcji powoduje zamknięcie pliku skojarzonego ze zmienną `wejscie`, a druga – ze zmienną `wyjscie`. Należy pamiętać, że przy korzystaniu z **metod** stosujemy notację z kropką.

  Metoda `close`

  Metoda, s. 39

```
wejscie.close();
wyjscie.close();
```

Jeśli nie zamkniemy pliku przeznaczonego do zapisu, część danych możemy utracić.

Program szyfrujący tekst wczytany z pliku tekstowego

Funkcja `main` programu szyfrującego tekst zapisany w pliku tekstowym i zapisującego szyfrogram także do pliku tekstowego może być następująca:

```
1.  int main()
2.  {
3.      ifstream wejscie("tekst_jawny.txt");
4.      ofstream wyjscie("szyfrogram.txt");
5.      string s;
6.      int klucz;
7.      cout<<"Klucz: "; cin>>klucz;
8.      while (!wejscie.eof())
9.      {
10.         getline(wejscie,s);
11.         wyjscie<<Cezar(s,klucz)<<endl;
12.     }
13.     wejscie.close();
14.     wyjscie.close();
15.     cout<<"Plik szyfrogram.txt zostal utworzony";
16.     return 0;
17. }
```

Fragment kodu źródłowego programu szyfrującego szyfrem Cezara dane wczytane z pliku tekstowego – funkcja `main`

Dobra rada

Jeśli chcesz wykorzystać w programie pliki tekstowe, pamiętaj o dodaniu na początku kodu dyrektywy:
`#include <fstream>`

W linii 3 deklarujemy zmienną `wejscie` typu **`ifstream`** i otwieramy do odczytu skojarzony z nią plik *tekst_jawny.txt*. W linii 4 deklarujemy zmienną `wyjscie` typu **`ofstream`** i otwieramy do zapisu skojarzony z nią plik *szyfrogram.txt*.

Pętla **`while`** (linie 8–12) wykonywana jest, dopóki nie zostaną odczytane wszystkie dane z pliku. W warunku pętli korzystamy z **metody** `eof`. Przyjmuje ona wartość **`true`**, kiedy program napotka koniec pliku (nie można już dalej wczytywać danych), a wartość **`false`** – w przeciwnym przypadku.

Metoda eof

Warto wiedzieć

Nazwa metody `eof` to skrót angielskiego wyrażenia *end of file*, oznaczającego koniec pliku.

Odczyt linii tekstu z pliku, jej zaszyfrowanie i zapis do pliku wyjściowego odbywają się w ten sam sposób co w przypadku wczytywania z klawiatury i wypisywania na ekranie (linie 10–11). Po zakończeniu wykonywania pętli pliki *tekst_jawny.txt* oraz *szyfrogram.txt* są zamykane (linie 13–14). Ponieważ szyfrogram zapisywany jest do pliku, nie zobaczymy go na ekranie. Dlatego aby pokazać, że proces szyfrowania się zakończył, wypiszemy odpowiedni komunikat (linia 15).

Dobra rada

Pamiętaj o zamykaniu plików. Jeśli nie zamkniesz pliku przeznaczonego do zapisu, część danych możesz stracić.

Ćwiczenie 5

Napisz program, który zaszyfruje szyfrem Cezara tekst wczytany z pliku tekstowego przekazanego przez nauczyciela (np. *tekst_jawny_1.txt*). Klucz powinien być większy od 0 i mniejszy od 26. Do obsługi plików tekstowych (w szczególności obejrzenia pliku wynikowego) możesz użyć systemowego notatnika.

Zapamiętaj

Praca z plikami w programie odbywa się za pośrednictwem specjalnych zmiennych plikowych. Są to zmienne typu obiektowego. Aby wczytać dane z pliku oraz zapisać wynik w pliku, postępujemy podobnie jak przy stosowaniu poleceń `cin` i `cout`. Zamiast nazwy polecenia podajemy nazwę zmiennej plikowej reprezentującej odpowiednio plik wejściowy lub wyjściowy. Trzeba pamiętać o otwarciu i zamknięciu pliku.

8.3. Szyfrujemy tekst z pliku z uwzględnieniem polskich znaków diakrytycznych

Program szyfrujący napis wczytany z pliku tekstowego nie uwzględniał polskich znaków diakrytycznych. Pokażemy teraz, jak w procesie szyfrowania wykorzystać alfabet składający się z 35 liter: 26 liter alfabetu łacińskiego oraz 9 liter z polskimi znakami diakrytycznymi. W programie nie będziemy się odwoływać do kodów ASCII, ale zdefiniujemy dwie stałe `alfabet_m` i `alfabet_w`, których wartościami będą odpowiednio alfabet złożony z małych i wielkich liter.

Warto wiedzieć

W języku polskim litery Q, V i X występują tylko w słowach zapożyczonych z innych języków. Współczesny polski alfabet składa się z 32 liter.

Stałe różnią się od zmiennych tym, że w trakcie działania programu nie można zmieniać ich wartości. **Definicję stałej** w języku C++ rozpoczynamy od słowa kluczowego **const**. Po tym słowie podajemy typ stałej, jej nazwę oraz wartość. W naszym przypadku stałe będą typu **string**, a ich wartościami będą ciągi znaków składające się z kolejnych liter alfabetu. Oto definicja stałych w naszym programie:

- Stała
- Definicja stałej

```
const string alfabet_m="aąbcćdeęfghijklłmnńoópqrsśtuvwxyzźż";
const string alfabet_w="AĄBCĆDEĘFGHIJKLŁMNŃOÓPQRSŚTUVWXYZŹŻ";
```

Litery alfabetu będziemy numerować od 0 do 34. Sprawdzenie, czy znak tekstu jawnego jest literą, będzie polegało na poszukiwaniu tego znaku w jednym z napisów będących wartościami stałych. Wykorzystamy do tego **metodę** **find**. Jeśli znak zostanie odnaleziony, program wyznaczy odpowiadający mu znak szyfrogramu.

Metoda find, s. 67

Oto kod źródłowy funkcji szyfrującej pojedynczy znak (literę alfabetu):

- Kod źródłowy funkcji wyznaczającej literę szyfrogramu dla litery tekstu jawnego z polskimi znakami diakrytycznymi

```
1.  char Cezar_PL(char znak, int klucz)
2.  {
3.      int i=alfabet_m.find(znak);
4.      if (i>=0 && i<35)
5.      {
6.          i=(i+klucz)%35;
7.          return alfabet_m[i];
8.      }
9.      i=alfabet_w.find(znak);
10.     if (i>=0 && i<35)
11.     {
12.         i=(i+klucz)%35;
13.         return alfabet_w[i];
14.     }
15.     return znak;
16. }
```

W linii 3 sprawdzamy, czy litera tekstu jawnego znajduje się w alfabecie złożonym z małych liter, a dokładniej szukamy pozycji tego znaku w alfabecie. Jeśli pozycja zostanie znaleziona (czyli jeśli jej wartość jest między 0 a 34), to wyznaczona zostanie pozycja litery, która zastąpi literę tekstu jawnego w szyfrogramie (linia 6). Funkcja zwraca literę znajdującą się na tej pozycji (linia 7).

W liniach 10–14 wykonywane są analogiczne czynności dla wielkiej litery. Jeżeli znak nie zostanie odnaleziony ani wśród małych liter, ani wśród wielkich, to wynikiem jest znak bez zmiany (linia 15). Zmodyfikowany kod funkcji Cezar, wykorzystującej funkcję Cezar_PL, jest następujący.

Dobra rada

Pamiętaj, że gdy stosujesz funkcje zwracające wartość, musisz użyć polecenia **return**. Dzięki niemu zwrócisz wynik i zakończysz działanie funkcji.

Kod źródłowy funkcji szyfrującej szyfrem Cezara wielkie i małe litery alfabetu z polskimi znakami diakrytycznymi

```
1. string Cezar(string s, int klucz)
2. {
3.     string szyfrogram="";
4.     for (int i=0;i<s.size();i++)
5.         szyfrogram=szyfrogram+Cezar_PL(s[i],klucz);
6.     return szyfrogram;
7. }
```

Warto wiedzieć

Wiersz polecenia systemu Windows, w którym wypisywane są wyniki działania programu, wykorzystuje stronę kodową CP852. Edytor Code::Blocks używa strony kodowej CP1250. Dlatego w poleceniu **cout** nie stosujemy polskich znaków diakrytycznych.

Istnieją różne systemy kodowania polskich znaków. Aby program uwzględniający polskie znaki diakrytyczne zadziałał poprawnie, plik *cpp* z kodem źródłowym oraz wykorzystywany przez program plik tekstowy muszą używać tej samej strony kodowej. Na przykład w środowisku Code::Blocks pracującym w polskiej wersji systemu Windows należy użyć strony kodowej CP1250.

Ćwiczenie 6

Napisz program szyfrujący tekst zapisany w pliku, składający się z liter alfabetu łacińskiego i polskich znaków diakrytycznych. Znaki inne niż litery powinny zostać niezmienione. Wykorzystaj plik tekstowy przekazany przez nauczyciela (np. *tekst_jawny_2.txt*).

8.4. Przykłady innych szyfrów podstawieniowych

Na przestrzeni wieków wymyślono wiele różnych szyfrów podstawieniowych. Oto wybrane z nich.

Szyfr permutacyjny

Szyfr monoalfabetyczny to szyfr, w którym jednej literze tekstu jawnego odpowiada zawsze taka sama litera w szyfrogramie.

Szyfr Cezara jest przykładem **szyfru monoalfabetycznego**. Innym przykładem takiego szyfru jest szyfr permutacyjny, w którym kluczem może być dowolna **permutacja** szyfrowanych znaków. Na przykład dla wielkich liter alfabetu łacińskiego klucz może wyglądać następująco:

RSUZOCBEYXWILTMJGDQAPNVFHK

Permutacja zbioru to dowolny ciąg utworzony ze wszystkich elementów tego zbioru. Liczbę permutacji zbioru n-elementowego obliczamy ze wzoru: $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$

Szyfrując tekst tym kluczem, zamiast litery A podstawiamy literę R, zamiast B – literę S, zamiast C – literę U itd. Próba złamania tego szyfru polegająca na sprawdzeniu wszystkich kluczy nie powiedzie się, ponieważ jest ich bardzo wiele, np. permutacji 26 liter alfabetu jest:

$$26! = 403\,291\,461\,126\,605\,635\,584\,000\,000$$

Szyfr Vigenère'a

Żeby utrudnić złamanie szyfru, poszukiwano metod, w których nie każde wystąpienie danego znaku będzie szyfrowane tym samym znakiem. W ten sposób powstały m.in. **szyfry polialfabetyczne**. Jednym z najbardziej znanych jest szyfr Vigenère'a, opublikowany w 1586 r. Opierał się on na tabeli Trithemiusa, w której każdy kolejny wiersz powstaje z poprzedniego wiersza: pierwszą literę przenosi się na koniec wiersza.

Szyfr polialfabetyczny to szyfr, w którym jednej literze tekstu jawnego mogą odpowiadać różne litery w szyfrogramie.

Jak zakodować tekst szyfrem Vigenère'a?

Kluczem w szyfrze Vigenère'a jest dowolny tekst złożony z liter występujących w tabeli, np. VIGENERE. Każdej literze tekstu jawnego przyporządkowuje się literę znajdującą się na przecięciu wiersza wyznaczonego przez tę literę i kolumny odpowiadającej literze klucza. Klucz jest cyklicznie powielany.

Tekst jawny:	**Z**	A	K	R	E	S	R	O	Z	S	Z	E	R	Z	O	N	Y	I	N	F	O	R	M	A	T	Y	K	A
	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
Klucz:	**V**	I	G	E	N	E	R	E	V	I	G	E	N	E	R	E	V	I	G	E	N	E	R	E	V	I	G	E
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
Szyfrogram:	**U**	I	Q	V	R	W	I	S	U	A	F	I	E	D	F	R	T	Q	T	J	B	V	D	E	O	G	Q	E

																					V				
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	**U**	V	W	X	Y

Kolumna wyznaczona przez pierwszą literę klucza – literę V

Wynik szyfrowania pierwszej litery tekstu jawnego – litery Z

Wiersz wyznaczony przez pierwszą literę w tekście jawnym – literę Z

Co warto wiedzieć o szyfrze Vigenère'a?

Szyfr znany jako szyfr Vigenère'a stworzył w rzeczywistości w 1553 r. włoski kryptograf Giovan Battista Bellaso. Blaise de Vigenère jest autorem bardziej skomplikowanego systemu szyfrowania opartego na szyfrze Bellasa.

Szyfr Vigenère'a bardzo długo uchodził za niemożliwy do złamania. W 1854 r. złamał go Charles Babbage, angielski matematyk i konstruktor mechanicznych maszyn liczących.

Podsumowanie

- Zagadnieniami związanymi z szyfrowaniem informacji zajmuje się kryptografia.
- W procesie szyfrowania informacja jawna jest przekształcana w szyfrogram (kryptogram). Do szyfrowania i deszyfrowania informacji używa się specjalnego klucza, którym może być np. liczba lub tekst.
- Szyfry podstawieniowe to szyfry, w których informacje utajnia się, zamieniając każdy znak informacji jawnej na inny znak lub grupę znaków.
- Dane do programów można wczytywać z plików tekstowych. Wyniki także można zapisywać w plikach tekstowych.
- W programach można definiować własny alfabet, np. z polskimi znakami diakrytycznymi, wykorzystując stałe napisowe i funkcje operujące na napisach.
- Stałe w języku C++ można definiować przy użyciu słowa kluczowego `const`. Po tym słowie podajemy typ stałej, jej nazwę oraz wartość.
- Szyfr permutacyjny polega na zastępowaniu każdej litery tekstu jawnego odpowiadającą jej literą permutacji alfabetu. Permutacja pełni funkcję klucza.
- Szyfr Cezara i szyfr permutacyjny są przykładami podstawieniowych szyfrów monoalfabetycznych, w których każdej literze informacji jawnej odpowiada dokładnie jedna litera szyfrogramu.
- Szyfr Vigenère'a polega na wielokrotnym zastosowaniu szyfru Cezara w zależności od pozycji litery w tekście jawnym. Jest przykładem podstawieniowego szyfru polialfabetycznego – ta sama litera może być zaszyfrowana różnymi literami w zależności od pozycji zajmowanej w tekście jawnym.

Zadania

* **1** Poszukaj informacji na temat dwóch szyfrów podstawieniowych, innych niż wymienione w tym temacie. Następnie przygotuj prezentację, w której na przykładach objaśnisz zasadę działania każdego z tych szyfrów.

* **2** Napisz program szyfrujący szyfrem Cezara tekst zapisany wielkimi literami alfabetu łacińskiego. Kluczem może być dowolna liczba całkowita (także ujemna).

* **3** Napisz program szyfrujący metodą podstawieniową tekst zapisany wielkimi literami alfabetu łacińskiego. Zastosuj dwa różne klucze: jeden dla znaków na pozycjach parzystych, a drugi dla znaków na pozycjach nieparzystych.

** **4** Napisz program szyfrujący metodą podstawieniową tekst z pliku zapisany wielkimi literami alfabetu łacińskiego. Zastosuj różne klucze dla samogłosek i spółgłosek. Sprawdź działanie programu dla tekstu udostępnionego przez nauczyciela (np. w pliku *wiadomosc.txt*)

** **5** Napisz program szyfrujący metodą podstawieniową tekst z pliku (np. *informacja.txt*) zapisany wielkimi literami alfabetu łacińskiego. Zastosuj różne przesunięcia na poszczególnych pozycjach. Kluczem ma być liczba całkowita dodatnia, przy czym cyfra najbardziej znacząca określa przesunięcie pierwszej pozycji, kolejna cyfra – przesunięcie drugiej pozycji itd. Klucz stosuj cyklicznie. Na przykład klucz 2019 oznacza, że pierwsza litera jest szyfrowana kluczem 2, druga pozostaje bez zmian, trzecia jest szyfrowana kluczem 1, czwarta – kluczem 9, piąta ponownie kluczem 2 itd.

** **6** Napisz program szyfrujący tekst złożony z wielkich liter alfabetu łacińskiego z zastosowaniem szyfru permutacyjnego. Klucz – permutację liter alfabetu – możesz uzyskać np. za pomocą następującej funkcji:

```
#include <cstdlib>
#include <ctime>

string PermutacjaLiter()
{
    srand(time(NULL));
    string perm = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    int i, j, k;
    char ch;
    for (i=0;i<100;i++)
    {
        j=rand()%26;
        k=rand()%26;
        ch=perm[j];
        perm[j]=perm[k];
        perm[k]=ch;
    }
    return perm;
}
```

** **7** Napisz program szyfrujący tekst złożony z wielkich liter alfabetu łacińskiego szyfrem Vigenère'a z wykorzystaniem podanego klucza.

** **8** Napisz program deszyfrujący tekst zaszyfrowany szyfrem Vigenère'a z użyciem danego klucza.

*** **9** Poszukaj informacji na temat szyfru XOR, a następnie napisz program szyfrujący dany tekst szyfrem XOR z użyciem klucza liczbowego z zakresu od 1 do 255. Wynikiem powinien być plik zawierający ciąg liczb oddzielonych spacjami – znaki tekstu jawnego zaszyfrowane szyfrem XOR. Tekst do zaszyfrowania udostępni ci nauczyciel (np. w pliku *dane_xor.txt*).

Sposób na zadania

Zadanie 1

Oceń, czy poniższe zdania są prawdziwe. Wybierz **P**, jeśli zdanie jest prawdziwe, albo **F** – jeśli zdanie jest fałszywe.

Zadanie 1.1 (0–1)

1.	Liczby $ABCD_{16}$ i 1010101101001101_2 mają tę samą wartość.	P	F
2.	Liczby $A0B2_{16}$ i 120262_8 mają tę samą wartość.	P	F
3.	Liczby 120_3 i 16_9 mają różną wartość.	P	F
4.	Liczby 127_8 i 1123_4 mają różną wartość.	P	F

Rozwiązanie

Tagi: systemy liczbowe, zamiana reprezentacji liczb między systemami liczbowymi

Krok 1

Żeby rozwiązać to zadanie, nie trzeba przeliczać liczb na system dziesiętny – można skorzystać z zależności między systemami liczbowymi. Każdej cyfrze szesnastkowej odpowiadają cztery cyfry binarne. Wszystkie cyfry podanej liczby szesnastkowej mają wartość większą od 9. Podzielmy liczbę binarną na 4-cyfrowe grupy: 1010 1011 0100 1101. Można zauważyć, że cyfrze szesnastkowej C odpowiada zbyt niska wartość binarna: 0100, zatem liczby te mają różną wartość. Zdanie 1 jest więc **fałszywe**.

Krok 2

Liczby te można porównać za pośrednictwem systemu binarnego. Jednej cyfrze szesnastkowej odpowiadają cztery cyfry binarne, a cyfrze ósemkowej – trzy cyfry binarne. $A0B2_{16}$ to 1010 0000 1011 0010 binarnie, a 120262_8 to 001 010 000 010 110 010 binarnie. Aby sprawdzić, czy jest to ta sama liczba, zapisujemy jedną nad drugą, uwzględniając zera nieznaczące:

```
  1010000010110010
001010000010110010
```

Otrzymujemy taki sam zapis binarny obu liczb. Zdanie 2 jest więc **prawdziwe**.

Krok 3

Systemy dziewiątkowy i trójkowy są ze sobą powiązane, ponieważ $9 = 3^2$. Jednej cyfrze systemu dziewiątkowego odpowiadają dwie cyfry systemu trójkowego (0 – 00, 1 – 01, 2 – 02, 3 – 10, 4 – 11, 5 – 12, 6 – 20, 7 – 21, 8 – 22). 16_9 to 01 20 trójkowo, czyli po pominięciu zera nieznaczącego: 120_3. Zdanie 3 jest więc **fałszywe**.

Krok 4

Liczby te można porównać za pośrednictwem systemu binarnego. Jednej cyfrze ósemkowej odpowiadają trzy cyfry binarne, a cyfrze czwórkowej – dwie cyfry binarne. 127_8 to 001 010 111 binarnie, a 1123_4 to 01 01 10 11 binarnie. Otrzymujemy dwie różne liczby binarne:

```
001010111
  01011011
```

Zdanie 4 jest więc **prawdziwe**.

Odpowiedź: FPFP

Zadanie 1.2 (0–1)

Alfabet łaciński liczy 26 liter. Do zakodowania małych i wielkich liter tego alfabetu można użyć:

1.	5 bitów.	P	F
2.	6 bitów.	P	F
3.	7 bitów.	P	F
4.	1 bajta.	P	F

Rozwiązanie

Tagi: reprezentacja informacji w komputerze, system binarny, kody ASCII

Krok 1

Na 5 bitach można zapisać $2^5 = 32$ różnych informacji, a więc 5 bitów to za mało do reprezentacji łącznie 52 znaków. Zdanie 1 jest więc **fałszywe**.

Krok 2

Na 6 bitach można zapisać $2^6 = 64$ różnych informacji, a więc jest to wystarczająca liczba bitów do reprezentacji łącznie 52 znaków. Zdanie 2 jest zatem **prawdziwe**.

Krok 3

Na 7 bitach można zapisać $2^7 = 128$ różnych informacji, a więc jest to wystarczająca liczba bitów do reprezentacji łącznie 52 znaków. Podstawowy kod ASCII, w którym są reprezentowane m.in. małe i wielkie litery alfabetu łacińskiego, jest kodem 7-bitowym. Zdanie 3 jest więc **prawdziwe**.

Krok 4

Jeden bajt to 8 bitów, na których można zapisać $2^8 = 256$ różnych informacji, a więc jest to wystarczająca liczba bitów do reprezentacji łącznie 52 znaków. Rozszerzony kod ASCII jest kodem 8-bitowym. Zdanie 4 jest więc **prawdziwe**.

Odpowiedź: FPPP

Zadanie 2

Liczba palindromiczna to liczba całkowita nieujemna, która składa się z tych samych cyfr, gdy zapisujemy ją od lewej do prawej i od prawej do lewej. Liczba pierwsza to liczba całkowita dodatnia, która ma dokładnie dwa różne dzielniki: 1 i samą siebie. Palindromiczna liczba pierwsza to liczba całkowita dodatnia, która jest zarówno pierwsza, jak i palindromiczna.

Zadanie 2.1 (0–1)

Uzupełnij poniższą tabelę, wpisując w drugiej kolumnie TAK lub NIE.

Liczba	Palindromiczna liczba pierwsza
11	TAK
101	
323	
353	
169 691	
311 113	

Rozwiązanie

Tagi: liczby pierwsze, liczby złożone, palindromy, własności liczb, podzielność liczb, reszta z dzielenia

Krok 1

Wszystkie liczby w tabeli poza liczbą 169 691 są liczbami palindromicznymi.

Krok 2

Sprawdzamy, które z pozostałych liczb są również liczbami pierwszymi. Wystarczy sprawdzać podzielność przez liczby pierwsze mniejsze lub równe pierwiastkowi z danej liczby. Dla liczby 101 są to 3, 5 i 7 – przez żadną z nich 101 się nie dzieli, więc jest palindromiczną liczbą pierwszą.

Krok 3

Dla liczby 323 sprawdzamy dodatkowo podzielność przez 11, 13 i 17 – liczba jest podzielna przez 17 ($323 = 17 \cdot 19$). Nie jest więc liczbą pierwszą. Podzielność liczby 353 należy sprawdzić dla tych samych liczb – przez żadną z nich 353 się nie dzieli, więc jest palindromiczną liczbą pierwszą.

Krok 4

Liczba 311 113 nie jest liczbą pierwszą, ponieważ dzieli się przez 11. W tym przypadku nie trzeba wykonywać obliczeń, ponieważ każda liczba palindormiczna o parzystej liczbie cyfr jest podzielna przez 11.

Odpowiedź: Prawidłowo uzupełniona tabela:

Liczba	Palindromiczna liczba pierwsza
11	TAK
101	TAK
323	NIE
353	TAK
169 691	NIE
311 113	NIE

Informacja do zadań 2.2 i 2.3

Poniżej podano specyfikację problemu sprawdzenia, czy liczba jest palindromiczną liczbą pierwszą oraz algorytm rozwiązania tego problemu zapisany w pseudokodzie.

Specyfikacja

Dane: n – liczba całkowita dodatnia.

Wynik: napis „TAK", gdy n jest palindromiczną liczbą pierwszą, „NIE" – w przeciwnym przypadku.

```
jeśli Palindrom(n) oraz Pierwsza(n) to wypisz "TAK"
w przeciwnym przypadku wypisz "NIE"
```

Zadanie 2.2 (0–1)

Uzupełnij zapis funkcji `Palindrom`.

```
funkcja Palindrom(n)
    m ← n
    w ← 0
    dopóki m > 0 wykonuj
        w ← w * 10 + _______
        m ← _______
    zwróć n = w i zakończ
```

Rozwiązanie

Tagi: palindromy, analiza algorytmu, pseudokod, własności liczb, reszta z dzielenia, część całkowita, funkcje

Należy porównać liczbę `n` z liczbą `w`, powstałą przez zapisanie od końca cyfr liczby `n`. W zmiennej `m` zapiszemy wartość zmiennej `n`. Wyznaczymy kolejne cyfry liczby `m` od prawej strony (reszta z dzielenia `m` przez 10) oraz dodamy je do aktualnej wartości zmiennej `w` z uwzględnieniem pozycji cyfry w liczbie. Następnie usuniemy te cyfry ze zmiennej `m` (część całkowita z dzielenia `m` przez 10).

Odpowiedź: Uzupełniony zapis algorytmu w pseudokodzie:

```
funkcja Palindrom(n)
    m ← n
    w ← 0
    dopóki m > 0 wykonuj
        w ← w * 10 + m mod 10
        m ← m div 10
    zwróć n = w i zakończ
```

Zadanie 2.3 (0–2)

Uzupełnij zapis funkcji `Pierwsza` tak, aby wykonywała jak najmniej operacji.

```
funkcja Pierwsza(n)
    jeśli n = 2 to zwróć prawda i zakończ
    jeśli n = 1 lub n mod 2 = 0 to zwróć fałsz i zakończ
    d ← 3
    dopóki _______ wykonuj
        jeśli n mod d = 0 to zwróć fałsz i zakończ
        d ← _______
    zwróć prawda i zakończ
```

Rozwiązanie

Tagi: liczby pierwsze, analiza algorytmu, pseudokod, własności liczb, efektywność algorytmu, funkcje

Po wyłączeniu sprawdzenia liczb parzystych wystarczy sprawdzać podzielność `n` przez liczby nieparzyste, których kwadrat jest mniejszy lub równy `n`.

Odpowiedź: Uzupełniony zapis algorytmu w pseudokodzie:

```
funkcja Pierwsza(n)
    jeśli n = 2 to zwróć prawda i zakończ
    jeśli n = 1 lub n mod 2 = 0 to zwróć fałsz i zakończ
    d ← 3
    dopóki d * d ≤ n wykonuj
        jeśli n mod d = 0 to zwróć fałsz i zakończ
        d ← d + 2
    zwróć prawda i zakończ
```

Zadanie 2.4 (0–2)

Liczba 11 jest jedyną palindromiczną liczbą pierwszą o parzystej liczbie cyfr. Wykorzystaj ten fakt i zapisz w wybranej przez siebie notacji algorytm sprawdzający, czy liczba całkowita dodatnia jest palindromiczną liczbą pierwszą. Algorytm powinien wykonywać jak najmniej operacji. Możesz korzystać z podstawowych działań arytmetycznych na liczbach całkowitych (+, −, *, **div**, **mod**). Nie możesz wykorzystywać operacji na napisach.

Rozwiązanie

Tagi: liczby pierwsze, palindromy, sposoby zapisu algorytmów, własności liczb, efektywność algorytmów

Krok 1

Z pierwszego zdania w treści zadania wynika, że palindromiczne liczby pierwsze mniejsze od 100 to 2, 3, 5, 7 oraz 11. Liczby te sprawdzimy na początku programu. Dzięki temu dalej możemy sprawdzać tylko liczby większe od 100. Wykorzystamy funkcję `Pierwsza` oraz zmodyfikowaną funkcję `Palindrom` o nazwie `NPpalindrom`. Jej wynikiem będzie wartość **prawda** tylko dla liczb palindromicznych o nieparzystej liczbie cyfr.

Krok 2

Stosujemy skrócone wyliczanie zdań logicznych: gdy wynikiem funkcji `NPpalindrom` będzie wartość **fałsz**, nie nastąpi wywołanie funkcji `Pierwsza`.

Odpowiedź: Zapis w pseudokodzie algorytmu rozwiązania wraz z funkcją `NPpalindrom`:

```
funkcja NPpalindrom(n)
    m ← n
    w ← 0
    ile ← 0
    dopóki m > 0 wykonuj
        w ← w * 10 + m mod 10
        m ← m div 10
        ile ← ile + 1
    jeśli ile mod 2 = 0 to zwróć fałsz i zakończ
    zwróć n = w i zakończ

jeśli n = 2 lub n = 3 lub n = 5 lub n = 7 lub n = 11 to
    wypisz "TAK" i zakończ
jeśli n ≤ 100 to wypisz "NIE" i zakończ
jeśli NPpalindrom(n) oraz Pierwsza(n) to wypisz "TAK"
w przeciwnym przypadku wypisz "NIE"
```

Trening

ROZWIĄŻ BEZ KOMPUTERA

Zadanie 1

Oceń, czy poniższe zdania są prawdziwe. Wybierz **P**, jeśli zdanie jest prawdziwe, albo **F** – jeśli zdanie jest fałszywe.

Zadanie 1.1 (0–1)

Suma liczb 10101011_2 i 11011_2 jest:

1.	liczbą parzystą.	P	F
2.	większa od 255_{10}.	P	F
3.	równa 3012_4.	P	F
4.	mniejsza od $C6_{16}$.	P	F

Odpowiedź zapisz na kartce lub w sposób wskazany przez nauczyciela.

Zadanie 1.2 (0–1)

Wykorzystując cztery bajty pamięci, można zapisać:

1.	liczbę ze znakiem $+2^{31}$.	P	F
2.	liczbę ze znakiem -2^{31}.	P	F
3.	2^{32} różnych informacji.	P	F
4.	liczbę bez znaku 2^{32}.	P	F

Odpowiedź zapisz na kartce lub w sposób wskazany przez nauczyciela.

Zadanie 2

Oto niepełna specyfikacja problemu oraz algorytm go rozwiązujący:

Specyfikacja

Dane: x, n – liczby całkowite dodatnie.

Wynik: _____

```
tmp ← x
y ← 1
dopóki n > 0 wykonuj
    jeśli n mod 3 > 0 to
        y ← y * tmp
        jeśli n mod 3 = 2 to y ← y * tmp
    n ← n div 3
    jeśli n > 0 to tmp ← tmp * tmp * tmp
wypisz y
```

Uwaga: **div** oznacza dzielenie całkowite, a **mod** – resztę z dzielenia całkowitego.

Zadanie 2.1 (0–1)

Uzupełnij poniższą tabelę.

x	n	y
3	2	
3	3	
2	10	

Odpowiedź zapisz na kartce lub w sposób wskazany przez nauczyciela.

Zadanie 2.2 (0–1)

Uzupełnij specyfikację problemu.

Specyfikacja

Dane: x, n – liczby całkowite dodatnie.

Wynik: _____

Odpowiedź zapisz na kartce lub w sposób wskazany przez nauczyciela.

Zadanie 2.3 (0–3)

Zapisz w wybranej przez siebie notacji (lista kroków, pseudokod, język programowania) algorytm rozwiązujący ten sam problem bez wykorzystania operacji dzielenia całkowitego i reszty z dzielenia przez 3. Postaraj się tak zapisać algorytm, aby wykonywał jak najmniej operacji mnożenia.

Odpowiedź zapisz na kartce lub w sposób wskazany przez nauczyciela.

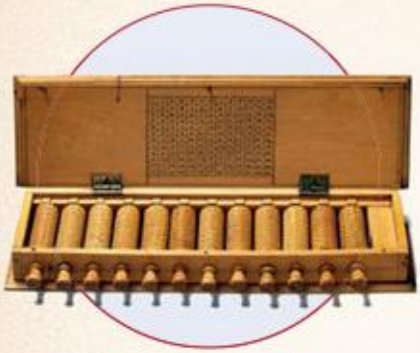
2 Rozwiązywanie problemów z wykorzystaniem struktur danych

9. Łamiemy szyfr Cezara
10. Poszukujemy liczby
11. Jak ocenić złożoność obliczeniową algorytmu?
12. Metody sortowania prostego
13. Szyfry przestawieniowe, anagramy
14. Sito Eratostenesa
15. Szukamy różnych podciągów
16. W poszukiwaniu lidera i idola

Wiesz, umiesz, zdasz

Kartka z historii

Algorytmika i programowanie



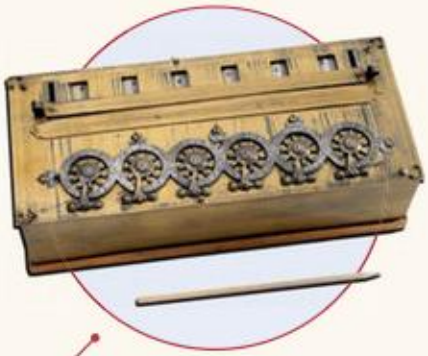
John Napier w 1614 r. opisał metodę logarytmów, czyli metodę mnożenia liczb za pomocą dodawania innych liczb, które można zapisać w tablicy. Napier wynalazł przyrząd, nazywany kostkami Napiera, który uznaje się za **prototyp komputera analogowego**. Kostki wymyślone przez szkockiego matematyka zdobyły dużą popularność – ich stosowanie rekomendował m.in. Izaak Newton.

1614 r.

1623 r.

Pierwszy **mechaniczny kalkulator** skonstruował prawdopodobnie niemiecki matematyk Wilhelm Schickard w 1623 r. Maszynę zamówił Jan Kepler do obliczeń astronomicznych – wykonywała cztery podstawowe działania na liczbach całkowitych. Choć drewniana konstrukcja spłonęła podczas wojny trzydziestoletniej, w 1957 r. zrekonstruowano ją na podstawie listów Schickarda do Keplera.

Francuski matematyk Blaise Pascal, aby pomóc ojcu w prowadzeniu rachunków, skonstruował maszynę liczącą, która pozwalała dodawać i odejmować. Ponadto była ona wyposażona w elementy pamięci mechanicznej. Maszynę, zaprezentowaną publicznie w 1645 r., nazywamy kalkulatorem Pascala lub **pascaliną**. Od nazwiska Francuza pochodzi także nazwa jednego z języków programowania.



1645 r.

1679 r.

W II połowie XVII w. Gottfried Wilhelm Leibniz opisał **dwójkowy system liczbowy**, w którym każdą liczbę można zapisać za pomocą kombinacji cyfr 0 i 1. Obecnie taki kod zero-jedynkowy (binarny) stanowi podstawę działania komputerów. Do licznych osiągnięć Leibniza należy m.in. skonstruowanie własnej maszyny liczącej, inspirowanej kalkulatorem Pascala.

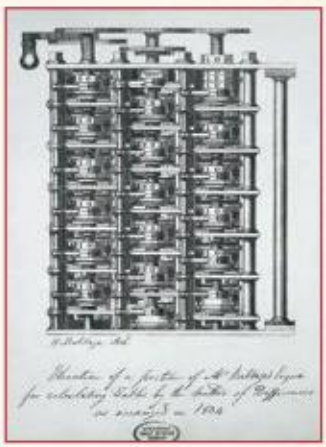
Na początku XIX w. Joseph Marie Jacquard zbudował **programowalną maszynę tkacką**, która automatycznie wykonywała wzór tkaniny zapisany na kartach perforowanych. Dzięki odpowiedniemu rozmieszczeniu otworów na takiej karcie można było sterować kolejnością wyboru nici w określonych kolorach. Aż do lat 80. XX w. karty i taśmy perforowane były nośnikami danych dla komputerów.

1804 r.

1817 r.

W 1817 r. polski zegarmistrz i wynalazca pochodzenia żydowskiego, Abraham Jakub Stern, skonstruował **machinę rachunkową** wykonującą pięć działań – oprócz czterech podstawowych działań arytmetycznych także pierwiastkowanie. Otrzymał wsparcie Stanisława Staszica oraz Towarzystwa Przyjaciół Nauk – dzięki rządowemu stypendium kontynuował swoje badania.

W 1837 r. Charles Babbage ogłosił koncepcję parowej **maszyny analitycznej**. Miała się ona składać z magazynu danych i pośrednich wyników obliczeń, młyna (jednostki liczącej) oraz mechanizmu sterującego, programowanego z użyciem kart perforowanych. Projekt nie doczekał się realizacji z powodu zbyt niskiego poziomu ówczesnej techniki.



1837 r.

1843 r.

Za **pierwszego programistę** niektórzy uważają Adę Lovelace. Jej notatki dotyczące maszyny analitycznej, wydane w 1843 r., zawierały diagram algorytmu napisany z zamiarem jego wykonania na fizycznej maszynie. W pracach Lovelace pojawiają się koncepcje: zmiennych, instrukcji warunkowej, pętli i funkcji. W 1980 r. Departament Obrony USA zatwierdził opis języka programowania nazwanego jej imieniem.

9. Łamiemy szyfr Cezara

Jedna z metod łamania szyfrów podstawieniowych – analiza częstości – została opisana po raz pierwszy już w IX w. przez arabskiego filozofa i matematyka Al-Kindiego. W tym temacie dowiesz się, na czym ona polega, oraz zastosujesz ją do odszyfrowania kryptogramu, który powstał po zaszyfrowaniu informacji jawnej szyfrem Cezara.

Cele lekcji

- Poznasz metody łamania szyfru Cezara.
- Zastosujesz analizę częstości do złamania szyfru Cezara.
- Poznasz przykład struktury danych – tablicę.
- Wykorzystasz algorytm zliczania częstości wystąpień znaków w tekście oraz algorytm znajdowania maksimum.

9.1. Metody łamania szyfru Cezara

Szyfr Cezara, s. 117–118

Jeśli chcemy poznać treść informacji jawnej zaszyfrowanej **szyfrem Cezara**, ale nie znamy klucza szyfrowania, możemy zastosować metodę prób i błędów. Należy wówczas szyfrować kryptogram szyfrem Cezara za pomocą kolejnych kluczy. Potencjalnych kluczy zazwyczaj nie jest wiele, np. dla alfabetu łacińskiego jest ich 25 (o 1 mniej niż liter alfabetu).

Dobra rada

Do wykonywania kolejnych prób deszyfrowania informacji możesz wykorzystać ten sam program, który służy do szyfrowania tekstu szyfrem Cezara. Natomiast to, czy odszyfrowany tekst jest zrozumiały, musisz już ocenić samodzielnie.

Ćwiczenie 1

Napisz na kartce krótki tekst złożony z wielkich liter alfabetu łacińskiego i zaszyfruj go szyfrem Cezara z wybranym kluczem. Następnie w parach spróbujcie odszyfrować nawzajem swoje kryptogramy.

Analiza częstości

Inną metodą stosowaną przy łamaniu szyfru Cezara jest **analiza częstości** wystąpień poszczególnych liter w szyfrogramie. Jeśli wiemy, w jakim języku jest informacja jawna, możemy założyć, że najczęściej występująca litera w szyfrogramie odpowiada najczęściej używanej literze w tym języku. Wówczas kluczem użytym do zaszyfrowania informacji jest różnica między pozycjami w alfabecie najczęściej występującej litery w szyfrogramie oraz najczęściej występującej litery w danym języku. W przypadku języka polskiego tą różnicą będzie pozycja najczęściej występującej litery w szyfrogramie (jeśli litery alfabetu numerujemy od 0), ponieważ w języku polskim najczęstsza jest litera A. Klucz deszyfrujący obliczymy, odejmując od 35 (liczby liter w alfabecie) pozycję najczęściej występującej litery w szyfrogramie. Warto podkreślić, że omawiana metoda nie zawsze jest skuteczna. Dla krótkich informacji analiza częstości może dać fałszywe wyniki.

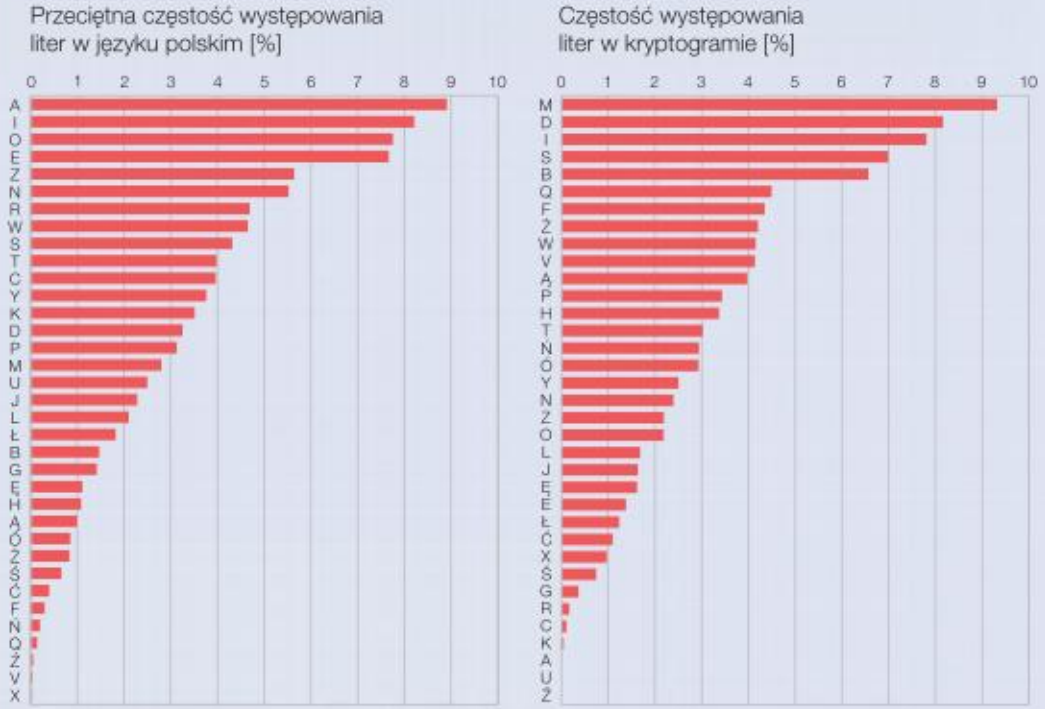
Analiza częstości w języku polskim

Pokażemy, jak wykorzystać analizę częstości do odszyfrowania kryptogramu, który powstał po zaszyfrowaniu informacji w języku polskim. Posłużymy się kilkustronicowym fragmentem reportażu podróżniczego *Zwierzęta, ludzie, bogowie* Ferdynanda Ossendowskiego.

Fragment kryptogramu

Tsžvdfdnef ts ńmońz hqmdfł b tsoszdqmd hs wflvsqmwńd, wtswyvbilóip hąp żąfłshbefą b ńspmqd fldyą. B żmiońe swyvsćqsxfme wńvdhdnef wmj hs hspz, bdzżdćąóip hżd swmshódqi ńsqmi, d tvbą wmshódfł ćsóqmivwńmi ńdvdęmqą.

Deszyfrujemy kryptogram



W kryptogramie najczęściej występuje litera M. Jej pozycja w alfabecie to 16 (jeśli pozycja A to 0), więc kluczem użytym do szyfrowania jest prawdopodobnie liczba 16, a kluczem deszyfrującym 19 (35 – 16 = 19). Próba deszyfrowania kluczem 19 się jednak nie powiedzie. Jeśli przeprowadzimy podobną analizę w odniesieniu do litery D – kolejnej najczęściej występującej w szyfrogramie – otrzymamy klucz deszyfrujący 30. Tym kluczem uda się odszyfrować informację.

Tekst jawny po odszyfrowaniu fragmentu kryptogramu

Powracając po kilku dniach z polowania do schroniska, spostrzegłem dym wychodzący z komina chaty. Z wielką ostrożnością skradając się do domu, zauważyłem dwa osiodłane konie, a przy siodłach żołnierskie karabiny.

Przykład pokazuje, że nie zawsze litera, która najczęściej występuje w szyfrogramie, pozwala ustalić klucz szyfrowania informacji jawnej. Dla uproszczenia przyjmiemy jednak, że reguła ta zawsze zachodzi.

Algorytm łamania szyfru Cezara

Zatem w **algorytmie łamania szyfru Cezara** uwzględnimy kroki:

1. Policzenie liczby wystąpień poszczególnych liter w szyfrogramie.
2. Znalezienie litery występującej w szyfrogramie najczęściej.
3. Wyznaczenie klucza, którym zaszyfrowano tekst.
4. Odszyfrowanie tekstu kluczem obliczonym ze wzoru:
   liczba liter alfabetu – klucz szyfrowania

Zapamiętaj

Analiza częstości występowania liter w szyfrogramie i w informacji jawnej pomaga odszyfrować tekst zaszyfrowany szyfrem podstawieniowym. Największe szanse na odszyfrowanie tekstu są wtedy, kiedy szyfrowany tekst jest dostatecznie długi.

9.2. Zliczamy wystąpienia liter w tekście

Żeby zliczyć wystąpienia poszczególnych liter, dla każdej litery potrzebna jest zmienna – licznik wystąpień. W przypadku alfabetu łacińskiego potrzeba takich zmiennych 26, dla alfabetu polskiego 35. Aby ułatwić sobie zadanie, skorzystamy z jednej ze struktur danych dostępnych w C++.

Struktura danych

Strukturę danych można określić jako pewnego rodzaju pojemnik na dane, który przechowuje i porządkuje je w określony sposób. Do zliczania wystąpień liter w tekście wykorzystamy strukturę danych zwaną **tablicą**. Pod jedną nazwą może ona przechowywać wiele wartości tego samego typu. O zapisywaniu wielu danych pod jedną nazwą mówiliśmy już przy okazji definiowania własnej **struktury**.

Tablica

Struktura, s. 109

Tablica, podobnie jak wykorzystywana wcześniej struktura, jest przykładem zmiennej **typu złożonego**. W języku C++ wyróżniamy także **typy proste**, które są predefiniowane w języku programowania, a ich nazwy są słowami kluczowymi, np. `int`, `char`, `bool`.

Typ złożony

Typ prosty

Przy **deklarowaniu tablicy** należy podać typ elementów tablicy, jej nazwę i liczbę elementów. Ogólna deklaracja tablicy ma postać:

Deklaracja tablicy

```
typ_elementów nazwa_tablicy[liczba_elementów];
```

Indeks, s. 38

Do elementu tablicy odwołujemy się za pomocą **indeksu** – tak jak w przypadku odwołania do znaku w napisie. Indeks elementu podajemy w nawiasach kwadratowych:

```
nazwa_tablicy[indeks]
```

W języku C++ tablice są zawsze indeksowane kolejnymi liczbami całkowitymi, zaczynając od 0. Tego typu tablicę nazywamy **tablicą jednowymiarową**. W temacie 16 omówimy **tablice dwuwymiarowe**.

Tablica jednowymiarowa

Tablica dwuwymiarowa, s. 219

Policzymy teraz wystąpienia liter w napisie złożonym z wielkich liter alfabetu łacińskiego i zapiszemy wyniki w tablicy `Liczniki`. Poszczególne elementy tablicy będą odpowiadały liczbie wystąpień kolejnych liter w napisie. Element `Licznik[i]`, gdzie $0 \leqslant i < 26$, określa liczbę wystąpień `i`-tej litery alfabetu. Oto specyfikacja problemu:

Specyfikacja

Dane: `napis` – napis złożony z wielkich liter alfabetu łacińskiego.

Wynik: `Liczniki` – tablica z liczbą wystąpień poszczególnych liter alfabetu łacińskiego w napisie.

Zapis algorytmu w pseudokodzie może być następujący:

```
dla i ← 0, 1, ..., 25 wykonuj Liczniki[i] ← 0
dla i ← 0, 1, ..., długość napisu - 1 wykonuj
    indeks ← kod_ASCII(napis[i]) - kod_ASCII('A')
    Liczniki[indeks] ← Liczniki[indeks] + 1
```

W pierwszej pętli nadajemy wartości początkowe elementom tablicy – zerujemy liczniki poszczególnych liter. Druga pętla przegląda poszczególne litery napisu. Ponieważ element tablicy o indeksie 0 odpowiada literze A, element o indeksie 1 – literze B itd., od kodu ASCII danej litery odejmujemy kod ASCII litery A, żeby otrzymać indeks elementu tablicy z zakresu od 0 do 25. Następnie powiększamy o 1 odpowiedni element tablicy. Oto kod źródłowy programu:

Kod źródłowy programu zliczającego wystąpienia liter w napisie złożonym z wielkich liter alfabetu łacińskiego

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    int i;
    string napis;
    cout<<"Podaj napis: "; cin>>napis;
    int Liczniki[26];
    for (i=0;i<26;i++)
        Liczniki[i]=0;
    for (i=0;i<napis.size();i++)
        Liczniki[napis[i]-'A']++;
    for (i=0;i<26;i++)
        if (Liczniki[i]>0)
        {
            cout<<char(i+'A')<<" - ";
            cout<<Liczniki[i]<<endl;
        }
    return 0;
}
```

Dobra rada

Podczas pisania programów uważaj, aby nie odwoływać się do nieistniejących elementów tablicy. Próba przypisania wartości takiemu elementowi jest odwołaniem do innego obszaru pamięci i może spowodować niespodziewane efekty, np. uszkodzić dane lub zawiesić program.

Warto wiedzieć

Wartości początkowe elementów tablicy można określić przy deklaracji tablicy. Na przykład nadanie wszystkim 26 elementom tablicy `Liczniki` wartości 0 będzie wyglądało następująco:

```
int Liczniki[26]={0};
```

Warto wiedzieć

Analiza częstości występowania liter w szyfrogramie może się nie sprawdzić, jeśli tekst jawny jest lipogramem, tzn. celowo napisano go tak, aby nie występowała w nim określona litera. Metoda ta nie zadziała także, jeśli w tekście jawnym użyto dużo słów zawierających rzadkie litery.

W linii 11 zadeklarowana została tablica `Liczniki` o 26 elementach typu całkowitego. W liniach 12–13 elementom tablicy zostały nadane wartości początkowe – każdemu z nich wartość 0.

Ponieważ znaki w języku C++ reprezentowane są przez ich kody ASCII, nie jest potrzebna konwersja znaku na liczbę. Należy jedynie pamiętać o przesunięciu indeksu tak, aby indeksowanie rozpoczynało się od zera (`napis[i]-'A'`). Element tablicy powiększamy o 1, korzystając z operatora `++` (linia 15). Pętla w liniach 16–21 odpowiada za wypisanie liczby wystąpień poszczególnych liter. Wyświetlane są tylko te litery, które występują w napisie – realizuje to instrukcja warunkowa w liniach 17–21. Wypisywana jest litera (konwersja kodu znaku na literę: `char(i+'A')`) oraz liczba jej wystąpień w napisie (linie 19–20).

Rysunek 9.1 przedstawia przykład wywołania programu.

```
Podaj napis: MATEMATYKA
A - 3
E - 1
K - 1
M - 2
T - 2
Y - 1
```

Rys. 9.1. Przykład wywołania programu zliczającego wystąpienia liter w napisie składającym się z wielkich liter alfabetu łacińskiego

Zmodyfikujemy teraz program tak, aby odczytywał dane z pliku tekstowego. W pliku mogą występować dowolne znaki. Program zliczy tylko litery alfabetu łacińskiego, przy czym nie będzie rozróżniał wielkości liter, np. dla jednej małej litery „a" i dwóch wielkich liter „A" zwróci wynik `A - 3`.

A to ciekawe

Kombinacje łatwe do przewidzenia

Podczas odszyfrowywania tekstu może się przydać nie tylko analiza częstości pojedynczych liter, lecz także znajomość często występujących obok siebie grup dwóch lub trzech znaków, czyli tzw. digramów i trigramów. Na przykład w języku polskim najczęstszymi digramami są „ch", „cz", „sz" i „rz", a trigramem – słowo „się". W języku angielskim z kolei często pojawia się słowo „the". Poszukiwanie powtarzających się par i trójek liter jest pomocne np. przy łamaniu szyfrów, w których podstawia się parę znaków zamiast jednego znaku.

Oto kod źródłowy zmodyfikowanego programu:

Kod źródłowy programu zliczającego litery alfabetu łacińskiego występujące w pliku tekstowym

```
1.  #include <iostream>
2.  #include <fstream>
3.  #include <string>
4.  #include <cctype>
5.
6.  using namespace std;
7.
8.  int main()
9.  {
10.     int i;
11.     string napis;
12.     char ch;
13.     int Liczniki[26];
14.     for (i=0;i<26;i++) Liczniki[i]=0;
15.     ifstream wejscie("dane_zliczanie.txt");
16.     while (!wejscie.eof())
17.     {
18.         getline(wejscie,napis);
19.         for (i=0;i<napis.size();i++)
20.         {
21.             ch=toupper(napis[i]);
22.             if (ch>='A' && ch<='Z') Liczniki[ch-'A']++;
23.         }
24.     }
25.     wejscie.close();
26.     for (i=0;i<26;i++)
27.         if (Liczniki[i]>0)
28.         {
29.             cout<<char(i+'A')<<" - ";
30.             cout<<Liczniki[i]<<endl;
31.         }
32.     return 0;
33. }
```

W linii 15 deklarujemy zmienną plikową `wejscie` typu `ifstream` oraz otwieramy plik z danymi skojarzony z tą zmienną. O **obsłudze plików tekstowych** pisaliśmy w temacie 8. W głównej pętli programu (linie 16–24) odczytywane są i przetwarzane kolejne wiersze z pliku. Wczytanie wiersza danych odbywa się za pomocą funkcji `getline` (linia 18), ponieważ w wierszu mogą występować spacje. Pętla w liniach 19–23 przetwarza jeden wiersz danych, tak jak to robił poprzedni program. Ponieważ w tekście mogą wystąpić też małe litery i inne znaki, najpierw ewentualna mała litera jest zamieniana na wielką (linia 21). Potem powiększany jest licznik – pod warunkiem, że znak był literą (linia 22). Po głównej pętli plik jest zamykany (linia 25). Pętla w liniach 26–31 wypisuje litery, które wystąpiły w tekście przynajmniej raz, oraz podaje liczbę ich wystąpień.

Obsługa plików tekstowych, s. 122–123

Ćwiczenie 2

Napisz program, który wczyta dane z pliku tekstowego otrzymanego od nauczyciela (np. *zliczanie.txt*), a następnie zliczy wystąpienia liter pełnego polskiego alfabetu i zapisze wyniki do pliku tekstowego. Porównaj swoje wyniki ze statystyką częstości liter w języku polskim.

Zapamiętaj

Tablica to struktura danych, w której pod jedną nazwą można przechowywać wiele wartości tego samego typu. Tablicę deklarujemy, podając typ elementów tablicy, nazwę tablicy, a po nazwie w nawiasach kwadratowych liczbę elementów. Poszczególne elementy nie mają swoich własnych nazw – odwołujemy się do nich za pośrednictwem indeksów. W języku C++ tablice są indeksowane kolejnymi liczbami całkowitymi, zaczynając od 0.

9.3. Znajdujemy maksimum

Program zliczający wystąpienia liter w tekście zmodyfikujemy tak, aby wypisał literę, która pojawia się w podanym tekście najczęściej. W tym celu wśród elementów tablicy należy znaleźć największą liczbę, a dokładniej wyznaczyć indeks elementu będącego maksimum tablicy. Oto algorytm poszukiwania indeksu maksimum w tablicy zapisany w pseudokodzie. Poszukiwany indeks tablicy zapamiętamy w zmiennej `indeks_m`.

```
indeks_m ← 0
dla i ← 1, 2, ..., 25 wykonuj
    jeśli Liczniki[i] > Liczniki[indeks_m] to indeks_m ← i
```

Ponieważ wartością początkową indeksu maksimum jest 0 (pozycja pierwszego elementu tablicy), pętla może rozpocząć przeglądanie tablicy od następnego elementu.

Fragment programu znajdującego i wypisującego najczęściej występującą literę może wyglądać tak jak poniżej.

Fragment kodu źródłowego programu znajdującego i wypisującego najczęściej występującą literę w tekście – znalezienie największego elementu tablicy i wypisanie go na ekranie

```
1.  for (i=1;i<26;i++)
2.      if (Liczniki[i]>Liczniki[indeks_m])
3.          indeks_m=i;
4.  cout<<"Najczesciej wystepujaca litera: ";
5.  cout<<char(indeks_m+'A');
```

Zwróć uwagę, że jeżeli najczęściej występujących liter jest kilka, to zostanie wypisana tylko jedna z nich – pierwsza w kolejności alfabetycznej.

Ćwiczenie 3

Napisz program, który wypisze literę najczęściej występującą w tekście złożonym z liter alfabetu łacińskiego. Jeśli takich liter jest więcej niż jedna, program powinien wypisać wszystkie.

9.4. Wyznaczamy klucz szyfrowania i odszyfrowujemy tekst

Zapiszmy teraz funkcję ZnajdzKlucz, która pobierze zaszyfrowany tekst z pliku *t_zaszyfrowany.txt*, następnie zliczy wystąpienia liter w szyfrogramie oraz wyznaczy klucz użyty do zaszyfrowania tekstu szyfrem Cezara. Tekst w pliku może zawierać polskie znaki diakrytyczne, dlatego należy pamiętać o tym, aby plik był zapisany przy użyciu właściwej strony kodowej. Przyjmiemy, że kluczem szyfrowania będzie pozycja (indeks) najczęściej występującej litery w szyfrogramie, ponieważ litera A statystycznie jest najczęściej używaną literą w języku polskim.

Funkcja ZnajdzKlucz implementuje algorytmy zliczania liter i znajdowania maksimum omówione w tym temacie oraz odwołuje się do treści z poprzedniego tematu (operacja wyszukiwania znaków w napisach – sprawdzenie, czy znak jest literą polskiego alfabetu).

```
1.  int ZnajdzKlucz()
2.  {
3.      int Liczniki[35];
4.      for (int i=0;i<35;i++) Liczniki[i]=0;
5.      string s;
6.      ifstream wejscie("t_zaszyfrowany.txt");
7.      while (!wejscie.eof())
8.      {
9.          getline(wejscie,s);
10.         for (int i=0;i<s.size();i++)
11.         {
12.             int j=alfabet_m.find(s[i]);
13.             if (j>=0 && j<35) Liczniki[j]++;
14.             else
15.             {
16.                 int j=alfabet_w.find(s[i]);
17.                 if (j>=0 && j<35) Liczniki[j]++;
18.             }
19.         }
20.     }
21.     wejscie.close();
22.     int klucz=0;
23.     for (int i=1;i<35;i++)
24.         if (Liczniki[i]>Liczniki[klucz]) klucz=i;
25.     return klucz;
26. }
```

Fragment kodu źródłowego programu deszyfrującego kryptogram, który powstał po zaszyfrowaniu tekstu szyfrem Cezara – funkcja znajdująca prawdopodobny klucz, którym zostały zaszyfrowane dane

W liniach 3–4 deklarowana i inicjowana jest tablica służąca do zliczania liter pełnego polskiego alfabetu (35 znaków). W linii 6 otwierany jest plik z danymi *t_zaszyfrowany.txt*. Pętla w liniach 7–20 przegląda kolejne wiersze pliku. Odczytuje linię tekstu (linia 9) i przegląda jej kolejne znaki (pętla w liniach 10–19). Najpierw sprawdza, czy dany znak jest małą literą alfabetu polskiego, i jeśli jest – powiększa odpowiedni licznik (linie 12–13). W przeciwnym przypadku sprawdza, czy znak jest wielką literą, i ewentualnie powiększa licznik (linie 16–17). Stałe tekstowe `alfabet_m` i `alfabet_w` reprezentują alfabet złożony odpowiednio z małych i wielkich liter alfabetu polskiego, tak jak w przykładach w poprzednim temacie. W liniach 23–24 znajdowany jest indeks maksimum, który jest potencjalnym kluczem szyfrowania, czyli wynikiem działania funkcji (linia 25).

Żeby odszyfrować szyfrogram, wystarczy go zaszyfrować z kluczem wyliczonym ze wzoru: `35 - znaleziony klucz`. Kod funkcji szyfrującej oraz funkcji `main` może wyglądać następująco:

Fragment kodu źródłowego programu deszyfrującego kryptogram, który powstał po zaszyfrowaniu tekstu szyfrem Cezara – funkcje `Szyfruj` i `main`

```
1.  void Szyfruj(int klucz)
2.  {
3.      string s;
4.      ifstream wejscie("t_zaszyfrowany.txt");
5.      ofstream wyjscie("t_odszyfrowany.txt");
6.      while (!wejscie.eof())
7.          {
8.              getline(wejscie,s);
9.              wyjscie<<Cezar(s,klucz)<<endl;
10.         }
11.     wejscie.close(); wyjscie.close();
12. }
13.
14. int main()
15. {
16.     Szyfruj(35-ZnajdzKlucz());
17.     cout<<"Plik t_odszyfrowany.txt zostal utworzony!";
18.     return 0;
19. }
```

Funkcja `Cezar` oraz użyta w niej funkcja `Cezar_PL`, które należy wykorzystać w programie, są takie same jak na s. 125 i 126.

Ćwiczenie 4

Napisz program umożliwiający łamanie szyfru Cezara, uwzględniający kroki algorytmu wyróżnione na s. 142. Program powinien pobierać szyfrogram z pliku i zapisywać tekst będący próbą odszyfrowania kryptogramu również w pliku. Do sprawdzenia działania programu wykorzystaj plik otrzymany od nauczyciela (np. *tajne.txt*).

Podsumowanie

- Szyfr Cezara i inne proste szyfry podstawieniowe można złamać metodą analizy częstości.
- Tablica to struktura danych, w której pod jedną nazwą można zapisać wiele elementów tego samego typu. Do elementu tablicy odwołujemy się za pośrednictwem indeksu.
- Tablicę deklarujemy, podając typ elementów tablicy, nazwę tablicy, a po nazwie w nawiasach kwadratowych liczbę elementów.
- W języku C++ tablice zawsze są indeksowane kolejnymi liczbami całkowitymi, zaczynając od zera.
- Podczas pisania programu z wykorzystaniem tablicy należy pamiętać o nadaniu wartości początkowych wszystkim elementom tablicy.
- Kiedy poszukujemy danego elementu w tablicy (np. maksimum), w zależności od potrzeb możemy wyszukiwać albo wartość tego elementu, albo jego położenie w tablicy, czyli indeks.
- W języku polskim najczęściej występującą literą jest litera A. Dlatego kluczem użytym do zaszyfrowania informacji szyfrem Cezara jest prawdopodobnie pozycja w alfabecie tej litery, która najczęściej występuje w szyfrogramie (jeśli litery alfabetu numerujemy od 0).
- Ponieważ szyfr Cezara ma niewiele potencjalnych kluczy, inną metodą złamania go jest próbowanie odszyfrowania tekstu za pomocą kolejnych kluczy.

Zadania

* **1** Napisz program, który wczyta liczbę całkowitą większą od zera i wypisze liczbę wystąpień poszczególnych cyfr w tej liczbie.

* **2** Napisz program, który będzie wczytywał z klawiatury liczby całkowite dodatnie do napotkania zera i wypisywał największą z podanych liczb.

* **3** Napisz program deszyfrujący krótki tekst zakodowany szyfrem Cezara, wczytany z klawiatury. Po każdej próbie deszyfrowania program zapyta, czy wypisany wynik jest poprawny. Jeśli uda się odszyfrować tekst, program zakończy działanie, w przeciwnym przypadku – spróbuje odszyfrować kryptogram kolejnym kluczem.

** **4** Sprawdź w internecie częstość występowania poszczególnych liter w języku angielskim. Omówiony w tym temacie program łamania szyfru Cezara zmodyfikuj tak, aby deszyfrował teksty w języku angielskim. Sprawdź działanie programu.

** **5** Napisz program, który będzie wczytywał z klawiatury liczby całkowite dodatnie do napotkania zera i wypisywał liczbę wystąpień poszczególnych cyfr we wszystkich wczytanych liczbach łącznie.

** **6** Przygotuj plik tekstowy z dowolnym tekstem składającym się ze słów o długości nie większej niż 10 liter i ze spacji oddzielających wyrazy. Napisz program, który odczyta dane z pliku wejściowego i wypisze, ile jest w nim wyrazów o poszczególnych długościach (od 1 do 10).

** **7** Przygotuj plik tekstowy z dowolnym tekstem zawierającym tylko słowa i oddzielające je spacje, bez znaków interpunkcyjnych. Napisz program, który odczyta dane z pliku wejściowego i wypisze, ile wyrazów rozpoczyna się na poszczególne litery.

*** **8** Napisz program, który pomoże złamać szyfr permutacyjny, zgodnie z poniższą specyfikacją. Plik potrzebny do rozwiązania zadania przekaże ci nauczyciel.

Specyfikacja

Dane: *szyfrogram_perm.txt* – plik zawierający tekst złożony z wielkich liter alfabetu łacińskiego i znaków oddzielających, zaszyfrowany szyfrem permutacyjnym.

Wynik: `klucz` – napis określający permutację wielkich liter alfabetu łacińskiego, czyli klucz, którym prawdopodobnie został zaszyfrowany tekst w pliku *szyfrogram_perm.txt*,
jawny_perm.txt – plik zawierający tekst z pliku *szyfrogram_perm.txt* odszyfrowany kluczem `klucz`.

*** **9** Napisz, zgodnie z poniższą specyfikacją, program wspomagający złamanie szyfru Vigenère'a, kiedy znamy długość klucza użytego do szyfrowania. Plik potrzebny do rozwiązania zadania oraz długość klucza przekaże ci nauczyciel.

Specyfikacja

Dane: *szyfrogram_Vigenere.txt* – plik zawierający tekst złożony z wielkich liter alfabetu łacińskiego, zaszyfrowany szyfrem Vigenère'a,
`dl_klucza` – długość klucza użytego do szyfrowania, $0 <$ `dl_klucza` < 5.

Wynik: *jawny_Vigenere.txt* – plik, który zawiera odszyfrowany tekst z pliku *szyfrogram_Vigenere.txt*.

10. Poszukujemy liczby

Na pewno zdarzyło ci się kiedyś szukać konkretnej rzeczy w jakimś zbiorze, np. książki na regale lub ubrania w szafie. Dużo łatwiej to zrobić, gdy zbiór jest uporządkowany zgodnie z pewną zasadą, np. książki według alfabetu, a ubrania według kategorii. Możesz wtedy zawęzić poszukiwania np. do litery P albo do spodni, zamiast przeglądać każdą rzecz po kolei. Czasami jednak mamy do czynienia ze zbiorami, których elementy nie są uporządkowane, lub nie wiemy, czy są uporządkowane. Dlatego w tym temacie zajmiemy się wyszukiwaniem liczb w zbiorach zarówno uporządkowanych, jak i nieuporządkowanych.

Cele lekcji

- Wyszukasz element w zbiorze nieuporządkowanym i uporządkowanym.
- Zastosujesz algorytmy przeszukiwania liniowego i binarnego.
- Poznasz kolejne sposoby przekazania parametru do funkcji – przez wskaźnik oraz przez referencję.
- Znajdziesz jednocześnie minimum i maksimum w zbiorze.

10.1. Sformułowanie problemu wyszukiwania

W programach wyszukujących określony element w zbiorze danymi wejściowymi będzie ciąg n losowych liczb całkowitych: $a_0, a_1, \ldots, a_{n-1}$. Liczby zapamiętamy w **tablicy**.

Tablica, s. 142

Funkcja `main` w programach, które napiszemy (niezależnie od zastosowanej metody wyszukiwania), powinna uwzględniać następujące etapy:

1. Wylosowanie elementów tablicy.
2. Wypisanie elementów tablicy.
3. Wyszukanie elementu w tablicy.
4. Wypisanie wyniku wyszukiwania.

Aby zrealizować dwa pierwsze etapy, zdefiniujemy funkcje `Losuj` i `Wypisz`. Pierwsza z nich wylosuje elementy tablicy, a druga wypisze wylosowane liczby.

Liczba elementów tablicy, czyli rozmiar tablicy, powinna być znana w momencie kompilacji programu. Dlatego rozmiar określimy za pomocą **stałej**, którą zdefiniujemy przed funkcjami. Jeśli chcielibyśmy sprawdzić działanie programu dla tablicy o innej liczbie elementów, wystarczy wówczas wprowadzić zmianę tylko w jednej linii kodu – w **definicji stałej** podać inny rozmiar tablicy.

Stała, s. 125

Definicja stałej, s. 125

Fragment kodu źródłowego programu wyszukującego element tablicy może wyglądać następująco. Fragment ten zawiera definicje stałej oraz funkcji losującej i wypisującej tablicę liczb.

Fragment kodu źródłowego programu wyszukującego element w tablicy – definicje stałej oraz funkcji losującej i wypisującej elementy tablicy

```
1.  #include <iostream>
2.  #include <cstdlib>
3.  #include <ctime>
4.
5.  using namespace std;
6.
7.  const int N=10;
8.
9.  void Losuj(int A[])
10. {
11.     for (int i=0;i<N;i++) A[i]=rand()%100;
12. }
13.
14. void Wypisz(int A[])
15. {
16.     for (int i=0;i<N;i++) cout<<A[i]<<" ";
17.     cout<<endl;
18. }
```

Warto wiedzieć

W środowisku Code::Blocks z kompilatorem GCC można określić rozmiar tablicy za pomocą zmiennej:

```
int n;
cout<<"Rozmiar tablicy: ";
cin>>n;
int A[n];
```

Nie jest to jednak rozwiązanie zgodne ze standardem języka C++ i może nie działać w innych środowiskach lub gdy użyje się innego kompilatora.

W linii 7 została zdefiniowana stała `N` o wartości `10`, określająca rozmiar tablicy. Parametrem każdej z funkcji `Losuj` i `Wypisz` jest tablica. Instrukcja w linii 11 wypełnia tablicę losowymi liczbami z zakresu od 0 do 99. Zakres losowania został ograniczony, żeby łatwo było zinterpretować działanie programu. Otrzymanie losowej liczby całkowitej nieujemnej umożliwia **funkcja** `rand`. Aby móc z niej skorzystać, trzeba dołączyć do programu bibliotekę `cstdlib`. Zakres losowania ograniczyliśmy, biorąc resztę z dzielenia wylosowanej liczby przez 100.

Funkcja `rand`

Przekazanie parametru przez wartość, s. 108
Zmienna lokalna, s. 108

Przypomnijmy, że **przekazanie parametru przez wartość** oznacza, że parametr funkcji jest chroniony, tzn. wszystkie operacje są wykonywane na kopii parametru, traktowanego jak **zmienna lokalna**. W przypadku, gdy parametrem funkcji jest tablica, w rzeczywistości nie jest nim sama tablica, ale jej adres – tablice w języku C++ są reprezentowane przez adresy (wskaźniki). Oznacza to, że wszystkie operacje są wykonywane na oryginalnej tablicy, a dokonane w niej zmiany zapamiętywane po zakończeniu działania funkcji. Taki sposób przekazania parametru nazywamy **przekazaniem parametru przez wskaźnik**. Tak więc wartości zwracane przez funkcję `Losuj` zostaną zachowane po zakończeniu jej działania.

Przekazanie parametru przez wskaźnik

Fragment kodu `main` wywołującej funkcje `Losuj` i `Wypisz` może wyglądać następująco:

Fragment kodu źródłowego programu wyszukującego element w tablicy – fragment funkcji `main`

```
1.  int A[N];
2.  srand(time(NULL));
3.  Losuj(A);
4.  Wypisz(A);
```

W linii 1 zadeklarowana jest tablica `N` liczb całkowitych. Instrukcja w linii 2 inicjuje generator liczb losowych i uzależnia losowanie liczb od czasu wskazywanego przez zegar komputera. Jeśli nie dodalibyśmy tej instrukcji, po każdym uruchomieniu programu wynikiem funkcji `Losuj` byłby ten sam ciąg liczb. Ustawienie punktu startowego dla mechanizmu losującego umożliwia **funkcja** `srand`. Z kolei dzięki **funkcji** `time` zwrócimy aktualny czas w postaci liczby całkowitej. Korzystanie z funkcji `srand` wymaga dołączenia biblioteki `cstdlib`, a korzystanie z funkcji `time` – biblioteki `ctime`.

Warto wiedzieć

Żeby wylosować liczbę z przedziału [*a*; *b*], należy użyć wyrażenia: `rand()%(b-a+1)+a`

Funkcja `srand`, **funkcja** `time`

Ćwiczenie 1

Napisz program losujący i wypisujący tablicę *n* liczb całkowitych nieujemnych, uwzględniający funkcje `Losuj` i `Wypisz`. Sprawdź działanie programu dla różnych wartości stałej określającej rozmiar tablicy.

Warto wiedzieć

Dzięki temu, że podczas przekazania parametru przez wskaźnik nie jest tworzona kopia tablicy, oszczędza się pamięć komputera.

Zapamiętaj

W języku C++ rozmiar tablicy określamy najczęściej za pomocą stałej. Jeśli tablica jest parametrem funkcji, to jest ona przekazywana przez wskaźnik (adres). Wszystkie operacje na takim parametrze są wykonywane na oryginalnej tablicy i zmiany są pamiętane po zakończeniu działania funkcji.

10.2. Poszukujemy liczby w zbiorze nieuporządkowanym

Jeżeli szukamy danego elementu w zbiorze i nie wiemy, czy zbiór jest uporządkowany, musimy sprawdzać każdy element po kolei. Jeśli pierwszy element nie jest równy szukanemu, sprawdzamy drugi itd. Czynności powtarzamy, dopóki nie odnajdziemy elementu lub nie przejrzymy wszystkich elementów zbioru. Taki sposób postępowania nazywamy **przeszukiwaniem liniowym** lub **sekwencyjnym**.

Przeszukiwanie liniowe (sekwencyjne)

Oto specyfikacja problemu wyszukiwania elementu w zbiorze nieuporządkowanym:

Specyfikacja

Dane: `A[0..n-1]` – tablica *n* liczb całkowitych,
x – liczba całkowita.

Wynik: wartość **`prawda`**, gdy liczba *x* znajduje się w tablicy, **`fałsz`** – w przeciwnym przypadku.

Możemy mieć szczęście i znaleźć szukany element już w pierwszym porównaniu. Jeśli elementu nie ma w tablicy lub znajduje się na ostatnim miejscu, musimy przejrzeć całą tablicę, czyli wykonać *n* porównań.

Warto wiedzieć

Przeszukiwanie liniowe stosuje się np. do znajdowania maksimum w tablicy.

Funkcję sprawdzającą, czy liczba x znajduje się w n-elementowej tablicy A, można zapisać w pseudokodzie następująco:

```
funkcja SzukajLin(A[],x)
    dla i ← 0, 1, ..., n – 1 wykonuj
        jeśli A[i] = x to zwróć prawda i zakończ
    zwróć fałsz i zakończ
```

Dobra rada

Możesz też ustalić, że jeśli szukana liczba jest w tablicy, wynikiem funkcji `SzukajLin` będzie pozycja tej liczby. Natomiast jeśli liczby nie ma w tablicy, wynikiem funkcji może być wartość n (indeks spoza zakresu).

Ćwiczenie 2

Napisz program, który wylosuje i wypisze tablicę liczb całkowitych, a następnie wczyta liczbę całkowitą i wypisze informację, czy ta liczba znajduje się w tablicy. Wyszukiwanie liczby powinna realizować funkcja `SzukajLin`, działająca zgodnie z algorytmem zapisanym w powyższym pseudokodzie.

Zapamiętaj

Jeśli nie wiemy, czy zbiór, w którym szukamy elementu, jest uporządkowany, stosujemy przeszukiwanie liniowe (sekwencyjne). Przeglądamy wówczas kolejne elementy zbioru tak długo, aż znajdziemy szukany element lub przejrzymy wszystkie elementy.

10.3. Poszukujemy liczby w zbiorze uporządkowanym

Gdy szukamy elementu w zbiorze uporządkowanym, możemy ograniczyć liczbę porównań. Zrobimy to poprzez stopniowe zawężanie zbioru, który przeszukujemy. Aby zilustrować sposób postępowania, rozważmy jedną z wersji gry w 20 pytań: pierwszy gracz wybiera liczbę całkowitą z określonego przedziału, natomiast drugi musi odgadnąć, jaką liczbę pomyślał ten pierwszy. Zgadujący może pytać, czy pomyślana liczba znajduje się w podanym przez niego przedziale.

Załóżmy, że liczba całkowita pomyślana przez pierwszego gracza znajduje się w przedziale $[a; b]$. Dla zgadującego najskuteczniejszą strategią będzie podzielenie tego przedziału na dwa równoliczne przedziały (z dokładnością do 1 dla nieparzystej długości przedziału). Najpierw wyznacza środek przedziału (c) ze wzoru $c = (a + b)$ **div** 2. Następnie pyta, czy szukana liczba znajduje się w przedziale $[a; c]$. Ponieważ poszukiwana liczba musi należeć do jednego z przedziałów $[a; c]$ lub $[c + 1; b]$, w zależności od odpowiedzi pierwszego gracza zgadujący odrzuca jeden z przedziałów, czyli połowę liczb. W kolejnych krokach postępuje podobnie. Gdy zawęzi przedział poszukiwań do jednej liczby, ta liczba jest rozwiązaniem.

Przeszukiwanie binarne

Opisany sposób wyszukiwania elementu nosi nazwę **przeszukiwania binarnego**. W kolejnych krokach zawężamy przeszukiwany przedział do połowy elementów.

Tabela 10.1 przedstawia przykład wyszukiwania w przedziale [1; 100] liczby pomyślanej przez pierwszego gracza (tu liczby 65).

Iteracja pętli	Początek przedziału (*a*)	Koniec przedziału (*b*)	Środek przedziału (*c*)	Przedział, o który pyta zgadujący	Udzielona odpowiedź
1	1	100	50	[1; 50]	Nie
2	51	100	75	[51; 75]	Tak
3	51	75	63	[51; 63]	Nie
4	64	75	69	[64; 69]	Tak
5	64	69	66	[64; 66]	Tak
6	64	66	65	[64; 65]	Tak
7	64	65	64	[64; 64]	Nie
–	65	65	–	–	–

Tabela 10.1. Przykład wyszukiwania liczby 65 w przedziale [1; 100]

Poszukiwanie elementu w uporządkowanej tablicy

Rozważmy teraz problem poszukiwania elementu w uporządkowanej tablicy liczb. W tym przypadku, kiedy zawężamy przedział poszukiwań, operujemy indeksami elementów tablicy, a nie wartościami elementów. Może się również zdarzyć, że szukanej liczby nie ma w tablicy. Oto specyfikacja problemu i zapis algorytmu w pseudokodzie:

Specyfikacja

Dane: `A[0..n-1]` – tablica *n* uporządkowanych niemalejąco liczb całkowitych,
x – liczba całkowita.

Wynik: wartość **prawda**, gdy liczba *x* znajduje się w tablicy, **fałsz** – w przeciwnym przypadku.

```
funkcja SzukajBin(A[],x)
    lewy ← 0
    prawy ← n - 1
    dopóki lewy < prawy wykonuj
        srodek ← (lewy + prawy) div 2
        jeśli A[srodek] < x to lewy ← srodek + 1
        w przeciwnym przypadku prawy ← srodek
    zwróć A[lewy] = x i zakończ
```

W zmiennych `lewy` i `prawy` przechowywane są indeksy początku i końca przeszukiwanego przedziału, a w zmiennej `srodek` – indeks środka tego przedziału. Przy wyznaczaniu środka przedziału nie sprawdzamy za każdym razem, czy jest on szukaną liczbą. Gdybyśmy chcieli tak robić, oznaczałoby to dodatkowe porównanie w każdej iteracji pętli.

Warto wiedzieć

Do określenia wartości funkcji nie trzeba używać instrukcji warunkowej:

```
jeśli A[lewy] = x to
  zwróć prawda i zakończ
w przeciwnym przypadku
  zwróć fałsz i zakończ
```

Nie uzyskalibyśmy znaczącej poprawy, a w przypadku nieznalezienia liczby byłoby wykonywanych dwa razy więcej porównań. Wynikiem funkcji jest wartość logiczna – wynik porównania `A[lewy] = x`.

Kod źródłowy funkcji `SzukajBin` może wyglądać następująco:

Fragment kodu źródłowego programu wyszukującego element w tablicy – funkcja realizująca algorytm przeszukiwania binarnego

```
1.  bool SzukajBin(int A[], int x)
2.  {
3.      int lewy=0, prawy=N-1, srodek;
4.      while (lewy<prawy)
5.      {
6.          srodek=(lewy+prawy)/2;
7.          if (A[srodek]<x) lewy=srodek+1;
8.          else prawy=srodek;
9.      }
10.     return (A[lewy]==x);
11. }
```

Tabele 10.2 oraz 10.3 ilustrują działanie algorytmu poszukiwania odpowiednio liczb 3 i 5 w tablicy, w której na kolejnych pozycjach znajdują się liczby: 1, 4, 5, 6, 9. W pierwszym przykładzie uzyskamy odpowiedź, że liczby 3 nie ma w tablicy (pętla wykona się trzy razy), a w drugim – że liczba 5 jest w tablicy (pętla wykona się dwa razy).

Iteracja pętli	Wartość zmiennej `lewy`	Wartość zmiennej `prawy`	Wartość zmiennej `srodek`	Element `A[srodek]`	Odpowiedź na pytanie, czy `A[srodek] < 3`
1	0	4	2	5	Nie
2	0	2	1	4	Nie
3	0	1	0	1	Tak
–	1	1	–	–	–

Tabela 10.2. Przykład wyszukiwania liczby 3 w tablicy z liczbami: 1, 4, 5, 6, 9

A to ciekawe

20 pytań sztucznej inteligencji

Gra w 20 pytań, polegająca na zgadywaniu, o jakim obiekcie pomyślała dana osoba, ma wiele wersji. W 1988 r. Robin Burgener na podstawie tej popularnej zabawy stworzył grę 20Q, która do odgadywania wykorzystuje sztuczną inteligencję. W 20Q można zagrać online. Według właścicieli gry sztuczna inteligencja zgaduje pomyślany przez użytkownika obiekt w 80% przypadków, a jeśli pozwolimy jej zadać 25 pytań, wygra aż 98% rozgrywek.

Iteracja pętli	Wartość zmiennej lewy	Wartość zmiennej prawy	Wartość zmiennej srodek	Element A[srodek]	Odpowiedź na pytanie, czy A[srodek] < 5
1	0	4	2	5	Nie
2	0	2	1	4	Tak
–	2	2	–	–	–

Tabela 10.3. Przykład wyszukiwania liczby 5 w tablicy z liczbami: 1, 4, 5, 6, 9

Parametrem funkcji SzukajBin jest uporządkowana niemalejąco tablica liczb. W związku z tym na potrzeby programu realizującego algorytm przeszukiwania binarnego zmodyfikujemy funkcję Losuj tak, aby losowała tak uporządkowaną tablicę liczb. Zakres losowania kolejnego elementu uzależnimy od wartości elementu poprzedniego – dodamy do niego wartość losową. Pierwszy element tablicy może być dowolną liczbą. Oto kod źródłowy zmodyfikowanej funkcji Losuj:

```
1.  void Losuj(int A[])
2.  {
3.      A[0]=rand()%20;
4.      for (int i=1;i<N;i++) A[i]=A[i-1]+rand()%20;
5.  }
```

Fragment kodu źródłowego programu wyszukującego element w tablicy – funkcja losująca uporządkowaną niemalejąco tablicę liczb

Kod źródłowy funkcji main programu poszukującego binarnie danego elementu w tablicy może być następujący:

```
1.  int main()
2.  {
3.      int x;
4.      int A[N];
5.      srand(time(NULL));
6.      Losuj(A);
7.      Wypisz(A);
8.      cout<<"Podaj liczbe do wyszukania: ";
9.      cin>>x;
10.     if (SzukajBin(A,x))
11.         cout<<"Liczba jest w tablicy";
12.     else
13.         cout<<"Liczby nie ma w tablicy";
14.     return 0;
15. }
```

Fragment kodu źródłowego programu wyszukującego binarnie dany element w tablicy – funkcja main

Ćwiczenie 3

Napisz program, który sprawdzi, czy w uporządkowanej tablicy *n* liczb całkowitych występuje podany element. Liczbę powinna wyszukiwać funkcja realizująca algorytm przeszukiwania binarnego.

Tabela 10.4 przedstawia liczbę powtórzeń pętli algorytmu realizującego przeszukiwanie binarne w zależności od liczby elementów w zbiorze.

Liczba elementów w zbiorze (n)	Liczba powtórzeń pętli (k)
10	4
16	4
100	7
128	7
1 000	10
1 024	10
10 000	14
100 000	17
1 000 000	20
1 048 576	20

Tabela 10.4. Liczba iteracji pętli w algorytmie przeszukiwania binarnego w zależności od liczby elementów zbioru

Warto wiedzieć

Algorytm przeszukiwania binarnego opiera się na metodzie „dziel i zwyciężaj", czyli podziale problemu na podproblemy tego samego typu dla mniejszej liczby danych. Więcej o metodzie „dziel i zwyciężaj" powiemy w temacie 20.

Liczba potrzebnych powtórzeń pętli rośnie bardzo powoli mimo znacznego wzrostu liczby elementów w zbiorze. Dla liczby elementów rzędu miliona potrzebnych jest tylko 20 powtórzeń. Gdyby wykorzystać przeszukiwanie liniowe, potrzebne byłoby średnio pół miliona powtórzeń.

Na podstawie tabeli 10.4 można wyznaczyć wzór opisujący zależność pomiędzy liczbą powtórzeń k a liczbą danych n w algorytmie przeszukiwania binarnego. Dla liczby danych będących potęgą dwójki liczba powtórzeń jest równa wykładnikowi potęgi:

$$n = 2^k$$

Zatem:

$$k = \log_2 n$$

Dla wartości n, które nie są potęgami liczby 2, wartość logarytmu o podstawie 2 nie będzie liczbą całkowitą. Liczba iteracji pętli jest wówczas równa wartości zaokrąglonej w górę. Zaokrąglenie w górę, nazywane często sufitem, oznaczamy .

$$k = \log_2 n$$

Zapamiętaj

Algorytm przeszukiwania binarnego służy do znajdowania elementu w zbiorze uporządkowanym. Jego działanie polega na podzieleniu przeszukiwanego zbioru na połowy i kontynowaniu poszukiwań w części, w której może występować szukany element. Czynności powtarza się tak długo, aż pozostanie jeden element – wówczas sprawdza się, czy to szukany wynik.

10.4. Jednoczesne znajdowanie minimum i maksimum

Czasami zachodzi potrzeba znalezienia w zbiorze danych zarówno elementu najmniejszego, jak i największego. Dzieje się tak na przykład wtedy, gdy chcemy poznać **rozpiętość zbioru**, czyli różnicę pomiędzy elementem największym (maksimum) i elementem najmniejszym (minimum).

● Rozpiętość zbioru

Oto specyfikacja problemu, którym się teraz zajmiemy:

Specyfikacja

Dane: `A[0..n-1]` – tablica n liczb całkowitych.

Wynik: `mini` – najmniejszy element w tablicy `A`,
`maks` – największy element w tablicy `A`.

Ponieważ nie wiemy, czy zbiór jest uporządkowany, narzucającym się rozwiązaniem jest zastosowanie algorytmu przeszukiwania liniowego: minimum i maksimum szukamy wtedy niezależnie. Zapis tego algorytmu w pseudokodzie może być następujący:

```
mini ← A[0]
dla i ← 1, 2, ..., n - 1 wykonuj
    jeśli A[i] < mini to mini ← A[i]
maks ← A[0]
dla i ← 1, 2, ..., n - 1 wykonuj
    jeśli A[i] > maks to maks ← A[i]
```

Obydwie instrukcje warunkowe można umieścić w jednej pętli, nie zmienia to jednak liczby porównań wykonywanych przez algorytm:

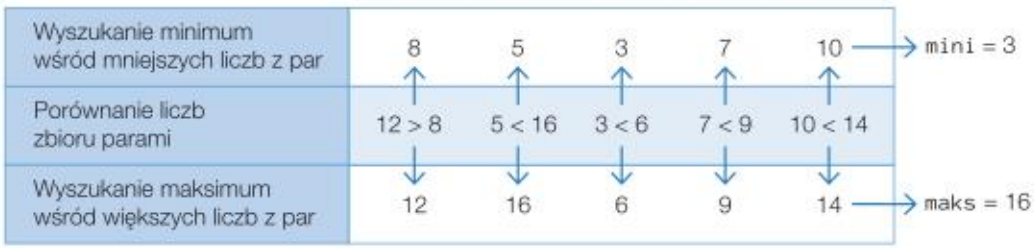
```
mini ← A[0]
maks ← A[0]
dla i ← 1, 2, ..., n - 1 wykonuj
    jeśli A[i] < mini to mini ← A[i]
    jeśli A[i] > maks to maks ← A[i]
```

Zauważ, że drugą instrukcję warunkową wystarczy wykonywać tylko wtedy, gdy nie była modyfikowana wartość zmiennej `mini`. Dlatego algorytm zmodyfikujemy następująco:

```
mini ← A[0]
maks ← A[0]
dla i ← 1, 2, ..., n - 1 wykonuj
    jeśli A[i] < mini to mini ← A[i]
    w przeciwnym przypadku
        jeśli A[i] > maks to maks ← A[i]
```

Jeśli zastosujemy ostatni algorytm, mamy szansę zmniejszyć liczbę porównań. W najgorszym przypadku zostanie wykonanych tyle samo porównań co w dwóch wcześniejszych algorytmach: $2 \cdot (n - 1)$.

Zastanówmy się, jak znaleźć minimum i maksimum zbioru, wykonując mniej operacji porównania niezależnie od położenia szukanych wartości w tablicy. W tym celu można porównywać elementy parami: pierwszy z drugim, trzeci z czwartym itd. Następnie porównujemy mniejszy element z pary z aktualnym minimum, a większy – z aktualnym maksimum. Działanie algorytmu dla ciągu 12, 8, 5, 16, 3, 6, 7, 9, 10, 14 pokazuje rysunek 10.1.



Rys. 10.1. Wyznaczenie minimum i maksimum zbioru liczb: 12, 8, 5, 16, 3, 6, 7, 9, 10, 14

Warto wiedzieć

Algorytm jednoczesnego wyszukiwania minimum i maksimum w zbiorze opiera się na metodzie „dziel i zwyciężaj". Problem został podzielony na dwa mniejsze problemy: znalezienie minimum wśród mniejszych liczb z par oraz znalezienie maksimum wśród większych liczb z par.

Algorytm porównujący parami sąsiednie elementy wygodniej będzie zapisać, jeśli założymy, że liczba elementów tablicy jest parzysta.

Dla zbioru n-elementowego liczba porównań elementów parami jest równa $n/2$. Aby znaleźć minimum wśród mniejszych elementów z par, trzeba wykonać $n/2 - 1$ porównań, aby znaleźć maksimum wśród większych elementów z par – również $n/2 - 1$ porównań. Łącznie, kiedy szukamy i minimum, i maksimum, liczba porównań wynosi:

$$n/2 + (n/2 - 1) + (n/2 - 1) = 3n/2 - 2$$

Oto zmodyfikowana specyfikacja problemu:

Specyfikacja

Dane: `A[0..n-1]` – tablica n liczb całkowitych, n jest liczbą parzystą.

Wynik: `mini` – najmniejszy element w tablicy `A`,
`maks` – największy element w tablicy `A`.

Dobra rada

Jeśli chcesz zastosować podany algorytm jednoczesnego wyszukiwania minimum i maksimum w tablicy o nieparzystej liczbie elementów, możesz powiększyć rozmiar tablicy o jeden i skopiować na ostatnią pozycję wartość ostatniego elementu.

Zapis algorytmu w pseudokodzie może być następujący:

```
jeśli A[0] > A[1] to
    mini ← A[1]
    maks ← A[0]
w przeciwnym przypadku
    mini ← A[0]
    maks ← A[1]
dla i ← 2, 4, 6, ..., n - 2 wykonuj
    jeśli A[i] > A[i+1] to
        jeśli A[i+1] < mini to mini ← A[i+1]
        jeśli A[i] > maks to maks ← A[i]
    w przeciwnym przypadku
        jeśli A[i] < mini to mini ← A[i]
        jeśli A[i+1] > maks to maks ← A[i+1]
```

Instrukcja warunkowa przed pętlą wyznacza wartości początkowe zmiennych `mini` i `maks`, porównując dwa pierwsze elementy tablicy. Wykonywane jest w tym celu jedno porównanie. Pętla wykona się $(n - 2) / 2$ razy. Po porównaniu dwóch sąsiednich elementów mniejszy z nich jest porównywany z aktualnym minimum, a większy – z aktualnym maksimum. W jednym obrocie pętli wykonywane są zatem trzy porównania. Ponieważ jedno porównanie jest wykonywane przed pętlą, w sumie powyższy algorytm wykona $1 + 3 \cdot (n - 2)/2 = 3 \cdot n/2 - 2$ porównań. Oto kod źródłowy funkcji realizującej ten algorytm:

Fragment kodu źródłowego programu wyszukującego jednocześnie minimum i maksimum w tablicy – funkcja `MiniMaks`

```
1.  void MiniMaks(int A[], int &mini, int &maks)
2.  {
3.      if (A[0]>A[1])
4.      {
5.          mini=A[1]; maks=A[0];
6.      }
7.      else
8.      {
9.          mini=A[0]; maks=A[1];
10.     }
11.     for (int i=2;i<N-1;i+=2)
12.         if (A[i]>A[i+1])
13.         {
14.             if (A[i+1]<mini) mini=A[i+1];
15.             if (A[i]>maks) maks=A[i];
16.         }
17.         else
18.         {
19.             if (A[i]<mini) mini=A[i];
20.             if (A[i+1]>maks) maks=A[i+1];
21.         }
22. }
```

Funkcja `MiniMaks` ma zwrócić dwie liczby całkowite: minimum i maksimum. Wartości są zwracane za pośrednictwem parametrów, dlatego funkcja jest typu **void**. Domyślnie parametry **typu prostego przekazywane są przez wartość**, czyli operacje wewnątrz funkcji wykonywane są na kopii parametru, a zmiany nie są zachowywane po zakończeniu działania funkcji. Żeby zachować zmiany w parametrze będącym zmienną typu prostego, działania powinny być wykonywane bezpośrednio na zmiennej, czyli parametrem aktualnym musi być adres zmiennej. Sygnalizuje to znak & (linia 1) umieszczony przed nazwą parametru formalnego. Taki sposób przekazania parametru w języku C++ nazywamy **przekazaniem parametru przez referencję**.

Typ prosty, s. 142

Przekazanie parametru przez wartość, s. 108

Przekazanie parametru przez referencję

Ćwiczenie 4

Napisz program wyszukujący jednocześnie największy i najmniejszy element tablicy. Wykorzystaj przedstawioną powyżej funkcję `MiniMaks`.

Podsumowanie

- Algorytm przeszukiwania liniowego wykorzystuje się do poszukiwania elementu w zbiorze nieuporządkowanym.
- W algorytmie przeszukiwania liniowego sprawdzamy, czy kolejne elementy zbioru są równe szukanemu – czynności powtarzamy do momentu, gdy znajdziemy element lub przejrzymy wszystkie elementy zbioru.
- Algorytm przeszukiwania binarnego stosuje się do poszukiwania elementu w zbiorze uporządkowanym.
- Kiedy stosujemy algorytm przeszukiwania binarnego, w kolejnych krokach dzielimy zbiór na pół i odrzucamy część, w której na pewno nie ma szukanego elementu. Czynności te powtarzamy, dopóki nie pozostanie jeden element.
- Żeby znaleźć równocześnie minimum i maksimum zbioru, można porównywać elementy zbioru parami: pierwszy z drugim, trzeci z czwartym itd. Po każdym porównaniu elementów w parze porównujemy mniejszy element z pary z aktualnym minimum zbioru, a większy z aktualnym maksimum.
- W przypadku, gdy parametrem funkcji jest tablica, do funkcji przekazywany jest adres tablicy, więc zmiany dokonane na takim parametrze są zachowywane po zakończeniu działania funkcji. Taki sposób przekazania parametru nazywamy przekazaniem przez wskaźnik.
- Jeśli chcemy zachować zmiany dokonane w funkcji na parametrze typu prostego, to powinniśmy w nagłówku funkcji przed nazwą parametru formalnego wstawić znak &. Wtedy do funkcji zostanie przekazany adres zmiennej będącej parametrem aktualnym. Taki sposób przekazania parametru nazywamy przekazaniem przez referencję.

Zadania

* **1** Napisz program, który sprawdzi, czy dana liczba znajduje się w tablicy n liczb całkowitych. Wykorzystaj funkcję realizującą algorytm przeszukiwania liniowego. Wynikiem funkcji powinna być pozycja szukanego elementu lub wartość n, jeśli elementu nie ma w tablicy.

* **2** Napisz program, który sprawdzi, czy dana liczba znajduje się w uporządkowanej tablicy n liczb całkowitych. Wykorzystaj funkcję realizującą algorytm przeszukiwania binarnego. Wynikiem funkcji powinna być pozycja szukanego elementu lub wartość n, jeśli szukanego elementu nie ma w tablicy.

* **3** Napisz program wyznaczający rozpiętość zbioru danych znajdujących się w tablicy n liczb całkowitych.

** **4** Napisz program, który w tablicy n liczb całkowitych z zakresu od 0 do 999 999 999 znajdzie liczbę o największej sumie cyfr.

** **5** Napisz program, który w tablicy n liczb całkowitych znajdzie pierwszy element parzysty. Przyjmij, że w tablicy najpierw występują liczby nieparzyste, potem parzyste oraz że tablica zawiera co najmniej jedną liczbę nieparzystą i jedną parzystą. Zastosuj algorytm przeszukiwania binarnego.

** **6** Przeanalizuj podany pseudokod i zastanów się, czy dla każdych danych realizuje on algorytm przeszukiwania binarnego. Odpowiedź uzasadnij.

```
funkcja S(A[],x)
    lewy ← 0
    prawy ← n - 1
    dopóki lewy < prawy wykonuj
        srodek ← (lewy + prawy) div 2
        jeśli A[srodek] > x to prawy ← srodek - 1
        w przeciwnym przypadku lewy ← srodek
    zwróć A[lewy] = x i zakończ
```

** **7** Napisz funkcję, która dla danej liczby całkowitej dodatniej znajdzie jej pierwiastek kwadratowy dyskretny. Zastosuj algorytm przeszukiwania binarnego.
Uwaga: Pierwiastek dyskretny z danej liczby to największa liczba całkowita mniejsza lub równa pierwiastkowi z tej liczby.

*** **8** Napisz funkcję, której wynikiem będzie liczba elementów z przedziału $[a; b]$ (a, b – liczby całkowite, $a < b$), znajdujących się w uporządkowanej niemalejąco tablicy n liczb całkowitych. Zastosuj algorytm przeszukiwania binarnego.

*** **9** Zapisz w wybranej przez siebie notacji algorytm zgadywania liczby w grze w 20 pytań przy założeniu, że jedna odpowiedź może być błędna. Jaka jest minimalna liczba pytań potrzebnych do odgadnięcia liczby?

11. Jak ocenić złożoność obliczeniową algorytmu?

Zasoby internetu są dziś jednym z najważniejszych źródeł wiedzy. Każdego dnia wyszukiwarki internetowe wykorzystywane są miliardy razy do znajdowania najbardziej pasujących informacji z ogromnej liczby stron internetowych. Samo znalezienie najwłaściwszych adresów internetowych to nie wszystko – wyszukiwarki mają też za zadanie błyskawicznie zwrócić wyniki. Wymaga to stosowania wielu bardzo szybko działających algorytmów. Z tego tematu dowiesz się, jak ocenić efektywność algorytmu.

Cele lekcji

- Dowiesz się, jak szacować czasową złożoność obliczeniową algorytmów.
- Poznasz notację dużego O.
- Porównasz złożoność czasową różnych algorytmów.
- Dowiesz się, czym jest pamięciowa złożoność obliczeniowa algorytmów i jak ją szacować.
- Poznasz pojęcie algorytmu optymalnego.

Ćwiczenie 1

W wybranej przez siebie wyszukiwarce internetowej wpisz frazę „złożoność obliczeniowa algorytmów". Ile wyników zostało znalezionych? W jakim czasie?

11.1. Czasowa złożoność obliczeniowa

Efektywność algorytmu mierzymy za pomocą złożoności obliczeniowej, na którą składają się złożoność czasowa oraz złożoność pamięciowa.

Czasowa złożoność obliczeniowa

Czasową złożoność obliczeniową algorytmu wyrażamy liczbą najczęściej wykonywanych operacji charakterystycznych dla algorytmu w zależności od rozmiaru danych. Operacje charakterystyczne dla danego algorytmu nazywamy **operacjami dominującymi** lub **elementarnymi**. Ich liczba rośnie zazwyczaj wraz ze wzrostem wielkości danych. Dla jednych algorytmów liczba operacji rośnie bardzo szybko, dla innych powoli.

Operacje dominujące (elementarne)

Porównajmy dwa algorytmy znajdowania elementu w uporządkowanym zbiorze n-elementowym – **przeszukiwanie liniowe** i **przeszukiwanie binarne**.

Przeszukiwanie liniowe, przeszukiwanie binarne, s. 153–158

Dla tych algorytmów operacjami dominującymi będą operacje porównania. W algorytmie przeszukiwania liniowego w najgorszym przypadku,

czyli gdy element szukany nie występuje w zbiorze albo zostaje znaleziony jako ostatni, wykonamy n porównań. W algorytmie przeszukiwania binarnego będzie to $\log_2 n$ porównań.

Tabela 11.1 pokazuje różnice we wzroście tych wartości.

Rozmiar danych (n)	Przeszukiwanie liniowe: liczba operacji (n)	Przeszukiwanie binarne: liczba operacji ($\log_2 n$)
10	10	4
100	100	7
1 000	1 000	10
10 000	10 000	14
100 000	100 000	17
1 000 000	1 000 000	20
10 000 000	10 000 000	24
100 000 000	100 000 000	27

Tabela 11.1. Wartości n i $\log_2 n$ jako liczba operacji w algorytmach przeszukiwania liniowego i binarnego

Warto wiedzieć

Symbol $\lceil \ \rceil$ oznacza zaokrąglenie w górę, często nazywane sufitem.

Warto wiedzieć

Zapis $\log_2 n$ oznacza logarytm przy podstawie 2 z n. Wyraża on wartość wykładnika potęgi o podstawie 2, takiego że $2^k = n$, tzn. $\log_2 n = k \Leftrightarrow n = 2^k$.

Warto wiedzieć

Czasową złożoność obliczeniową wyraża liczba dominujących operacji, a nie czas wykonania algorytmu.

Wartość, która wyraża zależną od rozmiaru danych liczbę operacji wykonanych w najgorszym przypadku, nazywamy **złożonością pesymistyczną**.

Złożoność pesymistyczna

Jeżeli poszukiwany element występuje w zbiorze, to za pomocą algorytmu przeszukiwania liniowego prawdopodobnie znajdziemy go z wykorzystaniem mniejszej liczby operacji, niż wskazywałaby złożoność pesymistyczna. Wykonamy (w zależności od położenia elementu w tablicy) raz mniej, raz więcej operacji, ale średnio będzie ich około $n/2$. Tak wyznaczaną liczbę operacji nazywa się **złożonością oczekiwaną** lub **średnią**.

Złożoność oczekiwana (średnia)

Kiedy oceniamy złożoność czasową algorytmu, interesuje nas rząd wielkości liczby wykonywanych instrukcji, a nie ich dokładna liczba. Przyjmujemy taki sposób szacowania, ponieważ uwzględniamy tylko operacje dominujące, a nie wszystkie wykonywane instrukcje. Poza tym nie ma znaczenia, czy algorytm wykona trochę więcej czy trochę mniej operacji. Wykonując program na komputerze, nie zauważymy tych różnic.

Realizacja tego samego algorytmu (wykonującego taką samą liczbę operacji) na różnych komputerach może też być różna w zależności od parametrów technicznych urządzenia – przede wszystkim z powodu różnic w procesorach i pamięciach komputerowych.

Z powyższych względów do szacowania złożoności czasowej algorytmu najczęściej stosujemy **notację dużego O**, wyrażającą rząd wielkości liczby wykonywanych operacji:

Notacja dużego O

O(*rząd wielkości liczby wykonywanych operacji dominujących*)

Złożoność liniowa
Złożoność logarytmiczna

Spośród przedstawionych wcześniej algorytmów przeszukiwania liniowego i binarnego pierwszy ma **złożoność liniową**: $O(n)$, a drugi – **złożoność logarytmiczną**: $O(\log n)$. Zwróć uwagę, że w notacji pominęliśmy podstawę logarytmu. Możemy to zrobić, ponieważ interesuje nas jedynie rząd wielkości.

Warto wiedzieć

Ze wzoru na zamianę podstawy logarytmu:
$\log_2 n = \frac{1}{\log 2} \cdot \log n$
wynika, że podstawa nie ma znaczenia w notacji dużego O.

Rozważmy problem polegający na sprawdzeniu, czy w n-elementowej tablicy liczb jakiś element się powtarza. Oto specyfikacja problemu:

Specyfikacja

Dane: `A[0..n-1]` – tablica n liczb całkowitych.

Wynik: wartość logiczna **`prawda`**, jeśli jakiś element występuje w tablicy A więcej niż raz, **`fałsz`** – w przeciwnym przypadku.

Warto wiedzieć

Formalnie piszemy, że $g(n) = O(f(n))$, gdy liczba operacji $g(n)$ jest rzędu co najwyżej $f(n)$, tzn. o ile $g(n) \leqslant C \cdot f(n)$ dla pewnej stałej C i dużych wartości n.

Jeden z algorytmów rozwiązujących ten problem może porównywać pierwszy element z kolejnymi – od 2 do ostatniego ($n - 1$ porównań), następnie drugi z kolejnymi – od 3 do ostatniego ($n - 2$ porównań) itd., aż do znalezienia dwóch takich samych liczb. Jeśli ich nie znajdzie lub jeśli znalezione liczby to przedostatni i ostatni element tablicy, to liczba wykonanych porównań jako operacji dominujących będzie wynosiła:

$$(n-1)+(n-2)+\ldots+2+1=\frac{(n-1)+1}{2}\cdot(n-1)=\frac{n^2}{2}-\frac{n}{2}$$

Złożoność kwadratowa

Omówiony algorytm ma **złożoność kwadratową**: $O(n^2)$. Dla dużych n wartość $n/2$ jest zdecydowanie mniejsza od wartości $n^2/2$. Na przykład już dla $n = 1000$ wartość $n/2$ stanowi 0,1% wartości $n^2/2$.

Warto wiedzieć

Przy ocenie złożoności algorytmu stosowana jest też notacja teta: $\Theta(f(n))$. Oznacza ona, że liczba operacji, które wykona algorytm, zawsze będzie rzędu $f(n)$. W przypadku notacji $O(f(n))$ możliwe jest, że dla specyficznych danych algorytm wykonuje mniej operacji.

Określając złożoność obliczeniową w notacji dużego O, bierzemy pod uwagę najwyższą potęgę oszacowanej liczby operacji. Współczynnik przy tej potędze pomijamy, ponieważ nie zmienia on rzędu wielkości.

Tabela 11.2 zawiera przykłady algorytmów o różnej złożoności obliczeniowej.

Zapis w notacji dużego O	Czasowa złożoność obliczeniowa	Przykład algorytmu
$O(\log n)$	Złożoność logarytmiczna	Przeszukiwanie binarne
$O(\sqrt{n})$	Złożoność pierwiastkowa	Sprawdzenie pierwszości liczby
$O(n)$	Złożoność liniowa	Przeszukiwanie liniowe
$O(n \cdot \log n)$	Złożoność liniowo-logarytmiczna	Sortowanie przez scalanie
$O(n^2)$	Złożoność kwadratowa	Sprawdzenie, czy w n-elementowej tablicy powtarza się jakiś element
$O(n^3)$	Złożoność sześcienna	Szukanie podciągu o maksymalnej sumie metodą naiwną

Tabela 11.2. Przykłady algorytmów o różnej złożoności obliczeniowej

Sortowanie przez scalanie, s. 298–306

Szukamy podciągu spójnego o maksymalnej sumie, s. 207–210

Ćwiczenie 2

Przygotuj arkusz kalkulacyjny, w którym dla różnych wartości n (zarówno małych, jak i dużych) obliczysz rząd wielkości liczby operacji dla danych z tabeli 11.2. Porównaj wykresy funkcji opisujących podane złożoności.

Liczba operacji wykonywanych przez niektóre algorytmy jest stała i nie zależy od rozmiaru danych. Mówimy wówczas o **złożoności stałej** i oznaczamy ją $O(1)$. Czasami też rozmiar danych określony jest więcej niż jednym parametrem. Na przykład przejrzenie wszystkich elementów n tablic, z których każda ma m elementów, będzie miało złożoność $O(n \cdot m)$.

Złożoność stała

Wyróżniamy również **złożoności wykładnicze**, które oznaczane są jako $O(2^n)$ i $O(n!)$. Wartości te, podobnie jak funkcje 2^n i $n!$, rosną bardzo szybko. Algorytmy o tego typu złożonościach działają efektywnie jedynie dla małych rozmiarów danych. Istnieją jednak problemy, dla których nie znamy szybszych rozwiązań.

Złożoność wykładnicza

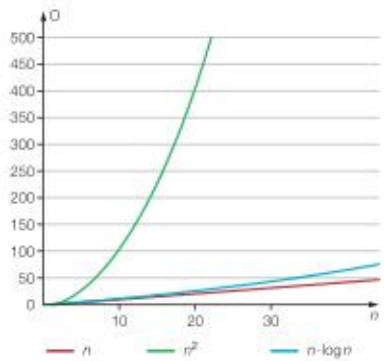
Warto wiedzieć

$n! > 2^n$ dla $n > 3$

Warto wiedzieć

Przykładem algorytmu o złożoności wykładniczej jest rozwiązanie problemu wież Hanoi.

Na rysunku 11.1 umieszczono wykresy funkcji reprezentujące liczbę operacji dla wybranych złożoności czasowych. Dla dużych wartości n już algorytmy o złożoności $O(n^2)$ działają wolno.



Rys. 11.1. Porównanie wykresów funkcji reprezentujących wybrane złożoności obliczeniowe

Zapamiętaj

Złożoność obliczeniowa algorytmów jest jednym z podstawowych zagadnień informatyki. Pozwala przewidywać działanie algorytmów dla różnych zestawów danych i dobierać właściwe algorytmy do konkretnych sytuacji.

Warto wiedzieć

W wielu konkursach programistycznych rozwiązania zadań są oceniane za pomocą różnych zestawów testów, które sprawdzają zarówno poprawność działania algorytmu, jak i jego złożoność czasową.

11.2. Pamięciowa złożoność obliczeniowa

Pamięciowa złożoność obliczeniowa określa zapotrzebowanie programu na pamięć komputerową. Najczęściej wyraża się ją, podobnie jak złożoność czasową, w notacji dużego O – szacuje się rząd wielkości pamięci potrzebnej do przechowania struktur danych wykorzystywanych przez program.

Pamięciowa złożoność obliczeniowa

Na przykład algorytmy wyszukiwania liczby w zbiorze n-elementowym wykorzystują pojedynczą tablicę o rozmiarze n, więc ich złożoność pamięciowa wynosi $O(n)$. Kilka zmiennych pomocniczych **typu prostego** nie wpływa na rząd wielkości potrzebnej pamięci.

Typ prosty, s. 142

Dobra rada

W kodzie źródłowym programu nie deklaruj zmiennych ani struktur danych, których nie użyjesz. Rezerwowana jest na nie pamięć, która nie zostanie wykorzystana.

Oszacowanie złożoności pamięciowej algorytmów, które wykorzystują struktury dynamiczne, może sprawiać kłopot. Struktury te zmieniają swój rozmiar w trakcie działania programu. Omówimy je w części trzeciej podręcznika. Należy także pamiętać, że na wielkość pamięci używanej przez program ma wpływ kod binarny samego programu.

Ćwiczenie 3

Jeśli zamieniamy wartości dwóch zmiennych z użyciem zmiennej pomocniczej, to rezerwowana jest na nią dodatkowa pamięć. Zapisz w pseudokodzie ten algorytm bez wykorzystania trzeciej zmiennej oraz funkcji `swap`.

11.3. Algorytm optymalny

Algorytm optymalny

Algorytm optymalny to taka metoda rozwiązania problemu, która ma najmniejszą możliwą złożoność czasową. Jeśli algorytm jest optymalny, to nie istnieje metoda o mniejszej złożoności czasowej. Często jednak bardzo trudno udowodnić, że taka metoda nie istnieje.

Warto wiedzieć

Dla specyficznych danych algorytm poszukujący niezależnie minimum i maksimum może wykonać mniej porównań niż algorytm optymalny.

Pośród różnych algorytmów omawianych wcześniej były także metody optymalne. Na przykład algorytm znajdowania minimum lub maksimum w nieuporządkowanym zbiorze, wykonujący $n - 1$ porównań, jest optymalny. Istotnie, jeśli jakaś liczba ma być maksimum, to w ciągu n-elementowym musi występować $n - 1$ liczb mniejszych lub równych tej liczbie. Każdą z nich musimy co najmniej raz porównać z innym elementem.

Jednoczesne znajdowanie minimum i maksimum, s. 159–161

Innym przykładem algorytmu optymalnego jest **algorytm jednoczesnego znajdowania minimum i maksimum**. Wykonuje on nieznacznie mniej porównań niż algorytm, który niezależnie poszukuje minimum i maksimum. Obydwa algorytmy mają jednak złożoność liniową: $O(n)$, więc w praktyce nie ma znaczenia, który z nich zostanie zastosowany.

Przeszukiwanie binarne, s. 154–158

Algorytmem optymalnym jest też **przeszukiwanie binarne**.

A to ciekawe

Algorytmy w sieci

Znalezienie w sieci informacji, których potrzebujemy, byłoby praktycznie niemożliwe bez ich uporządkowania. Systemy rankingowe wyszukiwarek internetowych przeglądają indeksy wyszukiwania, stosując wiele różnych algorytmów, analizujących takie aspekty jak: treść zapytania, przydatność stron, wiarygodność źródła, czas zamieszczenia informacji, lokalizacja użytkownika. Gdyby wykorzystywane algorytmy nie działały bardzo szybko dla dużych zestawów danych, na wyniki dla pojedynczego zapytania trzeba byłoby czekać nawet kilkanaście minut lub dłużej.

Podsumowanie

- Wyróżniamy dwa rodzaje złożoności obliczeniowej: czasową i pamięciową.
- Złożoność czasową określamy, szacując rząd wielkości liczby wykonywanych operacji dominujących (wykonywanych w algorytmie najczęściej) względem rozmiaru danych.
- Złożoność czasowa pesymistyczna to rząd wielkości liczby wykonywanych operacji dominujących w najgorszym przypadku.
- Złożoność czasowa oczekiwana (średnia) to rząd wielkości liczby wykonywanych operacji dominujących dla typowych danych.
- Złożoność pamięciowa to rząd wielkości pamięci potrzebnej do utworzenia struktur danych wykorzystywanych przez algorytm. Ten rząd wielkości jest wyrażony względem rozmiaru danych.
- Do wyrażenia złożoności obliczeniowej stosujemy notację dużego O.
- Algorytmy o złożonościach czasowych logarytmicznej: $O(\log n)$, liniowej: $O(n)$ i liniowo-logarytmicznej: $O(n \cdot \log n)$ działają szybko także dla dużych zestawów danych. Algorytmy o złożoności kwadratowej: $O(n^2)$ mogą działać wolniej dla dużych zestawów danych.
- Algorytmy o złożoności sześciennej: $O(n^3)$ oraz wykładniczej: $O(2^n)$ i $O(n!)$ są nieefektywne dla dużych danych. Działają poprawnie, ale wolno.
- Jeśli złożoność obliczeniowa nie zależy od rozmiaru danych, oznaczamy ją $O(1)$.
- Algorytm optymalny to taka metoda rozwiązania problemu, która ma najmniejszą możliwą złożoność czasową.

Zadania

* **1** Podaj przykład algorytmu o złożoności pamięciowej $O(1)$ i złożoności czasowej zależnej od rozmiaru danych.

* **2** Oszacuj złożoność pamięciową i czasową algorytmu sprawdzającego, czy dany napis jest palindromem.

** **3** Podaj przykład algorytmu o złożoności czasowej $O\left(n \cdot \sqrt{n}\right)$.

** **4** Napisz program losujący tablicę n liczb całkowitych i sprawdzający, czy jakaś liczba powtarza się w tej tablicy.

** **5** Podaj optymalny algorytm liczenia sumy kolejnych liczb naturalnych od a do b, gdzie $a < b$. Oszacuj złożoność czasową tego algorytmu.

** **6** Oszacuj złożoność czasową algorytmów zamiany liczby dziesiętnej na binarną oraz szybkiego podnoszenia do potęgi.

*** **7** Oszacuj pesymistyczną złożoność czasową algorytmu Euklidesa w wersjach z odejmowaniem oraz z dzieleniem.

12. Metody sortowania prostego

Sortowanie to jeden z podstawowych problemów informatycznych. Jest wykorzystywane np. przy wyszukiwaniu elementu o określonych własnościach czy kompresji danych. Istnieje wiele różnych algorytmów sortowania o różnej efektywności. Chociaż współczesne oprogramowanie (w tym biblioteki języków programowania) dostarcza nam gotowe narzędzia do porządkowania, to zrozumienie pojęcia sortowania jest często uważane za pierwszy krok do opanowania algorytmiki. W tym temacie zajmiesz się metodami sortowania prostego.

Cele lekcji

- Poznasz metody sortowania prostego: sortowanie bąbelkowe (przez prostą zamianę) oraz sortowanie przez wstawianie.
- Przypomnisz sobie algorytm sortowania przez wybieranie poznany w szkole podstawowej.
- Porównasz metody sortowania prostego.
- Oszacujesz czasową złożoność obliczeniową algorytmów sortowania prostego.

Sortowanie

Przypomnijmy, że **sortowanie** to porządkowanie danych według określonych kryteriów. Porządkować można dowolny ciąg danych tego samego typu, np. liczby rosnąco, słowa alfabetycznie, dane osobowe według daty urodzenia. Musi być tylko jednoznacznie określony sposób porównania dwóch elementów danego typu.

Przykłady w tym temacie będą dotyczyć liczb całkowitych. Omówimy różne metody sortowania i utworzymy funkcje sortujące dane.

12.1. Sformułowanie problemu sortowania

W programach, które napiszemy, danymi wejściowymi będzie ciąg n losowych liczb całkowitych: $a_0, a_1, \ldots, a_{n-1}$, natomiast wynikiem – elementy tego ciągu uporządkowane niemalejąco. Liczby zapamiętamy w tablicy.

Oto specyfikacja problemu:

Specyfikacja

Dane: `A[0..n-1]` – tablica n liczb całkowitych.

Wynik: tablica `A` z uporządkowanymi elementami,
`A[0]` $\leqslant$ `A[1]` $\leqslant \ldots \leqslant$ `A[n-1]`.

Funkcja `main` programu sortującego liczby (niezależnie od zastosowanego algorytmu sortowania) powinna uwzględniać: wylosowanie i wypisanie tablicy liczb, posortowanie elementów tablicy, wypisanie posortowanej tablicy liczb.

Do wylosowania tablicy liczb całkowitych użyjemy funkcji `Losuj`, natomiast do wypisania elementów tablicy – funkcji `Wypisz`. Kod źródłowy funkcji losującej oraz funkcji wypisującej elementy tablicy może wyglądać tak samo jak w **programach wyszukujących element w tablicy**. Rozmiar tablicy określimy za pomocą stałej `N`, którą zadeklarujemy przed funkcjami.

Fragment kodu źródłowego programu wyszukującego element w tablicy, s. 152

Funkcja `main` programu sortującego niemalejąco tablicę liczb może wyglądać następująco:

Fragment kodu źródłowego programu sortującego niemalejąco tablicę liczb – funkcja `main`

```
1.  int main()
2.  {
3.      int A[N];
4.      srand(time(NULL));
5.      Losuj(A);
6.      Wypisz(A);
7.      Sortuj(A);
8.      Wypisz(A);
9.      return 0;
10. }
```

W linii 3 jest deklarowana tablica `N` liczb całkowitych, a w linii 4 inicjowany generator liczb losowych. W funkcji `main` użyliśmy funkcji `Sortuj`, która będzie sortować tablicę liczb zgodnie ze specyfikacją problemu. Należy pamiętać o dodaniu w programie bibliotek `cstdlib` i `ctime` (w pierwszej jest dostępna funkcja `srand`, a w drugiej – funkcja `time`) oraz definicji stałej `N` określającej rozmiar tablicy.

12.2. Sortowanie bąbelkowe (przez prostą zamianę)

Sortowanie bąbelkowe (przez prostą zamianę)

Najprostszą metodą sortowania jest **sortowanie bąbelkowe**, nazywane też **sortowaniem przez prostą zamianę**. W metodzie tej porównywane są sąsiadujące ze sobą elementy. Jeżeli są ustawione w niewłaściwym porządku, to są zamieniane miejscami – stąd nazwa „przez prostą zamianę". Na przykład jeżeli chcemy posortować zbiór niemalejąco, a pierwszy z porównywanych elementów jest większy od drugiego – zamieniamy je miejscami.

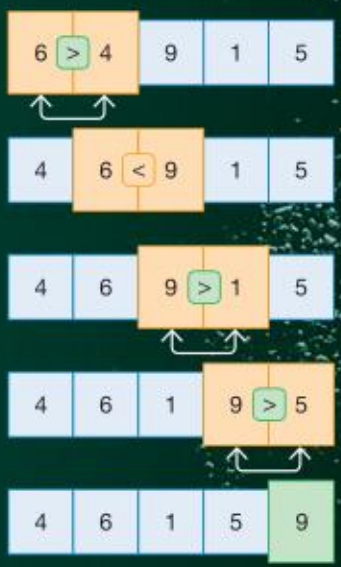
Nazwa „sortowanie bąbelkowe" powstała przez analogię do zjawiska wypływania na powierzchnię wody bańki powietrza: bańka o największej objętości wypływa jako pierwsza. Podczas sortowania największy element porządkowanego ciągu również „wypływa" jako pierwszy (jako pierwszy jest ustawiany na właściwym miejscu).

Algorytm sortowania bąbelkowego

Porządkowanie elementów tą metodą pokażemy na przykładzie pięcioelementowej tablicy, w której na kolejnych pozycjach znajdują się liczby: 6, 4, 9, 1, 5. Elementy uporządkujemy niemalejąco. Pierwszy krok ilustruje, jak umieścić właściwy element na ostatniej pozycji. W kolejnych krokach postępujemy analogicznie.

Krok 1

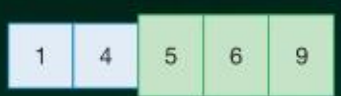
- Porównujemy pierwszy element z drugim. 6 > 4, więc zamieniamy liczby miejscami.
- Porównujemy drugi element z trzecim. 6 < 9, więc nie zamieniamy liczb miejscami.
- Porównujemy trzeci element z czwartym. 9 > 1, więc zamieniamy liczby miejscami.
- Porównujemy czwarty element z piątym. 9 > 5, więc zamieniamy liczby miejscami.
- W wyniku czterech porównań największy element tablicy (liczba 9) znalazł się na ostatniej pozycji.



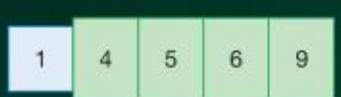
W drugim kroku umieścimy właściwy element na przedostatniej pozycji. Przy porównaniach pomijamy ostatni element tablicy, ponieważ jest już na właściwym miejscu. W trzecim kroku przy porównaniach pomijamy dwa ostatnie elementy tablicy, w czwartym – trzy ostatnie itd. Poniżej przedstawione są wyniki otrzymane w kolejnych krokach.

Krok 2 Porównujemy pary liczb: 4 i 6, 6 i 1 oraz 6 i 5. Liczba 6 znalazła się na właściwej, przedostatniej pozycji.

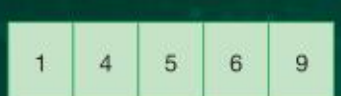
Krok 3 Porównujemy pary liczb 4 i 1 oraz 4 i 5. W wyniku tych operacji liczba 5 jest na właściwej pozycji.



Krok 4 Porównujemy parę liczb 1 i 4. Znalezienie drugiego elementu ciągu wskazuje jednocześnie najmniejszy element ciągu.



Wynikiem działania algorytmu jest tablica:



Oto zapis algorytmu porządkującego elementy tablicy niemalejąco metodą sortowania bąbelkowego:

```
dla i ← 1, 2, ..., n - 1 wykonuj
    dla j ← 0, 1, ..., n - i - 1 wykonuj
        jeśli A[j] > A[j+1] to
            pom ← A[j]
            A[j] ← A[j+1]
            A[j+1] ← pom
```

Dobra rada

Aby uporządkować elementy nierosnąco, zmień w instrukcji warunkowej znak > na <.

Zwróć uwagę na zakres zmiennych sterujących i oraz j poszczególnych pętli. Jedno wykonanie zewnętrznej pętli wyznacza największy element badanego ciągu – „wypłynięcie jednego bąbelka". Ciągu jednoelementowego nie musimy sortować, dlatego liczba powtórzeń zewnętrznej pętli jest o jeden mniejsza od liczby elementów tablicy. Liczba powtórzeń wewnętrznej pętli jest zależna od wartości zmiennej sterującej zewnętrznej pętli. W kolejnych iteracjach zewnętrznej pętli liczba powtórzeń wewnętrznej pętli zmniejsza się o jeden. Do zamiany miejscami dwóch liczb wykorzystaliśmy zmienną pomocniczą pom.

Kod źródłowy funkcji `Sortuj` realizującej algorytm sortowania bąbelkowego zapisany w pseudokodzie może być następujący:

```
1.  void Sortuj(int A[])
2.  {
3.      int i, j, pom;
4.      for (i=1;i<N;i++)
5.          for (j=0;j<N-i;j++)
6.              if (A[j]>A[j+1])
7.              {
8.                  pom=A[j];
9.                  A[j]=A[j+1];
10.                 A[j+1]=pom;
11.             }
12. }
```

Fragment kodu źródłowego programu sortującego niemalejąco tablicę liczb – funkcja realizująca algorytm sortowania bąbelkowego

Dobra rada

Do zamiany wartości dwóch sąsiednich elementów tablicy możesz wykorzystać funkcję **swap**. Instrukcja z użyciem tej funkcji wyglądałaby następująco:

```
swap(A[j],A[j+1]);
```

Zwróć uwagę, że jeśli w jednej iteracji wewnętrznej pętli nie zostanie dokonana żadna zamiana liczb, to tablica jest już posortowana. Można więc zmodyfikować algorytm tak, aby kończył działanie, kiedy nie dokona żadnej zamiany. W praktyce nie stosuje się tego rozwiązania, ponieważ należy wykonywać dodatkowe porównanie przy każdej iteracji zewnętrznej pętli, a szanse na to, że zamiany zostaną zakończone znacząco wcześniej, są niewielkie.

Ćwiczenie 1

Napisz i przetestuj program sortujący niemalejąco tablicę liczb całkowitych algorytmem sortowania bąbelkowego.

12.3. Sortowanie przez wybieranie

Sortowanie przez wybieranie

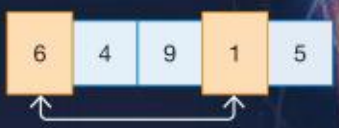
Sortowanie przez wybieranie polega na wyszukaniu elementu, który powinien się znaleźć na danej pozycji, i zamianie miejscami tego elementu z elementem znajdującym się dotychczas na tej pozycji.

Jeśli np. chcemy posortować tablicę niemalejąco, w pierwszej iteracji poszukujemy najmniejszej liczby w całej tablicy i zamieniamy ją miejscami z pierwszym elementem tablicy. W drugiej iteracji poszukujemy liczby, która powinna się znaleźć w tablicy jako druga. Pomijamy pierwszy element tablicy, ponieważ jest już na właściwym miejscu. Po znalezieniu szukanej liczby zamieniamy ją miejscami z drugim elementem tablicy. W kolejnych krokach postępujemy podobnie. Może się zdarzyć, że kolejny poszukiwany element jest już na właściwej pozycji.

Algorytm sortowania przez wybieranie

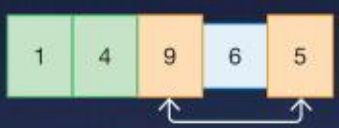
Działanie algorytmu pokażemy dla tablicy, w której na kolejnych pozycjach znajdują się liczby: 6, 4, 9, 1, 5. Elementy uporządkujemy niemalejąco.

Krok 1 Szukamy najmniejszego elementu w całej tablicy. Jest nim liczba 1. Zamieniamy ją miejscami z pierwszą liczbą w tablicy – liczbą 6.

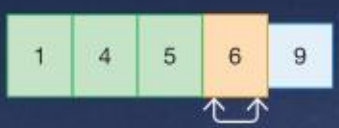


Krok 2 Szukamy najmniejszego elementu tablicy, pomijając element pierwszy. Szukaną liczbą jest 4 – znajduje się ona na właściwej pozycji.

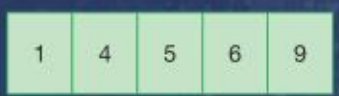
Krok 3 Pomijamy pierwszy i drugi element tablicy i szukamy najmniejszej wśród pozostałych liczb – jest to liczba 5. Zamieniamy ją miejscami z trzecim elementem tablicy – liczbą 9.



Krok 4 Pomijamy trzy pierwsze elementy tablicy i szukamy liczby na przedostatnią pozycję. Jest to liczba 6, która znajduje się już na właściwym miejscu. Wyznaczenie przedostatniego elementu wskazuje jednocześnie element na ostatnią pozycję.



Wynikiem działania algorytmu jest tablica:



Zapis algorytmu porządkującego elementy tablicy niemalejąco metodą sortowania przez wybieranie może być następujący:

```
dla i ← 0, 1, ..., n - 2 wykonuj
    m ← i
    dla j ← i + 1, i + 2, ..., n - 1 wykonuj
        jeśli A[j] < A[m] to m ← j
    pom ← A[i]
    A[i] ← A[m]
    A[m] ← pom
```

Warto wiedzieć

Aby wyeliminować sytuację, w której liczbę zamieniamy samą ze sobą, trzeba by wprowadzić dodatkowe porównanie elementów przy każdej iteracji zewnętrznej pętli. Sytuacja taka dla dłuższych ciągów będzie jednak zachodzić rzadko.

W każdej iteracji zewnętrznej pętli jest znajdowany najmniejszy element badanego ciągu. Liczba powtórzeń wewnętrznej pętli zależy od wartości zmiennej sterującej zewnętrznej pętli. Rozpoczynamy od porównania elementu znajdującego się na pozycji, na którą wybierana jest liczba, z następnym elementem. Czasami liczba jest zamieniana sama ze sobą. Oto zapis funkcji sortującej tablicę liczb:

```
1.  void Sortuj(int A[])
2.  {
3.      int i, j, m, pom;
4.      for (i=0;i<N-1;i++)
5.      {
6.          m=i;
7.          for (j=i+1;j<N;j++)
8.              if (A[j]<A[m]) m=j;
9.          pom=A[i];
10.         A[i]=A[m];
11.         A[m]=pom;
12.     }
13. }
```

Fragment kodu źródłowego programu sortującego niemalejąco tablicę liczb – funkcja realizująca algorytm sortowania przez wybieranie

Ćwiczenie 2

Napisz i przetestuj program sortujący niemalejąco tablicę liczb całkowitych algorytmem sortowania przez wybieranie.

12.4. Sortowanie przez wstawianie

Metoda **sortowania przez wstawianie** polega na przeglądaniu kolejnych elementów ciągu i wstawianiu ich w odpowiednich miejscach już uporządkowanego podciągu (fragment ciągu po wstawieniu elementu powinien być uporządkowany). Aby wstawić element ciągu we właściwe miejsce, zapamiętuje się go w zmiennej pomocniczej. Następnie elementy posortowanego podciągu, które powinny się znaleźć za danym elementem, przesuwa się w prawo o jedną pozycję. Na końcu kopiuje się wartość ze zmiennej pomocniczej w zwolnione miejsce.

Sortowanie przez wstawianie

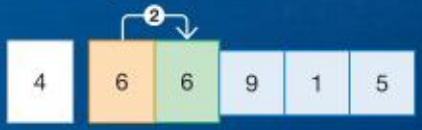
Algorytm sortowania przez wstawianie

Porządkowanie elementów tą metodą pokażemy na przykładzie pięcioelementowej tablicy, w której na kolejnych pozycjach znajdują się liczby: 6, 4, 9, 1, 5. Elementy uporządkujemy niemalejąco. Pierwszy krok ilustruje, jak uporządkować niemalejąco dwa elementy tablicy. Liczby przy strzałkach oznaczają kolejność wykonywania operacji.

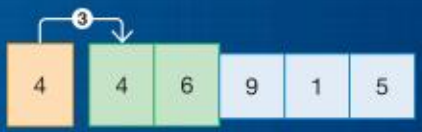
Krok 1

- Przyjmujemy, że pierwszy element tablicy jest pierwszym uporządkowanym elementem. Drugi element, liczbę 4, zapamiętujemy w zmiennej pomocniczej.

- Liczba 4 powinna się znaleźć przed liczbą 6, dlatego aby zrobić dla niej miejsce, przesuwamy liczbę 6 o jedną pozycję w prawo.



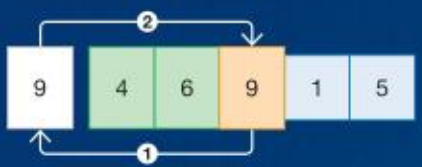
- Liczbę 4 ze zmiennej pomocniczej zapisujemy jako pierwszy element tablicy.



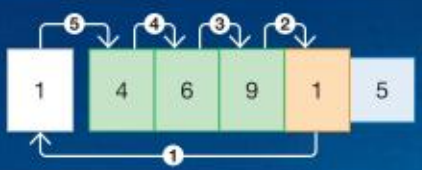
- Wynikiem jest uporządkowanie niemalejąco dwóch pierwszych elementów wyjściowej tablicy.

Postępując analogicznie, w drugim kroku uporządkujemy niemalejąco trzy elementy tablicy, w następnym – cztery itd. Poniżej znajduje się skrócony opis kolejnych kroków.

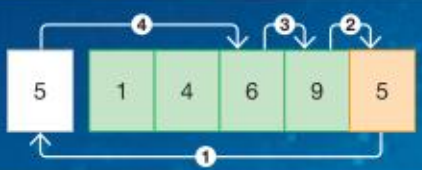
Krok 2 Zapamiętujemy liczbę 9 w zmiennej pomocniczej. Liczba 9 powinna się znaleźć za liczbami 4 i 6, dlatego nie trzeba zmieniać położenia tych liczb, a liczba 9 wraca na swoją pozycję.



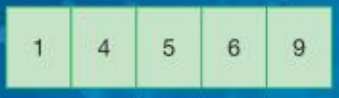
Krok 3 Zapamiętujemy liczbę 1 w zmiennej pomocniczej. Przesuwamy w prawo o jedną pozycję kolejno liczby: 9, 6 i 4. Zapisujemy liczbę 1 na pierwszym miejscu w tablicy.



Krok 4 Zapamiętujemy liczbę 5 w zmiennej pomocniczej. Przesuwamy liczbę 9, a następnie liczbę 6 o jedną pozycję w prawo. Wstawiamy liczbę 5 ze zmiennej pomocniczej w zwolnione miejsce.



Wynikiem działania algorytmu jest tablica:



Ponieważ na początku przyjmujemy, że pierwszy element tablicy jest uporządkowany, rozpoczynamy porządkowanie od drugiego elementu (o indeksie 1). Zapis algorytmu porządkującego liczby niemalejąco metodą sortowania przez wstawianie może wyglądać następująco:

```
dla i ← 1, 2, ..., n - 1 wykonuj
    pom ← A[i]    // zapisanie rozpatrywanego
                  // elementu w zmiennej pomocniczej
    j ← i - 1     // pozycja pierwszego elementu
                  // do ewentualnego przesunięcia
    dopóki j ≥ 0 oraz A[j] > pom wykonuj
        A[j+1] ← A[j]  // przesunięcie elementu
        j ← j - 1      // pozycja kolejnego elementu
    A[j+1] ← pom // wstawienie elementu ze zmiennej
                 // pomocniczej na zwolnioną pozycję
```

Warunek pętli **dopóki** można uprościć, wprowadzając **wartownika**. Wówczas funkcję zmiennej pomocniczej (oraz wartownika) będzie pełnić element A[0]. Elementy sortowane będą miały indeksy od 1 do n (liczba elementów tablicy będzie wynosić $n + 1$). Nie trzeba będzie sprawdzać, czy indeks tablicy jest większy lub równy zero. Nawet gdy wstawiany będzie element najmniejszy, to w ostatnim obrocie pętli (dla $j = 0$) zostanie on porównany sam ze sobą i pętla zakończy działanie. Oto zmodyfikowana specyfikacja problemu i pseudokod:

Wartownik, s. 64

Specyfikacja

Dane: A[1..n] – tablica n liczb całkowitych.

Wynik: tablica A z uporządkowanymi elementami, $A[1] \leqslant \ldots \leqslant A[n]$.

```
dla i ← 2, 3, ..., n wykonuj
    A[0] ← A[i]
    j ← i - 1
    dopóki A[j] > A[0] wykonuj
        A[j+1] ← A[j]
        j ← j - 1
    A[j+1] ← A[0]
```

A to ciekawe

Sortowanie w tańcu

Okazuje się, że można tłumaczyć metodę sortowania, dobrze się przy tym bawiąc. W różnych serwisach możemy obejrzeć, jak wyjaśniają zasady sortowania tancerze. Jeden z takich filmów dostępny jest pod linkiem https://www.youtube.com/watch?v=lyZQPjUT5B4.

Oto kod źródłowy funkcji realizującej pseudokod podany jako ostatni:

Fragment kodu źródłowego programu sortującego niemalejąco tablicę liczb – funkcja realizująca algorytm sortowania przez wstawianie

```
1.  void Sortuj(int A[])
2.  {
3.      int i, j;
4.      for (i=2;i<=N;i++)
5.      {
6.          A[0]=A[i];
7.          j=i-1;
8.          while (A[j]>A[0])
9.          {
10.             A[j+1]=A[j];
11.             j--;
12.         }
13.         A[j+1]=A[0];
14.     }
15. }
```

Element tablicy znajdujący się na pozycji 0 pełni funkcję wartownika, a porządkowane liczby to elementy tablicy na pozycjach od 1 do `N`. W związku z tym tablica w funkcji `main` powinna być deklarowana jako `N+1`-elementowa:

```
int A[N+1];
```

Dodatkowo w programie należy tak zmodyfikować funkcje `Losuj` i `Wypisz`, aby pętle przeglądały elementy tablicy o indeksach od 1 do `N`.

Ćwiczenie 3

Napisz i przetestuj program sortujący niemalejąco tablicę liczb całkowitych algorytmem sortowania przez wstawianie.

12.5. Porównanie metod sortowania prostego

Operacje dominujące (elementarne), s. 164

Porównamy teraz przedstawione w tym temacie algorytmy sortowania. Dla algorytmów sortowania prostego **operacjami dominującymi** (**elementarnymi**) będą operacje porównania elementów ciągu.

Policzmy te operacje dla algorytmów sortowania bąbelkowego oraz sortowania przez wybieranie. Można to dosyć łatwo zrobić, ponieważ w zapisie tych algorytmów występują tylko pętle **`for`**, które w tych algorytmach są wykonywane tyle samo razy niezależnie od sortowanych danych (wylosowanych liczb). Zewnętrzna pętla wykonuje się $n - 1$ razy. Wewnętrzna pętla za pierwszym razem jest powtarzana także $n - 1$ razy, za każdym kolejnym – o jeden mniej. Korzystając ze wzoru na sumę wyrazów ciągu arytmetycznego, możemy obliczyć liczbę porównań. Jest ona równa:

$$(n-1)+(n-2)+\ldots+2+1=\frac{(n-1)+1}{2}\cdot(n-1)=\frac{n^2}{2}-\frac{n}{2}$$

Zatem **czasowa złożoność obliczeniowa** jest kwadratowa: $O(n^2)$.

Czy algorytmy sortowania bąbelkowego i sortowania przez wybieranie są równorzędne? Zwróć uwagę, że nie liczyliśmy wszystkich operacji, a jedynie dominujące operacje porównania. Pamiętajmy, że algorytmy jeszcze zamieniają elementy tablicy, wykonując operacje przypisania wartości. Algorytm sortowania bąbelkowego realizuje to w wewnętrznej pętli, a sortowania przez wybieranie – w zewnętrznej, więc zdecydowanie mniej razy. Można powiedzieć, że ten drugi sposób porządkowania jest sprawniejszy.

Czasowa złożoność obliczeniowa, s. 164

Trudniej będzie oszacować liczbę porównań w algorytmie sortowania przez wstawianie. Wewnętrzna pętla `while` wykonuje się różną liczbę razy w zależności od danych – relacji pomiędzy liczbami w tablicy. Załóżmy sytuację pesymistyczną, kiedy za każdym razem przesuwamy wszystkie uporządkowane już elementy, czyli dany element jest wstawiany na pierwszą pozycję (tablica jest odwrotnie uporządkowana). Wówczas algorytm wykona dokładnie tyle samo porównań co algorytm sortowania bąbelkowego lub przez wybieranie. **Złożoność pesymistyczna** jest równa $\frac{n^2}{2} - \frac{n}{2}$, czyli $O(n^2)$. Dla większej liczby elementów ryzyko wylosowania liczb odwrotnie uporządkowanych jest bliskie zeru. Dla losowych ciągów będzie przesuwanych średnio około połowy elementów, więc sortowanie przez wstawianie wykona rzędu $\frac{n^2}{4}$ porównań. **Złożoność oczekiwana (średnia)** jest więc także kwadratowa: $O(n^2)$.

Złożoność pesymistyczna, s. 165

Złożoność oczekiwana (średnia), s. 165

Sortowanie przez wstawianie jest nieznacznie sprawniejsze w porównaniu z dwoma pozostałymi metodami sortowania prostego. Należy jednak pamiętać, że algorytm sortowania przez wstawianie, przesuwając elementy, wykonuje wiele operacji przypisania wartości (więcej niż sortowanie przez wybieranie, ale mniej niż sortowanie bąbelkowe).

Ćwiczenie 4

Napisz program sortujący niemalejąco tablicę n liczb całkowitych algorytmem sortowania przez wstawianie, a następnie zmodyfikuj go tak, aby liczył wykonywane porównania. Przetestuj działanie programu dla różnych wartości n. Porównaj otrzymane wyniki z wartością $\frac{n^2}{4}$.

Zapamiętaj

Jeśli sortujemy niewielką liczbę elementów, nie ma znaczenia, której metody sortowania użyjemy. Wszystkie omówione w tym temacie algorytmy sortowania działają wolno przy porządkowaniu dużych zbiorów danych.

Podsumowanie

- Sortowanie bąbelkowe (przez prostą zamianę) polega na porównywaniu sąsiednich elementów ciągu i ewentualnej ich zamianie tak, aby spełniały relację porządkującą.
- W sortowaniu przez wybieranie w kolejnych krokach znajduje się najmniejszy element sortowanego ciągu i przenosi ten element na odpowiednią pozycję ciągu wynikowego (przez zamianę elementów miejscami).
- Sortowanie przez wstawianie polega na wstawianiu kolejnych elementów porządkowanego ciągu w już uporządkowany fragment ciągu.
- Dla algorytmów sortowania prostego operacją dominującą jest porównywanie elementów.
- Dla algorytmów sortowania prostego czasowa złożoność obliczeniowa (zarówno oczekiwana, jak i pesymistyczna) jest kwadratowa: $O(n^2)$.

Zadania

* **1** Napisz program, który uporządkuje liczby w tablicy tak, aby pierwszym elementem tablicy była liczba z najmniejszą cyfrą jedności, a ostatnim – liczba z największą cyfrą jedności. Jeśli dwie liczby mają taką samą cyfrę jedności, uznaj je za równe.

* **2** Napisz program, który posortuje niemalejąco tablicę liczb całkowitych. Jako pierwszy powinien zostać wyznaczony ostatni element tablicy, następnie przedostatni itd. Podczas rozwiązywania zadania skorzystaj z algorytmu sortowania przez wybieranie.

* **3** Zmodyfikuj omówiony w temacie program realizujący algorytm sortowania bąbelkowego tak, aby sprawdzał, czy w danej iteracji wewnętrznej pętli któreś z elementów zostały zamienione miejscami. Algorytm powinien kończyć działanie, jeżeli w danej iteracji nie było ani jednej zamiany elementów.

** **4** Napisz program sortujący liczby całkowite nieujemne według sumy ich cyfr: na pierwszym miejscu powinna się znaleźć liczba, której suma cyfr jest najmniejsza.

** **5** Napisz program sortujący liczby całkowite nieujemne według liczby jedynek w ich zapisie binarnym: na pierwszym miejscu powinna się znaleźć liczba z największą liczbą jedynek w zapisie binarnym.

** **6** Napisz program porządkujący liczby w tablicy tak, aby najpierw znalazły się w niej liczby nieparzyste, a potem liczby parzyste.

** **7** Napisz program, który uporządkuje niemalejąco tablicę liczb, wykorzystując algorytm sortowania przez wstawianie. Miejsce do wstawienia elementu program powinien znajdować metodą przeszukiwania binarnego.
Uwaga: Przesunięcie elementów nadal będzie liniowe.

*** **8** Napisz program, który uporządkuje liczby całkowite w tablicy tak, aby pierwszym elementem tablicy była liczba z najmniejszą cyfrą jedności, a ostatnim – liczba z największą cyfrą jedności. Jeśli dwie liczby mają taką samą cyfrę jedności, uznaj je za równe. Rozwiąż zadanie tak, aby złożoność czasowa algorytmu była nie większa niż $10n$.

*** **9** Napisz program losujący tablicę liczb całkowitych z zakresu od 2 do 1000 oraz sortujący je według najmniejszego czynnika pierwszego. Najpierw w tablicy powinny się znaleźć liczby podzielne przez 2, potem podzielne przez 3 (a niepodzielne przez 2), następnie podzielne przez 5 (a niepodzielne przez 2 i 3) itd.

13. Szyfry przestawieniowe, anagramy

Co mają ze sobą wspólnego słowa „algorytm" i „logarytm"? A słowa „strzałka" i „szkarłat" lub „narty" i „tyran"? Jak łatwo zauważyć, słowa w poszczególnych parach składają się z tych samych liter, są więc anagramami. Sztuka tworzenia wyrazów i zdań z tych samych liter co inne wyrazy i zdania znana jest od starożytności. Niektórzy traktowali ją jak zabawę słowną, inni zaczęli wykorzystywać przestawianie liter w tekście, aby ukryć znaczenie swojego przekazu. W tym temacie zaszyfrujesz tekst, przestawiając litery w tekście jawnym.

Cele lekcji

- Dowiesz się, na czym polegają szyfry przestawieniowe.
- Poznasz przykłady szyfrów przestawieniowych.
- Sprawdzisz, czy dwa słowa są anagramami.
- Wykorzystasz poznane wcześniej algorytmy sortowania i zliczania elementów.

Szyfr przestawieniowy

Szyfry przestawieniowe to szyfry, w których szyfrogram powstaje przez zmianę kolejności liter w tekście jawnym. Szyfrogram składa się więc z tych samych znaków co tekst jawny. Musi istnieć możliwość odszyfrowania wiadomości, a więc określony sposób przestawiania liter, tak aby odbiorca potrafił odwrócić zmiany. Współcześnie szyfry przestawieniowe, podobnie jak **szyfry podstawieniowe**, nie zapewniają bezpieczeństwa danych i nie są stosowane.

Szyfr podstawieniowy, s. 117

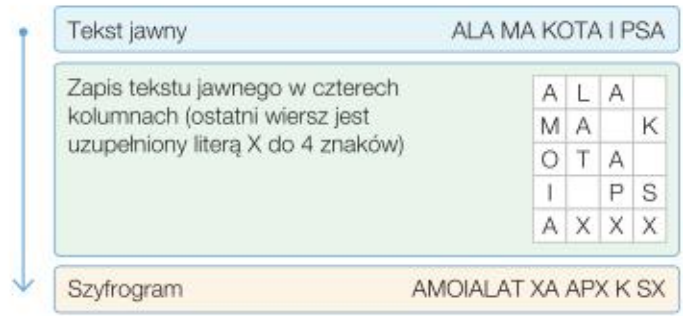
13.1. Przykłady szyfrów przestawieniowych

Jednym z najprostszych sposobów takiego szyfrowania jest przestawianie sąsiednich liter na pozycjach nieparzystych i parzystych (pierwszy znak zamieniamy miejscami z drugim, trzeci z czwartym itd.). Aby poznać treść tekstu jawnego, wystarczy ponownie zastosować ten sam sposób szyfrowania.

Szyfr kolumnowy

Bardziej złożonym przykładem szyfru przestawieniowego jest **szyfr kolumnowy**. Tekst jawny zapisujemy w wierszach, przy czym każdy wiersz składa się z takiej samej (ustalonej przez szyfrującego) liczby znaków. Jeśli ostatni wiersz jest niepełny, można go uzupełnić znakami niewystępującymi w tekście, tak aby przy odszyfrowywaniu wiadomości łatwo było te znaki usunąć. Aby utworzyć szyfrogram, wypisujemy znaki zapisane w kolejnych kolumnach. Liczba znaków w wierszu (odpowiadająca liczbie kolumn) to klucz szyfrowania.

Rysunek 13.1 przedstawia przykład szyfrowania informacji szyfrem kolumnowym z kluczem 4.



Rys. 13.1. Przykład szyfrowania informacji szyfrem kolumnowym z kluczem 4

Aby zaszyfrować tekst tą metodą, nie trzeba pamiętać układu kolumn, wystarczy dopisywać do szyfrogramu znaki w odpowiedniej kolejności.

Napiszemy teraz program, który zaszyfruje szyfrem kolumnowym tekst wczytany z klawiatury. Oto specyfikacja problemu i zapis algorytmu w pseudokodzie:

Dobra rada

Jeśli chcesz szyfrować tylko litery, usuń najpierw z tekstu jawnego wszystkie inne znaki, np. spacje.

Specyfikacja

Dane: `tj` – tekst jawny złożony z wielkich liter alfabetu łacińskiego, `kl` – klucz szyfrowania, liczba całkowita większa od 1 i mniejsza od długości `tj`.

Wynik: `szyfrogram` – tekst jawny `tj` zaszyfrowany szyfrem kolumnowym o liczbie kolumn `kl`.

```
szyfrogram ← ""
dopóki (długość tj) mod kl ≠ 0 wykonuj tj ← tj + 'X'
// uzupełnienie ostatniego wiersza literami X
dla j ← 0, 1, …, kl - 1 wykonuj
// j - numer kolumny
    dla i ← 0, 1, …, (długość tj) div kl - 1
    // i - numer wiersza
    wykonuj szyfrogram ← szyfrogram + tj[i*kl+j]
    // tworzenie szyfrogramu z kolejnych znaków
    // znajdujących się w j-tej kolumnie i i-tym wierszu
```

Dobra rada

Pamiętaj, że ostatni wiersz musisz uzupełnić znakami, które nie występują w tekście jawnym. Założyliśmy, że w tekście jawnym nie będą występowały litery X.

Najpierw tekst jawny zostaje uzupełniony znakami nieznaczącymi tak, aby liczba znaków w nim występujących była wielokrotnością liczby kolumn (wartości klucza). Następnie przeglądamy znaki tekstu jawnego w kolejności, w jakiej będą występować w szyfrogramie (znaki w kolejnych kolumnach). Indeks znaku określa wzór `i*kl+j`, gdzie `i` jest numerem wiersza, `kl` – kluczem, a `j` – numerem kolumny. Zwróć uwagę, że wiersze i kolumny są numerowane od zera.

Oto kod źródłowy funkcji realizującej powyższy algorytm.

Fragment kodu źródłowego programu szyfrującego napis szyfrem kolumnowym – funkcja `Przestaw` (odczytywanie znaków z kolejnych kolumn)

```
1.  string Przestaw(string tj, int kl)
2.  {
3.      string szyfrogram="";
4.      int i, j;
5.      while (tj.size()%kl!=0) tj=tj+'X';
6.      for (j=0;j<kl;j++)
7.          for (i=0;i<tj.size()/kl;i++)
8.              szyfrogram+=tj[i*kl+j];
9.      return szyfrogram;
10. }
```

Zwróć uwagę na instrukcję w linii 8. Stosujemy w niej skrócony zapis instrukcji powiększenia zmiennej o określoną wartość (zapis `+=`). Instrukcja w linii 8 odpowiada instrukcji:

```
szyfrogram = szyfrogram + tj[i*kl+j];
```

Warto wiedzieć

Zapis `zmienna+=wartość` oznacza powiększenie zmiennej o podaną wartość. W podobny sposób można przedstawiać inne działania arytmetyczne. Na przykład `x*=2` jest skróconym zapisem instrukcji `x=x*2`.

Ćwiczenie 1

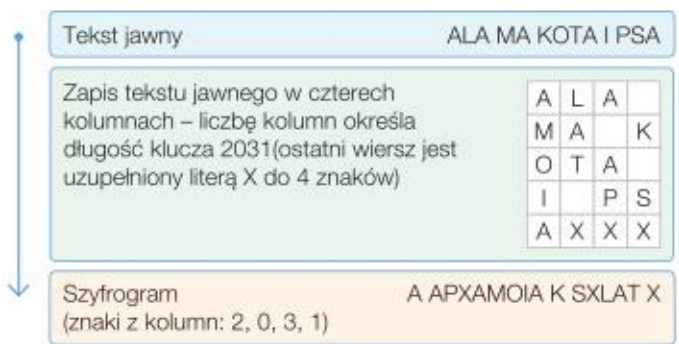
Napisz program szyfrujący szyfrem kolumnowym tekst wczytany z klawiatury. Wykorzystaj w tym programie zdefiniowaną powyżej funkcję `Przestaw`. Pamiętaj, aby tekst wczytać za pomocą funkcji `getline`, ponieważ może on zawierać spacje.

Dobra rada

W przypadku, gdy kolumn jest więcej niż 10 (kolumny numerowaliśmy od 0 do 9), możesz do ich oznaczenia użyć liter – podobnie jak w pozycyjnych systemach liczbowych o podstawie większej od 10.

Tekst zaszyfrowany w omówiony sposób można bardzo łatwo odszyfrować, używając tej samej funkcji.

Istnieją różne odmiany szyfru kolumnowego. W wersji przedstawionej wcześniej znaki były odczytywane kolumnami po kolei, zaczynając od pierwszej kolumny po lewej. Można jednak ustalić inną kolejność odczytywania znaków. W takiej sytuacji kluczem będzie np. napis złożony z cyfr określających kolejność przeglądania kolumn, a liczbę kolumn będzie wyznaczać długość klucza. Rysunek 13.2 przedstawia przykład szyfrowania informacji szyfrem kolumnowym z kluczem 2031.



Rys. 13.2. Przykład szyfrowania informacji szyfrem kolumnowym z kluczem 2031

Specyfikacja problemu szyfru kolumnowego, w którym odczytujemy znaki w kolejności określonej przez klucz, jest następująca.

Specyfikacja

Dane: `tj` – tekst jawny złożony z wielkich liter alfabetu łacińskiego, `kl` – napis złożony z niepowtarzających się cyfr dziesiętnych mniejszych od długości klucza `kl`.

Wynik: `szyfrogram` – tekst jawny `tj` zaszyfrowany szyfrem kolumnowym o liczbie kolumn i ich kolejności określonych przez klucz `kl`.

Oto kod źródłowy funkcji szyfrującej napis zgodnie ze specyfikacją:

```
1.  string Przestaw(string tj, string kl)
2.  {
3.      string szyfrogram="";
4.      int i, j;
5.      while (tj.size()%kl.size()!=0) tj=tj+'X';
6.      for (j=0;j<kl.size();j++)
7.          for (i=0;i<tj.size()/kl.size();i++)
8.              szyfrogram+=tj[i*kl.size()+kl[j]-'0'];
9.      return szyfrogram;
10. }
```

Fragment kodu źródłowego programu szyfrującego napis szyfrem kolumnowym – zmodyfikowana funkcja `Przestaw` (kolejność odczytywania kolumn określają cyfry klucza)

Liczbę kolumn wyznacza długość klucza, a nie jego wartość. Stąd zmiany w liniach 5–8 przy odwołaniu do liczby kolumn. Numer kolejnej kolumny jest określony teraz nie przez wartość indeksu `j`, ale przez wartość `kl[j]`. Ponieważ `kl[j]` jest znakiem (program odczytuje go jako kod ASCII cyfry), w linii 8 odejmujemy kod ASCII cyfry 0.

Ćwiczenie 2

Napisz program szyfrujący szyfrem kolumnowym tekst wczytany z klawiatury. Kolejność odczytywania znaków z kolumn określają cyfry tworzące klucz, a liczbę kolumn wskazuje długość klucza.

Zapamiętaj

Szyfry przestawieniowe to szyfry, w których szyfrogram powstaje przez zamianę kolejności znaków tekstu jawnego. W szyfrogramie występują wszystkie znaki tekstu jawnego.

13.2. Sprawdzamy, czy dwa słowa są anagramami

Anagram to słowo lub zdanie, które powstało przez przestawienie liter innego słowa lub zdania. Na potrzeby tego zagadnienia przyjmiemy, że słowo to dowolny ciąg liter, a anagram to napis powstały z innego słowa przez przestawianie liter. Sprawdzimy, czy dwa słowa są anagramami.

Anagram

Warto wiedzieć

Za wynalazcę anagramu uznaje się greckiego poetę Likofrona, który żył w III w. p.n.e. Jest on autorem np. anagramów *Ptolemaios* (Ptolemeusz) i *apo melitos* (z miodu).

Oto specyfikacja problemu:

Specyfikacja

Dane: s1, s2 – napisy złożone z wielkich liter alfabetu łacińskiego.

Wynik: wartość logiczna **prawda**, jeśli s1 i s2 są anagramami, **fałsz** – w przeciwnym wypadku.

Zauważ, że jeśli napisy są różnej długości, odpowiedź jest oczywista – nie są one anagramami. Jeśli napisy są tej samej długości, można próbować zmieniać kolejność liter w jednym napisie tak, aby otrzymać drugi napis. Działanie algorytmu polegałoby na generowaniu kolejnych **permutacji** liter jednego słowa i porównywaniu otrzymanej permutacji z drugim słowem. Ten algorytm jest czasochłonny, ponieważ liczba wszystkich permutacji jest równa $n!$, gdzie n oznacza liczbę liter słowa.

Permutacja zbioru, s. 126

Algorytm wykorzystujący sortowanie

Aby sprawdzić, czy dwa słowa są anagramami, można posortować alfabetycznie litery w obu napisach, a następnie porównać znaki znajdujące się na tych samych pozycjach w obu wyrazach. Jeśli znaki są takie same, analizowane wyrazy są anagramami. Zapis tego algorytmu w postaci funkcji w pseudokodzie może być następujący:

```
funkcja Anagramy(s1,s2)
    jeśli długość s1 ≠ długość s2 to
        zwróć fałsz i zakończ
    Sortuj(s1)
    Sortuj(s2)
    zwróć s1 = s2 i zakończ
```

Wynikiem funkcji jest wartość logiczna: rezultat porównania napisów s1 i s2. Do posortowania liter słowa można użyć dowolnej metody spośród opisanych w temacie dotyczącym algorytmów sortowania. W tym przykładzie wykorzystamy **algorytm sortowania przez wybieranie**. Oto kody źródłowe funkcji Sortuj i Anagramy:

Sortowanie przez wybieranie, s. 174–175

Fragment kodu źródłowego programu sprawdzającego, czy dwa słowa są anagramami – funkcja sortująca litery w napisie

```
1.  void Sortuj(string &s)
2.  {
3.      int i, j, m;
4.      char ch;
5.      for (i=0;i<s.size()-1;i++)
6.      {
7.          m=i;
8.          for (j=i+1;j<s.size();j++)
9.              if (s[j]<s[m]) m=j;
10.         ch=s[i]; s[i]=s[m]; s[m]=ch;
11.     }
12. }
```

```
1. bool Anagramy(string s1, string s2)
2. {
3.     if (s1.size()!=s2.size()) return false;
4.     Sortuj(s1); Sortuj(s2);
5.     return (s1==s2);
6. }
```

Fragment kodu źródłowego programu sprawdzającego, czy dwa słowa są anagramami – funkcja porównująca znaki napisów

Zwróć uwagę na nagłówek funkcji `Sortuj`. Domyślnie parametry typu `string`, podobnie jak parametry typów prostych, **przekazywane są przez wartość**. Są zatem chronione, czyli operacje wewnątrz funkcji wykonywane są na kopii parametru. Zmiany nie są zachowywane po zakończeniu działania funkcji. Aby zachować zmiany, należy **przekazać parametr przez referencję**.

Przekazanie parametru przez wartość, s. 108

Przekazanie parametru przez referencję, s. 161

Ćwiczenie 3

Napisz program wczytujący dwa słowa i wypisujący „TAK", gdy podane słowa są anagramami, „NIE" – w przeciwnym wypadku.

Algorytm wykorzystujący zliczanie

Innym sposobem sprawdzenia, czy słowa są anagramami, jest policzenie wystąpień poszczególnych liter w obu napisach, a następnie porównanie liczby tych wystąpień. Zapis algorytmu w pseudokodzie może być następujący:

```
funkcja Anagramy(s1,s2)
    jeśli długość s1 ≠ długość s2 to
        zwróć fałsz i zakończ
    Zlicz(s1,Ile1)
    Zlicz(s2,Ile2)
    dla i ← 0, 1, ..., długość alfabetu - 1 wykonuj
        jeśli Ile1[i] ≠ Ile2[i] to
            zwróć fałsz i zakończ
    zwróć prawda i zakończ
```

W podanym algorytmie wykorzystujemy funkcję `Zlicz`. Jej zadaniem jest wyznaczenie liczby wystąpień poszczególnych liter w danym napisie (pierwszy parametr). Wynikiem funkcji jest tablica (drugi parametr), w której na kolejnych pozycjach znajdą się liczby wystąpień kolejnych liter alfabetu. Tablica będzie się składać z 26 elementów – zgodnie ze specyfikacją w napisie mogą wystąpić tylko wielkie litery alfabetu łacińskiego. Na początku definicji funkcji należy wyzerować wartości elementów tablicy. Do liter słowa odwołujemy się za pośrednictwem kodów ASCII, natomiast tablice są indeksowane od 0. Należy przesunąć wartości indeksu tak, aby element zerowy tablicy odpowiadał literze A. Oto zapis funkcji `Zlicz` w pseudokodzie oraz kody źródłowe funkcji `Zlicz` i `Anagramy`.

```
funkcja Zlicz(s,Ile)
    dla i ← 0, 1, …, długość alfabetu - 1 wykonuj
        Ile[i] ← 0
    dla i ← 0, 1, …, długość s - 1 wykonuj
        powiększ Ile[kod_ASCII(s[i])-kod_ASCII('A')]
```

Fragment kodu źródłowego programu sprawdzającego, czy dwa słowa są anagramami – funkcje zliczająca znaki w napisie oraz porównująca otrzymane wyniki

```
1.  void Zlicz(string s, int Ile[])
2.  {
3.      for (int i=0;i<26;i++) Ile[i]=0;
4.      for (int i=0;i<s.size();i++) Ile[s[i]-'A']++;
5.  }
6.
7.  bool Anagramy(string s1, string s2)
8.  {
9.      int Ile1[26], Ile2[26];
10.     if (s1.size()!=s2.size()) return false;
11.     Zlicz(s1,Ile1); Zlicz(s2,Ile2);
12.     for (int i=0;i<26;i++)
13.         if (Ile1[i]!=Ile2[i]) return false;
14.     return true;
15. }
```

Dobra rada

W języku C++ odwołanie do znaku oznacza odwołanie do jego kodu ASCII, więc nie musisz dokonywać konwersji znaku na jego kod.

Ćwiczenie 4

Napisz program sprawdzający, czy dwa słowa są anagramami. Program powinien realizować algorytm zliczający poszczególne litery w napisie.

Zastanówmy się nad czasową złożonością obliczeniową omówionych algorytmów. Złożoność pierwszego algorytmu zależy od użytej metody sortowania. Ponieważ zastosowaliśmy algorytm sortowania przez wybieranie, złożoność pierwszego algorytmu jest kwadratowa względem długości słowa. Drugi algorytm przegląda trzykrotnie tablicę z licznikami oraz dwukrotnie słowo. Jego złożoność czasowa jest liniowa względem większej z wartości: długość słowa, liczba liter alfabetu. W przypadku krótszych słów pierwszy algorytm wykona mniej operacji.

A to ciekawe

Pomysł na nazwę

Niektórzy twórcy zapisują swoje nazwiska wspak. Tak utworzony pseudonim nazywamy ananimem (jest to szczególny typ anagramu). Na przykład Stefan Żeromski posługiwał się pseudonimem Stefan Iksmoreż (ananim nazwiska). W podobny sposób są tworzone różne inne nazwy, np. nazwa robaka komputerowego Nimda powstała przez odwrócenie porządku liter w słowie „admin".

Podsumowanie

- Podczas szyfrowania informacji szyfrem przestawieniowym zmienia się kolejność znaków w tekście jawnym. W szyfrogramie występują wszystkie znaki tekstu jawnego.
- Przykładem szyfru przestawieniowego jest szyfr kolumnowy.
- Aby zaszyfrować tekst jawny szyfrem kolumnowym, zapisujemy tekst jawny w wierszach o takiej samej długości (ustalonej przez szyfrującego), a następnie odczytujemy kolejno znaki z poszczególnych kolumn (kolejność przeglądania kolumn należy wcześniej ustalić).
- Aby sprawdzić, czy dwa słowa są anagramami, można posortować znaki w słowach i porównać znaki na kolejnych pozycjach lub policzyć wystąpienia poszczególnych znaków w obu słowach i porównać liczby tych wystąpień.

Zadania

* **1** Napisz program wczytujący trzy słowa i sprawdzający, czy są one anagramami.

* **2** Napisz program, który wczyta dowolny tekst z klawiatury i zaszyfruje go szyfrem przestawieniowym polegającym na zamianie miejscami znaków na pozycjach parzystych i nieparzystych.

** **3** Napisz program, który sprawdzi, czy dwa słowa są anagramami. Program powinien wczytać dane z pliku tekstowego. W zapisie słów mogą wystąpić litery z polskiego alfabetu.

** **4** Napisz program wczytujący dowolny tekst z klawiatury oraz liczbę będącą kluczem szyfrowania (liczbę kolumn). Program powinien szyfrować wczytany tekst szyfrem przestawieniowym polegającym na spiralnym odczytaniu tekstu jawnego zapisanego wierszami zgodnie z poniższym przykładem.
Przykład:
Tekst jawny: ALA MA KOTA I PSA
Klucz: 4
Szyfrogram: K SXXXAIOMALA AP TA

** **5** Napisz program szyfrujący tekst szyfrem kolumnowym z kluczem określającym liczbę kolumn i kolejność ich przeglądania tak, aby mogło być więcej niż 10 kolumn.

*** **6** Napisz program, który z pliku tekstowego przekazanego przez nauczyciela (np. *szyfrogram_kolumnowy.txt*) wczyta jednowierszowy szyfrogram i klucz szyfrowania. Szyfrogram powstał przez zaszyfrowanie tekstu jawnego szyfrem kolumnowym. Klucz (drugi wiersz pliku) jest napisem złożonym z niepowtarzających się cyfr dziesiętnych mniejszych od długości klucza i określa liczbę kolumn oraz kolejność ich przeglądania. Wynikiem działania programu powinien być tekst jawny wyświetlony na ekranie.

14. Sito Eratostenesa

W XIX w. w Kalifornii wybuchła gorączka złota. Jedną z najbardziej popularnych metod stosowanych przez poszukiwaczy cennego kruszcu było przesiewanie minerałów na sicie. Samorodki złota, które są cięższe, opadały na dno sita, a lżejsze piasek i muł były odsiewane. Okazuje się, że podobną metodę można zastosować do poszukiwania liczb pierwszych w zbiorze liczb całkowitych dodatnich. Z tego tematu dowiesz się, jak to zrobić.

Cele lekcji

- Zrozumiesz, jak działa algorytm sita Eratostenesa.
- Napiszesz program realizujący algorytm sita Eratostenesa.
- Poznasz złożoność obliczeniową algorytmu sita Eratostenesa.

Poznaliśmy wcześniej algorytmy, które pozwalają sprawdzić, czy dana liczba całkowita dodatnia jest liczbą pierwszą. Zajmiemy się teraz problemem wyznaczenia wszystkich liczb pierwszych mniejszych lub równych danej liczbie całkowitej większej od 1. Jednym z rozwiązań jest sprawdzanie, czy kolejne liczby z badanego zakresu są pierwsze, czyli wielokrotne wykorzystanie znanego już algorytmu.

Ćwiczenie 1

Oszacuj złożoność czasową algorytmu, który wyznaczałby liczby pierwsze mniejsze lub równe liczbie całkowitej n większej od 1 metodą naiwną, czyli sprawdzałby pierwszość kolejnych liczb całkowitych większych od 1 i mniejszych lub równych n.

14.1. Jak działa sito Eratostenesa?

Omówimy teraz metodę wyznaczania liczb pierwszych mniejszych lub równych liczbie n zaproponowaną już w starożytności przez greckiego matematyka Eratostenesa z Cyreny. Polega ona na usunięciu ze zbioru liczb całkowitych z zakresu od 2 do n wszystkich liczb złożonych. W ten sposób w zbiorze pozostaną tylko liczby pierwsze. Metodę tę nazwano **sitem Eratostenesa**.

Sito Eratostenesa

W praktyce algorytm polega na usuwaniu (odsiewaniu) ze zbioru liczb $\{2, \ldots, n\}$ wszystkich liczb złożonych, które są wielokrotnościami kolejnych analizowanych liczb ze zbioru, począwszy od 2.

Algorytm sita Eratostenesa

Prześledzimy teraz działanie tego algorytmu dla $n = 50$. Wyznaczymy zatem wszystkie liczby pierwsze z zakresu od 2 do 50.

Algorytm sita Eratostenesa

Zbiór wyjściowy składa się z liczb całkowitych z zakresu od 2 do 50:

{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50}

Krok 1 Usuwamy wielokrotności liczby 2 większe od 2. Otrzymujemy zbiór:
{2, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49}

Krok 2 Usuwamy wielokrotności liczby 3 większe od 3. Otrzymujemy zbiór:
{2, 3, 5, 7, 11, 13, 17, 19, 23, 25, 29, 31, 35, 37, 41, 43, 47, 49}

Krok 3 Usuwamy wielokrotności liczby 5 większe od 5. Otrzymujemy zbiór:
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 49}

Krok 4 Usuwamy wielokrotności liczby 7 większe od 7. Otrzymujemy zbiór:
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47}

Otrzymany zbiór zawiera wszystkie liczby pierwsze z zakresu od 2 do 50.

Zwróć uwagę, że niektóre wielokrotności danej liczby zostały już usunięte w poprzednich krokach jako wielokrotności mniejszej liczby. Na przykład liczby 6 i 10 zostały usunięte jako wielokrotności liczby 2.

Dla liczb: 2, 3, 5, 7 pierwszymi usuwanymi wielokrotnościami były liczby: 4, 9, 25, 49, które są ich kwadratami. Jest to ogólna zasada – w każdym kroku algorytmu pierwszą usuwaną wielokrotnością jest kwadrat liczby, której krotności usuwamy. Algorytm kończy działanie, jeśli kwadrat rozpatrywanej liczby jest większy od prawego końca przedziału (liczby n). Zauważ, że każda liczba, której wielokrotności usuwamy, jest liczbą pierwszą.

Warto wiedzieć

Eratostenes z Cyreny (276–194 p.n.e.) to pochodzący ze starożytnej Grecji matematyk, astronom, geograf, filozof i poeta. Jest autorem doświadczenia uznawanego za jeden z dziesięciu największych eksperymentów z fizyki – pomiaru obwodu Ziemi. Oszacował także odległość Ziemi od Księżyca i Słońca.

Ćwiczenie 2

Prześledź działanie algorytmu sita Eratostenesa dla $n = 100$.

Zapamiętaj

Algorytm sita Eratostenesa służy do znajdowania liczb pierwszych z zakresu od 2 do n. Z podanego zbioru usuwamy wielokrotności kolejnych liczb, a pierwszą usuwaną krotnością jest kwadrat danej liczby. Kiedy kwadrat rozpatrywanej liczby jest większy od prawego końca przedziału, algorytm kończy działanie.

14.2. Implementacja algorytmu sita Eratostenesa

Określmy specyfikację problemu i zapiszmy w pseudokodzie algorytm znajdowania liczb pierwszych metodą sita Eratostenesa.

Specyfikacja

Dane: n – liczba całkowita, $2 \leqslant n \leqslant 1\,000\,000$.

Wynik: `Pierwsze` – zbiór liczb pierwszych mniejszych lub równych n.

```
Pierwsze ← {2, ..., n}
d ← 2
dopóki d * d ⩽ n wykonuj
  dla i ← d, d + 1, ..., n div d wykonuj
    Pierwsze ← Pierwsze - {i * d}
  d ← najmniejszy element zbioru Pierwsze, który jest
      większy od dotychczasowej wartości d
```

Dobra rada

Wewnętrzną pętlę możesz również zapisać następująco:

```
i ← d
dopóki i * d ⩽ n wykonuj
  Pierwsze ←
  Pierwsze - {i * d}
  i ← i + 1
```

Po jednej iteracji zewnętrznej pętli zostają usunięte wielokrotności aktualnie rozpatrywanej liczby (zmienna `d`, wartość początkowa 2), następnie zmienna `d` przyjmuje kolejną wartość. Pętla jest powtarzana, dopóki kwadrat liczby jest mniejszy lub równy prawemu końcowi przedziału (`n`), w którym poszukujemy liczb pierwszych.

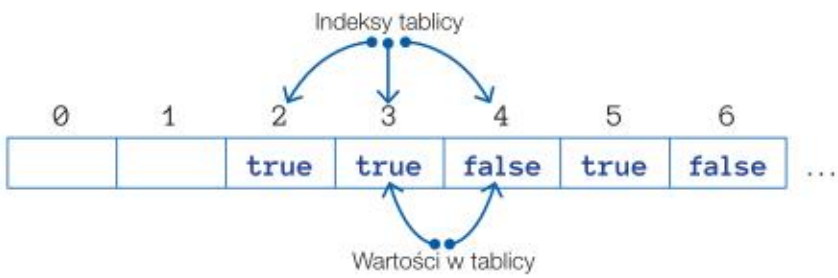
W wewnętrznej pętli usuwane są `i`-te wielokrotności wartości zmiennej `d`. Ponieważ iloczyn `i * d` musi być mniejszy lub równy `n`, największa wartość `i` jest równa wynikowi dzielenia całkowitego `n` przez `d` (`n div d`).

Zwróć uwagę, że następuje próba usunięcia wszystkich liczb postaci `i * d`, a niektóre z nich były już usunięte wcześniej jako wielokrotności mniejszej liczby (np. liczba 45 została usunięta jako wielokrotność liczby 3, ale algorytm wykonuje tę operację także dla liczby 5).

Do reprezentowania w programie zbioru liczb najprościej będzie wykorzystać tablicę. Zauważ, że nie musimy w niej przechowywać samych liczb. Wystarczy, że każdy element tablicy będzie zawierał informację logiczną (`true` lub `false`) określającą, czy liczba reprezentowana przez odpowiadający mu indeks tablicy jest liczbą pierwszą czy złożoną. Rysunek 14.1 przedstawia fragment tablicy `Pierwsze` z wartościami opisującymi początkowe liczby zbioru.

Dobra rada

Pamiętaj, że w języku C++ wartość `false` jest reprezentowana przez liczbę 0, a wartość `true` – przez liczbę 1.



Rys. 14.1. Fragment tablicy `Pierwsze` z zaznaczeniem liczb pierwszych i złożonych

Tablice w języku C++ są indeksowane zawsze od zera, więc zerowy i pierwszy element tablicy nie zostaną wykorzystane. Ostatni indeks w tablicy n-elementowej ma wartość $n - 1$, dlatego wygodniej będzie szukać liczb pierwszych mniejszych od n.

Oto zmodyfikowana specyfikacja problemu znajdowania liczb pierwszych, w którym wykorzystywana jest tablica wartości logicznych:

Specyfikacja

Dane: n – liczba całkowita, $2 \leqslant n \leqslant 1\,000\,000$.

Wynik: tablica wartości logicznych `Pierwsze[0..n-1]`, `Pierwsze[i]` ma wartość **`prawda`**, gdy `i` jest liczbą pierwszą, **`fałsz`** – w przeciwnym przypadku.

Ćwiczenie 3

Zapisz w pseudokodzie algorytm sita Eratostenesa zgodny z powyższą specyfikacją.

Kod źródłowy funkcji znajdującej liczby pierwsze metodą sita Eratostenesa zgodnie z powyższą specyfikacją może być taki jak poniżej. Rozmiar tablicy `Pierwsze` określa stała `N`, zdefiniowana przed funkcją.

```
1.  void SitoEratostenesa(bool Pierwsze[])
2.  {
3.      Pierwsze[2]=true;
4.      for (int i=3;i<N;i++)
5.          Pierwsze[i]=(i%2==1);
6.      int d=3;
7.      while (d*d<N)
8.      {
9.          for (int i=d;i*d<N;i+=2)
10.             Pierwsze[i*d]=false;
11.         do
12.             d+=2;
13.         while (!Pierwsze[d]);
14.     }
15. }
```

Fragment kodu źródłowego programu wyszukującego liczby pierwsze z zakresu od 2 do *N* – definicja funkcji realizującej algorytm sita Eratostenesa

W definicji funkcji wykorzystujemy fakt, że liczba 2 jest jedyną parzystą liczbą pierwszą, a kolejne liczby parzyste są złożone. Uwzględniamy to już podczas inicjowania tablicy `Pierwsze` (linia 1).

Elementom tablicy o parzystych indeksach większych od 2 przypisujemy wartość **false**, a pozostałym – wartość **true** (linie 4–5). Jedna iteracja głównej pętli **while** (linie 7–14) odpowiada za nadanie wartości **false** tym elementom tablicy, których indeksy są wielokrotnościami aktualnie rozpatrywanej liczby – wartości zmiennej d (linie 9–10). Aby wyznaczyć kolejną liczbę pierwszą, korzystamy z **instrukcji iteracji (pętli) do while**. Pętla ta najpierw wykonuje instrukcję, a potem sprawdza warunek powtarzania pętli. Jej składnia w języku C++ jest następująca:

Instrukcja iteracji (pętla) do while

```
do
    instrukcja;
while (warunek);
```

Warto wiedzieć

Zamiast pętli **do while** można użyć pętli **while**. Pętlę **do while** zastępujemy wówczas instrukcjami:
`instrukcja;`
`while (warunek) instrukcja;`

Instrukcja w pętli **do while** jest wykonywana, dopóki warunek przyjmuje wartość logiczną **prawda**. Ponieważ warunek jest sprawdzany po wykonaniu instrukcji, pętla wykona się co najmniej raz. Instrukcja wykonywana w pętli może być instrukcją złożoną, a więc blokiem instrukcji ujętych w nawiasy klamrowe.

Aby określić następną liczbę, której wielokrotności będziemy oznaczać przez **false**, najpierw określamy nową wartość zmiennej d, a potem sprawdzamy, czy jest ona liczbą pierwszą (linie 11–13). Co druga wielokrotność każdej liczby jest parzysta, więc usuwając krotności danej liczby, zwiększamy wartość zmiennej sterującej pętli **for** o 2 (linia 9). Analogicznie kiedy poszukujemy kolejnej liczby, której wielokrotności będziemy usuwać, zwiększamy wartość zmiennej d o 2. Pętla **do while** jest wykonywana, dopóki kolejne wartości zmiennej d są liczbami złożonymi (linie 11–13).

Dobra rada

Zapis `i+=2` odpowiada zapisowi `i=i+2`, który powoduje zwiększenie wartości zmiennej i o 2.

Pamiętaj, że tablice są **przekazywane do funkcji przez wskaźnik**. Wartości tablicy `Pierwsze` określone w funkcji `SitoEratostenesa` będą więc zachowane po wykonaniu funkcji. Pętla wypisująca liczby pierwsze po wywołaniu funkcji `SitoEratostenesa` może być taka jak poniżej. Wypisujemy te indeksy tablicy, które w tablicy mają wartość **true**.

Przekazanie parametru przez wskaźnik, s. 152

Fragment kodu źródłowego programu wyszukującego liczby pierwsze z zakresu od 2 do *N* – pętla odpowiadająca za wypisanie liczb pierwszych z zakresu od 2 do *N*

```
1.  for (int i=2;i<N;i++)
2.      if (Pierwsze[i]) cout<<i<<" ";
```

Ćwiczenie 4

Napisz i przetestuj program znajdujący liczby pierwsze z przedziału $[2; n]$ metodą sita Eratostenesa.

Ocena złożoności obliczeniowej algorytmu sita Eratostenesa

Zwróć uwagę, że w algorytmie sita Eratostenesa operacją dominującą jest usuwanie liczby (instrukcja `Pierwsze[i*d]=false` oraz inicjowanie tablicy). Można oszacować, że algorytm wykona tych operacji nie więcej niż $n \cdot \log_2 n$. Jest to liczba znacznie mniejsza niż $n \cdot \sqrt{n}$ prób

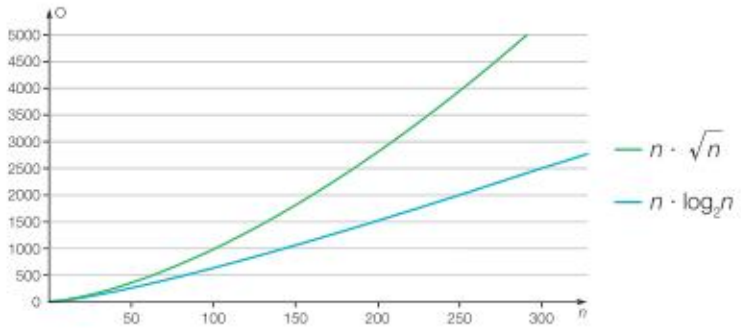
sprawdzania podzielności w algorytmie naiwnym, badającym pierwszość każdej liczby niezależnie (tabela 14.1 i rys. 14.2).

Warto wiedzieć

Często obliczenia potrzebne do oszacowania złożoności algorytmu są obszerne i skomplikowane, dlatego ich nie przytaczamy.

Liczba elementów zbioru (n)	Przybliżona liczba operacji	
	$n \cdot \log_2 n$	$n \cdot \sqrt{n}$
10	33	32
100	660	1000
1000	10000	31600
10 000	133000	1 000 000
100 000	1660000	31620000

Tabela 14.1. Liczba operacji dla złożoności algorytmu rzędu $n \cdot \log_2 n$ i $n \cdot \sqrt{n}$



Rys. 14.2. Wykresy funkcji reprezentujących złożoności $n \cdot \log_2 n$ i $n \cdot \sqrt{n}$

Złożoność czasowa algorytmu sita Eratostenesa jest więc równa $O(n \cdot \log n)$. Złożoność pamięciowa wynosi $O(n)$, ponieważ algorytm sita Eratostenesa wykorzystuje tablicę n-elementową.

Rozmiar tablicy można znacznie zredukować, np. jeśli będziemy w niej przechowywać tylko informacje o liczbach nieparzystych. Wówczas i-ty element tablicy pamiętałby informację o liczbie $2i + 1$.

Warto wiedzieć

Złożoność czasową algorytmu sita Eratostenesa można oszacować dokładniej. Okazuje się, że wynosi ona $O(n \cdot \log(\log n))$.

A to ciekawe

W poszukiwaniu liczb pierwszych

Liczby pierwsze, które można zapisać w postaci $2^n - 1$, są nazywane liczbami pierwszymi Mersenne'a (od nazwiska francuskiego matematyka). W poszukiwanie kolejnych takich liczb zaangażowani są uczestnicy projektu GIMPS (ang. *Great Internet Mersenne Prime Search*), którzy użyczają w tym celu mocy obliczeniowej swoich komputerów. Dzięki zastosowaniu obliczeń rozproszonych uzyskuje się łączną moc, która odpowiada pracy dziesiątków tysięcy lat pojedynczego procesora. W czasie powstawania tego podręcznika największą liczbą pierwszą Mersenne'a jest $2^{82589933} - 1$.

Podsumowanie

- Algorytm sita Eratostenesa pozwala wyznaczyć liczby pierwsze z określonego przedziału.
- Algorytm nie bada podzielności liczb, lecz usuwa (oznacza jako liczby złożone) wielokrotności kolejnych liczb zbioru, większe od tych liczb.
- Najmniejszą wielokrotnością, którą algorytm usuwa, jest kwadrat rozpatrywanej liczby.
- Algorytm kończy działanie, gdy kwadrat liczby, której krotności ma usunąć, jest większy od prawego końca przedziału.
- Liczba operacji wykonywanych przez algorytm sita Eratostenesa jest znacznie mniejsza niż podczas sprawdzania pierwszości każdej liczby niezależnie.

Zadania

* **1** Napisz program, który wyznaczy liczby pierwsze w przedziale liczbowym podanym przez użytkownika.

* **2** Napisz program wypisujący pary liczb bliźniaczych nie większych od n, $2 \leqslant n \leqslant 1\,000\,000$. W rozwiązaniu wykorzystaj algorytm sita Eratostenesa.
Uwaga: Liczby bliźniacze to liczby pierwsze, których różnica wynosi 2. Liczbami bliźniaczymi są np. pary: 3 i 5, 5 i 7, 11 i 13.

** **3** Program utworzony w tym temacie zmodyfikuj tak, aby w tablicy była przechowywana wyłącznie informacja o liczbach nieparzystych (czyli rozmiar tablicy był dwukrotnie mniejszy).

** **4** Program wypisujący liczby półpierwsze nie większe od n, $2 \leqslant n \leqslant 1000$. Wypisywane liczby nie muszą być uporządkowane. W rozwiązaniu wykorzystaj algorytm sita Eratostenesa.
Uwaga: Liczba półpierwsza to liczba będąca iloczynem dwóch liczb pierwszych. Liczbami półpierwszymi są np.: 4, 6, 9, 10, 14, 15.

*** **5** Napisz program realizujący algorytm sita Eratostenesa z użyciem szablonu struktury danych **set** z biblioteki STL. Struktura **set** reprezentuje w pamięci komputera zbiór.
Wskazówka: Żeby móc korzystać z typu **set**, należy dodać dyrektywę `#include <set>`.
Do dodania i usuwania elementów zbioru możesz wykorzystać metody **insert** i **erase**:

```
set<int> zbior;
zbior.insert(liczba);
zbior.erase(liczba);
```

Do pobrania kolejnego elementu zbioru możesz wykorzystać instrukcję:

```
liczba=*zbior.upper_bound(liczba);
```

Do wypisania elementów zbioru należy użyć zmiennej typu **iterator**:

```
set<int>::iterator it;
for(it=zbior.begin();it!=zbior.end();it++) cout<<*it<<" ";
```

15. Szukamy różnych podciągów

Czasami zachodzi potrzeba znalezienia fragmentu danych spełniającego pewne warunki. Na przykład trener zapisuje wyniki zawodnika osiągane podczas treningów, aby się dowiedzieć, kiedy rezultaty są coraz lepsze, a kiedy następuje spadek formy. Dużą popularnością cieszą się aplikacje, za pomocą których możemy monitorować swoją aktywność (sen, ruch, wysiłek fizyczny itp.), a potem np. sprawdzać, jaki był najdłuższy okres regularnych treningów albo kiedy spaliśmy najkrócej. W tym temacie zajmiemy się poszukiwaniem danych spełniających określone warunki.

Cele lekcji

- Obliczysz długość najdłuższego spójnego podciągu niemalejącego.
- Wyznaczysz najdłuższy spójny podciąg niemalejący.
- Poznasz i porównasz różne algorytmy znajdowania maksymalnej sumy elementów podciągu spójnego.
- Znajdziesz podciąg spójny o maksymalnej sumie elementów.

Podciąg to wybrane elementy ciągu wyjściowego zachowujące kolejność występowania. Na przykład liczby 3, 2, 5, 7 tworzą podciąg ciągu 3, 1, 2, 5, 7, 4. Jeśli elementy podciągu występują w ciągu wyjściowym obok siebie, to taki podciąg nazywamy **podciągiem spójnym**. Na przykład ciąg 1, 2, 5 jest podciągiem spójnym ciągu 3, 1, 2, 5, 7, 4. W tym temacie będziemy poszukiwać tylko podciągów spójnych.

Podciąg

Podciąg spójny

15.1. Szukamy długości najdłuższego spójnego podciągu niemalejącego

W **podciągu niemalejącym** każdy kolejny element jest większy lub równy poprzedniemu. Na przykład w ciągu 3, 1, 2, 5, 7, 4 możemy wyróżnić następujące spójne podciągi niemalejące:

- 1, 2
- 1, 2, 5
- 1, 2, 5, 7
- 2, 5
- 2, 5, 7
- 5, 7

Podciąg niemalejący

W powyższym przykładzie najdłuższy spójny podciąg niemalejący tworzą liczby: 1, 2, 5, 7. Jego długość jest równa 4.

Zapiszmy specyfikację problemu znajdowania długości najdłuższego spójnego podciągu niemalejącego.

Specyfikacja

Dane: `A[0..n-1]` – tablica n liczb całkowitych.

Wynik: `maks_dl` – długość najdłuższego spójnego podciągu niemalejącego w tablicy `A`.

Warto wiedzieć

Długość najdłuższego podciągu niemalejącego będzie równa jeden, gdy ciąg wyjściowy jest malejący.

Podciągi jednoelementowe potraktujemy jako podciągi niemalejące. Może się więc zdarzyć, że szukana długość będzie równa jeden. Może być też równa długości ciągu wyjściowego.

Algorytm poszukiwania długości najdłuższego spójnego podciągu niemalejącego jest podobny do **znajdowania maksimum w tablicy**. Długość aktualnie badanego fragmentu ciągu będziemy pamiętać w zmiennej `akt_dl`, a największą dotychczas znalezioną długość podciągu spójnego – w zmiennej `maks_dl`. Ponieważ każdy element ciągu tworzy podciąg jednoelementowy, początkowe wartości zmiennych `akt_dl` i `maks_dl` będą równe 1. Pętla główna przegląda kolejne elementy tablicy, zaczynając od drugiego. Jeśli dany element tablicy jest większy lub równy poprzedniemu, to powiększamy wartość zmiennej `akt_dl` o 1. Ponadto sprawdzamy, czy długość badanego podciągu jest większa od dotychczas znalezionej największej długości podciągu. Jeśli jest większa, zmieniamy wartość zmiennej `maks_dl`. Gdy przeglądany element tablicy jest mniejszy od poprzedniego, staje się on początkiem następnego podciągu, a więc wartość zmiennej `akt_dl` musimy ustawić ponownie na 1. Oto zapis algorytmu w pseudokodzie oraz kod źródłowy funkcji realizującej ten algorytm:

Znajdujemy maksimum, s. 146

```
maks_dl ← 1
akt_dl ← 1
dla i ← 1, ..., n - 1 wykonuj
    jeśli A[i] ≥ A[i-1] to
        akt_dl ← akt_dl + 1
        jeśli akt_dl > maks_dl to maks_dl ← akt_dl
    w przeciwnym przypadku akt_dl ← 1
```

Fragment kodu źródłowego programu poszukującego długości najdłuższego spójnego podciągu niemalejącego – definicja funkcji znajdującej tę długość

```
1.  int DNSPNM(int A[])
2.  // Długość Najdłuższego Spójnego Podciągu NieMalejącego
3.  {
4.      int maks_dl=1, akt_dl=1, i;
5.      for (i=1;i<N;i++)
6.          if (A[i]>=A[i-1])
7.          {
8.              akt_dl++;
9.              if (akt_dl>maks_dl) maks_dl=akt_dl;
10.         }
11.         else akt_dl=1;
12.     return maks_dl;
13. }
```

N oznacza w programie stałą określającą rozmiar tablicy. Do wylosowania i wypisania elementów tablicy można wykorzystać te same funkcje, które stosowaliśmy w **programach wyszukujących element w tablicy**.

Rysunek 15.1 przedstawia przykładowe wywołanie programu.

Fragment kodu źródłowego programu wyszukującego element w tablicy, s. 152

```
15 70 31 44 52 70 9 66 38 11
Dlugosc najdluzszego spojnego podciagu niemalejacego: 4
```

Rys. 15.1. Przykład wywołania programu znajdującego długość najdłuższego spójnego podciągu niemalejącego

Ćwiczenie 1

Napisz program znajdujący długość najdłuższego spójnego podciągu niemalejącego. Wykorzystaj w nim zdefiniowaną wcześniej funkcję `DNSPNM`.

15.2. Szukamy najdłuższego spójnego podciągu niemalejącego

Żeby znaleźć najdłuższy spójny podciąg niemalejący, wystarczy wyznaczyć indeks pierwszego elementu tego podciągu. Koniec podciągu jest wówczas wyznaczany przez indeks pierwszego elementu mniejszego od poprzedniego lub przez koniec ciągu. Jeśli w danym ciągu jest kilka najdłuższych podciągów, nie jest ważne, który z nich znajdziemy.

Aby wyznaczyć indeks pierwszego elementu podciągu, zmodyfikujemy algorytm wyznaczający długość najdłuższego spójnego podciągu niemalejącego. Będziemy pamiętać nie tylko długość aktualnie badanego podciągu i podciągu dotychczas najdłuższego, lecz także indeksy pierwszych elementów tych podciągów. Zapiszemy je odpowiednio w zmiennych `akt_pocz` i `maks_pocz`.

A to ciekawe

Poszukiwanie podciągów w DNA

Podstawowym składnikiem DNA są nukleotydy. W DNA występują cztery rodzaje nukleotydów, a każdy z nich zawiera inną zasadę azotową: adeninę, guaninę, cytozynę lub tyminę. Nić DNA można więc opisać jako ciąg składający się z liter reprezentujących te nukleotydy: A, G, C i T. Dzięki poszukiwaniu powtarzających się podciągów we fragmentach DNA badacze potrafią np. ustalić pokrewieństwo osób lub wskazać sprawcę przestępstwa. Z kolei znalezienie w kodzie określonych genów pozwala określić predyspozycje do zachorowania na niektóre choroby.

Zmodyfikowany zapis algorytmu w pseudokodzie może być następujący:

```
maks_dl ← 1
akt_dl ← 1
maks_pocz ← 0
akt_pocz ← 0
dla i ← 1, …, n - 1 wykonuj
    jeśli A[i] ≥ A[i-1] to
        akt_dl ← akt_dl + 1
        jeśli akt_dl > maks_dl to
            maks_dl ← akt_dl
            maks_pocz ← akt_pocz
    w przeciwnym przypadku
        akt_dl ← 1
        akt_pocz ← i
```

Wartością początkową każdej ze zmiennych `maks_pocz` i `akt_pocz` jest zero (indeks pierwszego elementu tablicy). Początkiem kolejnego podciągu do zbadania jest pierwszy element, który jest mniejszy od elementu poprzedniego.

Zapisując powyższy algorytm w postaci funkcji, warto się zastanowić, co powinno być wartością funkcji: indeks pierwszego elementu znalezionego podciągu czy długość podciągu. Jeśli znamy indeks pierwszego elementu, będziemy mogli wypisać ten podciąg, podczas gdy informacja tylko o jego długości takiej możliwości nie daje. Jeśli wystąpi więcej niż jeden najdłuższy podciąg, funkcja znajdzie pierwszy taki podciąg i zwróci indeks jego pierwszego elementu.

Oto kod źródłowy funkcji realizującej podany algorytm:

Fragment kodu źródłowego programu wyznaczającego najdłuższy spójny podciąg niemalejący – definicja funkcji NSPNM, której wynikiem jest indeks pierwszego elementu tego podciągu

```
1.  int NSPNM(int A[])
2.  // Najdłuższy Spójny Podciąg NieMalejący
3.  // wynikiem jest indeks pierwszego elementu podciągu
4.  {
5.      int maks_dl=1, akt_dl=1, maks_pocz=0, akt_pocz=0, i;
6.      for (i=1;i<N;i++)
7.          if (A[i]>=A[i-1])
8.          {
9.              akt_dl++;
10.             if (akt_dl>maks_dl)
11.             {
12.                 maks_dl=akt_dl; maks_pocz=akt_pocz;
13.             }
14.         }
15.         else
16.         {
17.             akt_dl=1; akt_pocz=i;
18.         }
19.     return maks_pocz;
20. }
```

Funkcja `main` programu znajdującego najdłuższy spójny podciąg niemalejący powinna wypisywać elementy tablicy od elementu znajdującego się na pozycji, która jest wartością funkcji `NSPNM`, do momentu, kiedy napotka element nienależący do podciągu lub koniec tablicy. Kod źródłowy funkcji `main` może wyglądać następująco:

Fragment kodu źródłowego programu wyznaczającego najdłuższy spójny podciąg niemalejący – funkcja `main`

```
1.  int main()
2.  {
3.      int i;
4.      int A[N];
5.      srand(time(NULL));
6.      Losuj(A);
7.      Wypisz(A);
8.      cout<<"Najdluzszy spojny podciag niemalejacy:";
9.      cout<<endl;
10.     i=NSPNM(A);
11.     do
12.     {
13.         cout<<A[i]<<" ";
14.         i++;
15.     }
16.     while (i<N && A[i]>=A[i-1]);
17.     return 0;
18. }
```

Dobra rada

Pamiętaj, że jeśli w pętli powtarzanych jest kilka instrukcji, należy je ująć w nawiasy klamrowe, tworzące blok instrukcji.

Ponieważ zostanie wypisany co najmniej jeden element tablicy, wygodnie będzie użyć pętli **`do while`**, która sprawdza warunek powtarzania po wykonaniu instrukcji, a nie przed wykonaniem (linie 11–16). Zwróć uwagę na warunek pętli w linii 16 – najpierw musimy sprawdzić, czy indeks jest z zakresu tablicy, a potem dopiero odwołać się do elementu tablicy o tym indeksie.

Ćwiczenie 2

Napisz program wypisujący najdłuższy spójny podciąg niemalejący. Wykorzystaj w nim zdefiniowaną w temacie funkcję `NSPNM`.

Można także zapisać funkcję `NSPNM` tak, aby przekazywała do programu obydwie wyliczone wartości: indeks pierwszego elementu podciągu i długość znalezionego podciągu. Jedna z tych wartości może być wynikiem funkcji, a druga może być **przekazana jako parametr przez referencję** lub obydwie mogą być przekazane jako parametry (wówczas funkcja powinna być typu **`void`**).

Przekazanie parametru przez referencję, s. 161

Zmodyfikowany nagłówek funkcji NSPNM może być następujący:

Fragment kodu źródłowego programu wyznaczającego najdłuższy spójny podciąg niemalejący – zmodyfikowany nagłówek funkcji NSPNM

```
1.  int NSPNM(int A[], int &maks_dl)
2.  // Najdłuższy Spójny Podciąg NieMalejący
3.  // wynikiem jest indeks pierwszego elementu podciągu
4.  // długość podciągu jest zwracana za pośrednictwem
5.  // parametru
```

Zwróć uwagę, że zmienna maks_dl jest teraz parametrem, więc nie może być już deklarowana jako zmienna lokalna. Pozostała treść funkcji pozostaje bez zmian. Znaleziony podciąg można wypisać wówczas w funkcji main za pomocą pętli **for**. Oto kod źródłowy funkcji main:

Fragment kodu źródłowego programu wyznaczającego najdłuższy spójny podciąg niemalejący – zmodyfikowana funkcja main

```
1.  int main()
2.  {
3.      int i, p, dl;
4.      int A[N];
5.      srand(time(NULL));
6.      Losuj(A);
7.      Wypisz(A);
8.      cout<<"Najdluzszy spojny podciag niemalejacy:";
9.      cout<<endl;
10.     p=NSPNM(A,dl);
11.     for (i=p;i<p+dl;i++) cout<<A[i]<<" ";
12.     return 0;
13. }
```

Warto wiedzieć

Do zwrócenia pary liczb można wykorzystać strukturę przechowującą dwie liczby. O strukturach pisaliśmy w temacie 7.

Żeby znaleźć najdłuższy spójny podciąg niemalejący, program przegląda tablicę jeden raz, więc złożoność czasowa tego algorytmu jest liniowa: $O(n)$.

Ćwiczenie 3

Napisz i przetestuj program wypisujący najdłuższy spójny podciąg niemalejący. Wykorzystaj w nim funkcję NSPNM, w której:

a. indeks pierwszego elementu podciągu będzie wartością funkcji, a długość znalezionego podciągu będzie zwracana do programu jako parametr przekazany przez referencję,

b. indeks pierwszego elementu podciągu i długość znalezionego podciągu będą zwracane do programu jako parametry przekazywane przez referencję.

Zapamiętaj

Podciąg to wybrane elementy danego ciągu zachowujące kolejność występowania. W podciągu spójnym elementy występują obok siebie tak jak w ciągu wyjściowym. Żeby wyznaczyć najdłuższy niemalejący podciąg spójny, wystarczy przejrzeć ciąg raz.

15.3. Szukamy maksymalnej sumy podciągu spójnego

Kolejnym problemem, którym się zajmiemy, będzie znalezienie największej sumy wyrazów wśród podciągów spójnych danego ciągu. Gdyby w tablicy występowały tylko liczby dodatnie, rozwiązanie byłoby oczywiste – największą sumę wyrazów ma podciąg taki sam jak ciąg wyjściowy. Problem staje się trudniejszy, gdy w tablicy występują także liczby ujemne. Weźmy pod uwagę na przykład ciąg liczb:
23, –49, 6, 23, 24, –42, 40, 4, –48, 39

Szukana suma wynosi 55. Jest ona sumą wyrazów podciągu złożonego z liczb: 6, 23, 24, –42, 40, 4.

Oto specyfikacja problemu znajdowania maksymalnej sumy podciągu:

Specyfikacja

Dane: `A[0..n-1]` – tablica n liczb całkowitych, co najmniej jedna liczba w tablicy jest nieujemna.

Wynik: `maks_suma` – maksymalna suma elementów podciągu spójnego w tablicy `A`.

Rozważymy różne algorytmy znajdowania maksymalnej sumy podciągu spójnego. Założyliśmy, że w tablicy znajduje się co najmniej jedna liczba nieujemna. Jeśliby takiej liczby nie było, rozwiązaniem problemu byłaby największa liczba w ciągu wyjściowym.

Znajdowanie maksymalnej sumy podciągu spójnego – algorytm 1

Najprostszym rozwiązaniem wydaje się liczenie sum wyrazów we wszystkich podciągach spójnych. Jeśli aktualnie obliczona suma będzie większa od dotychczas znalezionej, zmieniamy wartość maksymalnej sumy. Powinniśmy zliczać sumę wszystkich podciągów rozpoczynających się od pierwszego elementu tablicy, następnie wszystkich podciągów rozpoczynających się od drugiego elementu itd. Wymaga to użycia potrójnie zagnieżdżonej pętli. Aktualnie liczoną sumę podciągu zapamiętamy w zmiennej `akt_suma`, a największą znalezioną sumę – w zmiennej `maks_suma`. Algorytm zapisany w pseudokodzie może wyglądać następująco:

```
maks_suma ← 0
dla i ← 0, …, n – 1 wykonuj
    dla j ← i, …, n – 1 wykonuj
        akt_suma ← 0
        // liczymy sumę podciągu od i do j
        dla k ← i, …, j wykonuj
            akt_suma ← akt_suma + A[k]
        jeśli akt_suma > maks_suma to
            maks_suma ← akt_suma
```

Oto kod źródłowy funkcji realizującej algorytm:

Fragment kodu źródłowego programu znajdującego maksymalną sumę podciągu spójnego – funkcja znajdująca tę sumę (algorytm 1)

```
1.  int MSPS(int A[])
2.  // Maksymalna Suma Podciągu Spójnego
3.  {
4.      int maks_suma=0, akt_suma, i, j, k;
5.      for  (i=0;i<N;i++)
6.          for (j=i;j<N;j++)
7.          {
8.              akt_suma=0;
9.              for (k=i;k<=j;k++) akt_suma+=A[k];
10.             if (akt_suma>maks_suma)
11.                 maks_suma=akt_suma;
12.         }
13.     return maks_suma;
14. }
```

W programie realizującym podany algorytm potrzebna jest funkcja losująca tablicę liczb, w której co najmniej jedna liczba będzie nieujemna. Kod źródłowy takiej funkcji może być następujący:

Fragment kodu źródłowego programu znajdującego maksymalną sumę podciągu spójnego – funkcja losująca tablicę *N* liczb całkowitych z przedziału [–50; 49]

```
1.  void Losuj(int A[])
2.  {
3.      for (int i=0;i<N;i++)
4.          A[i]=rand()%100-50;
5.      A[rand()%N]=rand()%50;
6.  }
```

Dobra rada

Możesz zmienić przedział losowania liczb na inny. Żeby wylosować liczbę z przedziału [*a*; *b*], skorzystaj z wyrażenia: `rand()%(b-a+1)+a`

Powyższa funkcja losuje liczby całkowite z przedziału [–50; 49]. W linii 5 w losowym miejscu tablicy dodatkowo wstawiana jest losowa liczba nieujemna. Dzięki temu mamy pewność, że w tablicy nie znajdą się same liczby ujemne.

Ćwiczenie 4

Napisz i przetestuj program znajdujący największą sumę podciągu spójnego z wykorzystaniem powyższych funkcji `MSPS` i `Losuj`.

Znajdowanie maksymalnej sumy podciągu spójnego – algorytm 2

Zwróć uwagę, że dla podciągów rozpoczynających się od tego samego elementu nie musimy liczyć każdej sumy od początku. Wystarczy po porównaniu aktualnej sumy z największą dotychczas znalezioną dodać do aktualnej sumy kolejny element tablicy. Na przykład dla kolejnych podciągów: 23, –49, 6 oraz 23, –49, 6, 23 wystarczy dodać do policzonej już sumy wyrazów pierwszego podciągu (–20) wartość kolejnego elementu (23). Pozwoli to usprawnić algorytm poprzez usunięcie jednej pętli.

Zmodyfikowany algorytm zapisany w pseudokodzie może wyglądać następująco:

```
maks_suma ← 0
dla i ← 0, …, n − 1 wykonuj
    akt_suma ← 0
    dla j ← i, …, n − 1 wykonuj
        akt_suma ← akt_suma + A[j]
        jeśli akt_suma > maks_suma to
            maks_suma ← akt_suma
```

Oto kod źródłowy funkcji realizującej zapisany w pseudokodzie algorytm:

```
1.  int MSPS(int A[])
2.  // Maksymalna Suma Podciągu Spójnego
3.  {
4.      int maks_suma=0, akt_suma, i, j;
5.      for (i=0;i<N;i++)
6.      {
7.          akt_suma=0;
8.          for (j=i;j<N;j++)
9.          {
10.             akt_suma+=A[j];
11.             if (akt_suma>maks_suma)
12.                 maks_suma=akt_suma;
13.         }
14.     }
15.     return maks_suma;
16. }
```

Fragment kodu źródłowego programu znajdującego maksymalną sumę podciągu spójnego – funkcja znajdująca tę sumę (algorytm 2)

Ćwiczenie 5

Napisz i przetestuj program znajdujący maksymalną sumę podciągu spójnego z wykorzystaniem powyższej funkcji.

Znajdowanie maksymalnej sumy podciągu spójnego – algorytm 3

Zastanówmy się, czy trzeba sprawdzać wszystkie podciągi spójne danego ciągu. Jeśli np. pierwszy element będzie ujemny, nie ma sensu rozpatrywać podciągów rozpoczynających się od tego elementu, ponieważ suma każdego podciągu zaczynającego się od drugiego elementu będzie większa. Ogólnie, gdyby aktualna suma miała przyjąć wartość ujemną lub zero, można rozpocząć badanie następnego podciągu od kolejnego elementu ciągu.

Zapis algorytmu w pseudokodzie oraz kod źródłowy funkcji realizującej ten algorytm są następujące.

```
maks_suma ← 0
akt_suma ← 0
dla i ← 0, …, n − 1 wykonuj
    jeśli akt_suma + A[i] > 0 to
        akt_suma ← akt_suma + A[i]
        jeśli akt_suma > maks_suma to
            maks_suma ← akt_suma
    w przeciwnym przypadku akt_suma ← 0
```

Fragment kodu źródłowego programu znajdującego maksymalną sumę podciągu spójnego – funkcja znajdująca tę sumę (algorytm 3)

```
1.  int MSPS(int A[])
2.  // Maksymalna Suma Podciągu Spójnego
3.  {
4.      int maks_suma=0, akt_suma=0, i;
5.      for (i=0;i<N;i++)
6.          if (akt_suma+A[i]>0)
7.          {
8.              akt_suma+=A[i];
9.              if (akt_suma>maks_suma)
10.                 maks_suma=akt_suma;
11.         }
12.         else akt_suma=0;
13.     return maks_suma;
14. }
```

Rysunek 15.2 przedstawia przykład wywołania programu. Program dla ciągu –26, 11, –45, 40, –49, 36, 16, 18, 42, –10 zwrócił liczbę 112 – jest ona sumą wyrazów podciągu 36, 16, 18, 42.

```
-26 11 -45 40 -49 36 16 18 42 -10
Maksymalna suma podciagu spojnego: 112
```

Rys. 15.2. Przykład wywołania programu znajdującego maksymalną sumę podciągu spójnego

Ćwiczenie 6

Napisz i przetestuj program realizujący powyższy algorytm.

15.4. Porównujemy algorytmy znajdowania maksymalnej sumy podciągu spójnego

Zastanówmy się nad złożonością czasową trzech omówionych algorytmów znajdowania maksymalnej sumy podciągu spójnego. Dla tych algorytmów operacją dominującą jest dodanie do aktualnej sumy wyrazów kolejnego elementu tablicy.

Obliczmy liczbę operacji dominujących dla pierwszego algorytmu. Dla podciągów rozpoczynających się od pierwszego elementu tablicy instrukcja dodawania wykona się $1 + 2 + \ldots + n$ razy, dla podciągów rozpoczynających się od drugiego elementu tablicy: $1 + 2 + \ldots + n - 1$ razy itd.

Otrzymujemy wyrażenie:

$$(1 + 2 + \ldots + n) + (1 + 2 + \ldots + n - 1) + \ldots + 2 + 1$$

Jest ono sumą n kolejnych **liczb trójkątnych** i wynosi:

$$\frac{n \cdot (n+1) \cdot (n+2)}{6} = \frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}$$

Liczba trójkątna to liczba będąca sumą kolejnych liczb naturalnych.

Zatem pierwszy algorytm ma złożoność czasową $O(n^3)$.

Drugi z algorytmów wykonuje o wiele mniej instrukcji dodawania. Dla podciągów rozpoczynających się od pierwszego elementu tablicy będzie to n operacji dodawania do aktualnej sumy, dla podciągów rozpoczynających się od drugiego elementu tablicy – $n - 1$ itd. W sumie algorytm wykona następującą liczbę operacji:

$$n + (n-1) + \ldots + 2 + 1 = \frac{(n+1) \cdot n}{2} = \frac{n^2}{2} + \frac{n}{2}$$

Jego złożoność czasowa jest więc kwadratowa: $O(n^2)$.

Trzeci z algorytmów tylko raz przegląda tablicę liczb, zatem jego złożoność czasowa jest liniowa: $O(n)$.

Tabela 15.1 zawiera porównanie złożoności czasowej poszczególnych algorytmów dla różnej liczby elementów ciągu.

Liczba elementów ciągu (n)	Liczba operacji dominujących		
	Algorytm 1 $O(n^3)$, dokładniej $\frac{n^3}{6} + \frac{n^2}{2} + \frac{n}{3}$	Algorytm 2 $O(n^2)$, dokładniej $\frac{n^2}{2} + \frac{n}{2}$	Algorytm 3 $O(n)$
100	171 700	5 050	100
1 000	167 167 000	500 500	1 000
10 000	166 716 670 000	50 005 000	10 000

Tabela 15.1. Porównanie złożoności czasowej algorytmów znajdowania maksymalnej sumy podciągu spójnego

Ćwiczenie 7

Sprawdź działanie poszczególnych algorytmów dla większych wartości n (bez wypisywania wylosowanego ciągu). Dla jakich wartości n musisz dłużej poczekać na wynik, kiedy program wykonuje algorytm 1, a dla jakich – gdy wykonuje algorytm 2?

15.5. Szukamy podciągu spójnego o maksymalnej sumie

Aby znaleźć podciąg o maksymalnej sumie, należy pamiętać indeks początku badanego podciągu oraz albo jego sumę, albo indeks końca. W celu wyznaczenia indeksu pierwszego elementu podciągu wystarczy zmodyfikować wybrany algorytm liczenia maksymalnej sumy podciągu.

Liniowy algorytm znajdowania maksymalnej sumy, uzupełniony o wyznaczanie indeksu pierwszego elementu podciągu, zapisany w pseudokodzie, wygląda następująco:

```
maks_suma ← 0
akt_suma ← 0
maks_pocz ← 0
akt_pocz ← 0
dla i ← 0, …, n – 1 wykonuj
    jeśli akt_suma + A[i] > 0 to
        akt_suma ← akt_suma + A[i]
        jeśli akt_suma > maks_suma to
            maks_suma ← akt_suma
            maks_pocz ← akt_pocz
    w przeciwnym przypadku
        akt_suma ← 0
        akt_pocz ← i + 1
```

Funkcja realizująca powyższy algorytm musi zwrócić dwie wartości: znalezioną sumę oraz indeks pierwszego elementu podciągu. Dlatego przekażemy obie te wartości przez referencję. Funkcja nie będzie miała wartości, będzie więc typu **void**. Oto jej kod źródłowy:

Fragment kodu źródłowego programu znajdującego podciąg spójny o maksymalnej sumie – funkcja znajdująca pierwszy element oraz maksymalną sumę podciągu

```
1.  void PSMS(int A[], int &maks_suma, int &maks_pocz)
2.  // Podciąg Spójny o Maksymalnej Sumie
3.  // maks_suma – maksymalna suma podciągu spójnego
4.  // maks_pocz – indeks pierwszego elementu podciągu
5.  // o maksymalnej sumie
6.  {
7.      maks_suma=0; maks_pocz=0;
8.      int akt_suma=0, akt_pocz=0, i;
9.      for (i=0;i<N;i++)
10.         if (akt_suma+A[i]>0)
11.         {
12.             akt_suma+=A[i];
13.             if (akt_suma>maks_suma)
14.             {
15.                 maks_suma=akt_suma;
16.                 maks_pocz=akt_pocz;
17.             }
18.         }
19.         else
20.         {
21.             akt_suma=0;
22.             akt_pocz=i+1;
23.         }
24. }
```

Warto wiedzieć

W języku C++ komentarz można rozpocząć od znaków /*, a zakończyć znakami */. Komentarz między takimi znakami może być wielowierszowy.

Zmieniając wartość dotychczas znalezionej największej sumy (linia 15), zmieniamy także wartość indeksu pierwszego elementu podciągu (linia 16). W przypadku, gdy dodanie kolejnego elementu tablicy da sumę niedodatnią, badany jest kolejny podciąg rozpoczynający się od następnego elementu tablicy (jego indeksem jest wartość i+1, linia 22).

Program wykorzystujący powyższą funkcję podczas wypisywania podciągu musi zliczać na nowo sumę jego wyrazów: od pierwszego wyrazu podciągu do momentu, gdy liczona suma będzie równa maksymalnej sumie. Oto przykładowy kod źródłowy funkcji `main`:

```
1.  int main()
2.  {
3.      int suma, pocz, tmp=0, i;
4.      int A[N];
5.      srand(time(NULL));
6.      Losuj(A);
7.      Wypisz(A);
8.      PSMS(A,suma,pocz);
9.      cout<<"Maksymalna suma podciagu spojnego: ";
10.     cout<<suma<<endl;
11.     cout<<"Podciag spojny o maksymalnej sumie: ";
12.     cout<<endl;
13.     i=pocz;
14.     while (tmp!=suma)
15.     {
16.         cout<<A[i]<<" ";
17.         tmp+=A[i];
18.         i++;
19.     }
20.     return 0;
21. }
```

Fragment kodu źródłowego programu znajdującego podciąg o maksymalnej sumie – funkcja `main`

Przykładowe wywołanie programu przedstawia rysunek 15.3.

```
1 -34 45 28 -21 44 -2 48 -11 19
Maksymalna suma podciagu spojnego: 150
Podciag spojny o maksymalnej sumie:
45 28 -21 44 -2 48 -11 19
```

Rys. 15.3. Przykład uruchomienia programu znajdującego podciąg spójny o maksymalnej sumie dla stałej `N` = 10

Ćwiczenie 8

Napisz i przetestuj program wypisujący podciąg spójny o maksymalnej sumie.

Łatwiej będzie wypisać znaleziony podciąg, gdy będziemy znali także indeks jego ostatniego elementu. W tym celu można dodać jeszcze jeden parametr (maks_kon) przekazywany przez referencję, określający tę wartość. Nagłówek funkcji PSMS będzie wówczas następujący:

Fragment kodu źródłowego programu znajdującego podciąg spójny o maksymalnej sumie – zmodyfikowany nagłówek funkcji znajdującej ten podciąg

```
1. void PSMS(int A[], int &maks_suma, int &maks_pocz,
2.           int &maks_kon)
3. // Podciąg Spójny o Maksymalnej Sumie
4. // maks_suma - maksymalna suma podciągu spójnego
5. // maks_pocz - indeks pierwszego elementu podciągu
6. // o maksymalnej sumie
7. // maks_kon - indeks ostatniego elementu podciągu
8. // o maksymalnej sumie
```

Aktualizując informację o znalezionym podciągu, należy także określić wartość parametru maks_kon. Końcem jest indeks aktualnie oglądanego elementu (wartość zmiennej sterującej pętli **for**, czyli i).

Fragment kodu źródłowego programu znajdującego podciąg spójny o maksymalnej sumie – instrukcja wyznaczająca indeks początkowy i indeks końcowy oraz sumę podciągu

```
1. if (akt_suma>maks_suma)
2.     {
3.         maks_suma=akt_suma;
4.         maks_pocz=akt_pocz;
5.         maks_kon=i;
6.     }
```

Znaleziony podciąg można wówczas wypisać za pomocą pętli **for** przeglądającej elementy tablicy w zakresie znalezionych wartości maks_pocz i maks_kon.

Ćwiczenie 9

Napisz i przetestuj program wypisujący podciąg spójny o maksymalnej sumie. Wykorzystaj zmodyfikowaną zgodnie z powyższym nagłówkiem funkcję PSMS.

Zapamiętaj

Żeby policzyć maksymalną sumę podciągu spójnego ciągu liczb całkowitych, w którym co najmniej jedna liczba jest nieujemna, nie trzeba liczyć sum wszystkich podciągów. Wystarczy raz przejrzeć elementy ciągu i podczas przeglądania sumować jego wyrazy. Jeśli obliczona suma będzie niedodatnia, sumowanie rozpoczynamy ponownie od następnego elementu.

Podsumowanie

- Podciąg to wybrane wyrazy ciągu zachowujące kolejność występowania.
- Podciąg spójny to taki podciąg, którego wyrazy w ciągu wyjściowym występują obok siebie.
- Podciąg niemalejący to taki podciąg, w którym każdy kolejny element jest większy lub równy poprzedniemu.
- Żeby znaleźć spójny podciąg niemalejący, wystarczy raz przejrzeć elementy ciągu.
- Algorytmy znajdowania maksymalnej sumy podciągu analizujące wszystkie możliwe sumy podciągów mają złożoność czasową $O(n^3)$ lub $O(n^2)$.
- Istnieje algorytm liniowy $O(n)$, który znajduje maksymalną sumę podciągu spójnego. Nie analizuje on wszystkich podciągów. Kiedy suma aktualnie badanego podciągu nie jest dodatnia, rozpoczyna badanie kolejnego podciągu od następnego elementu.
- Jeśli funkcja ma zwrócić więcej niż jedną wartość typu prostego, warto zastosować przekazanie parametru przez referencję.

Zadania

* **1** Napisz program wypisujący najdłuższy spójny podciąg malejący ciągu n liczb całkowitych.

* **2** Napisz program, który wypisze najdłuższy spójny podciąg rosnący ciągu n liczb całkowitych, a w przypadku występowania więcej niż jednego najdłuższego podciągu wypisze ostatni znaleziony.

** **3** Napisz program, który wypisze najdłuższy spójny podciąg rosnący ciągu n liczb całkowitych, a w przypadku występowania więcej niż jednego najdłuższego podciągu wypisze wszystkie.

** **4** Napisz program wypisujący najdłuższy podciąg spójny złożony z liczb całkowitych dodatnich ciągu n liczb całkowitych. Wykorzystaj w nim funkcję losującą tablicę tak, aby występowały w niej zarówno liczby dodatnie, jak i ujemne.

** **5** Napisz program znajdujący maksymalną sumę podciągu spójnego ciągu n liczb całkowitych, w którym mogą wystąpić same liczby ujemne. Program napisz tak, żeby przejrzał tablicę co najwyżej dwa razy.

** **6** Napisz program wyznaczający w ciągu n liczb całkowitych liczbę podciągów spójnych o podanej (wczytanej) sumie. Program napisz tak, aby wykonywał jak najmniej operacji.

*** **7** Napisz program wypisujący z ciągu n liczb całkowitych wszystkie podciągi spójne o podanej (wczytanej) sumie. Program napisz tak, aby wykonywał jak najmniej operacji.

16. W poszukiwaniu lidera i idola

Słowa „lider" i „idol" są dziś bardzo popularne. Często mówi się o liderach zespołów, partii, wyścigów, klasyfikacji oraz o idolach muzycznych, sportowych, filmowych. Lider to ktoś, kto przewodzi grupie ludzi lub – jak np. w zawodach sportowych czy sondażach wyborczych – wygrywa, ma przewagę nad innymi. Z kolei jedną z głównych cech idola jest to, że jest rozpoznawalny, czyli znany przez wiele osób, których sam najczęściej nie zna. Wymienione cechy lidera i idola sprawiły, że pojęcia te przeniesiono na grunt algorytmiki. W tym temacie zajmiemy się właśnie poszukiwaniem lidera i idola w zbiorze.

Cele lekcji

- Znajdziesz lidera w zbiorze danych.
- Poszukasz idola w grupie osób.
- Poznasz algorytmy znajdowania lidera i idola różniące się złożonością czasową.
- Wykorzystasz funkcję `sort` z biblioteki STL.
- Poznasz tablice dwuwymiarowe i zastosujesz je podczas znajdowania idola.

16.1. Jak znaleźć lidera w zbiorze?

Lider

W ujęciu algorytmicznym **liderem** nazywamy taki element zbioru n-elementowego, który występuje w tym zbiorze więcej niż $n/2$ razy.

Sformułujmy specyfikację problemu i zastanówmy się nad algorytmami go rozwiązującymi.

Warto wiedzieć

W wyborach przeprowadzanych zgodnie z ordynacją większościową wygrywa kandydat, który uzyskał większość wszystkich oddanych głosów. To znaczy, że na n oddanych głosów zwycięzca musi zdobyć przynajmniej $n/2 + 1$ głosów.

Specyfikacja

Dane: `A[0..n-1]` – tablica n liczb całkowitych nieujemnych.

Wynik: `lider` – liczba, która występuje w tablicy `A` więcej niż `n div 2` razy, lub –1, gdy takiej liczby nie ma.

Zauważ, że lider w zbiorze n-elementowym może być co najwyżej jeden. Gdyby liderami były dwa elementy, musielibyśmy mieć co najmniej $2 \cdot (n/2 + 1) = n + 2$ elementów, czyli więcej, niż liczy zbiór.

Najprostszym sposobem na znalezienie lidera wydaje się sprawdzanie kolejnych elementów tablicy. Wystarczy policzyć, ile razy w tablicy powtarza się pierwszy element. Jeśli występuje więcej niż $n/2$ razy, to jest on liderem. Jeśli nie, to powtarzamy czynności dla drugiego elementu tablicy itd. Jeśli żaden element nie będzie występował odpowiednio wiele razy, to lidera nie ma, czyli wynikiem będzie wartość –1.

Oto zapis tego algorytmu w pseudokodzie:

```
funkcja SzukajLidera(A[])
    dla i ← 0, 1, ..., n - 1 wykonuj
        ile ← 0
        dla j ← 0, 1, ..., n - 1 wykonuj
            jeśli A[j] = A[i] to ile ← ile + 1
        jeśli ile > n div 2 to zwróć A[i] i zakończ
    zwróć -1 i zakończ
```

Ćwiczenie 1

Zastanów się, jak można by nieco usprawnić powyższy algorytm. Czy trzeba sprawdzać wszystkie elementy tablicy? Kiedy można stwierdzić, że lider w zbiorze nie występuje? Zapisz poprawiony algorytm w pseudokodzie.

Zastanówmy się, czy posortowanie elementów tablicy ułatwiłoby znalezienie lidera. Ponieważ musi on występować więcej niż $n/2$ razy, to jedynym kandydatem na lidera w ciągu uporządkowanym jest element środkowy. Wystarczy więc policzyć, ile razy występuje w tablicy element znajdujący się na pozycji n **div** 2.

Oto algorytm poszukiwania lidera z wykorzystaniem sortowania zapisany w pseudokodzie:

```
funkcja SzukajLidera(A[])
    Sortuj(A)
    ile ← 0
    kandydat ← A[n div 2]
    dla i ← 0, 1, ..., n - 1 wykonuj
        jeśli A[i] = kandydat to ile ← ile + 1
    jeśli ile > n div 2 to zwróć kandydat i zakończ
    w przeciwnym przypadku zwróć -1 i zakończ
```

W powyższym zapisie zastosowaliśmy funkcję o nazwie Sortuj, która sortuje liczby wybraną metodą.

Warto wiedzieć

Element środkowy w tablicy n-elementowej, indeksowanej od 0, ma indeks n **div** 2. Dla n nieparzystego jest to rzeczywiście element środkowy. Dla n parzystego w tablicy występują dwa elementy środkowe – można wówczas wybrać dowolny z nich. Indeks n **div** 2 wskaże element środkowy znajdujący się bliżej końca tablicy.

Ćwiczenie 2

Porównaj dwa powyższe algorytmy przy założeniu, że w drugim zastosujesz jedną z poznanych wcześniej **metod sortowania prostego**. Oceń złożoność czasową tych algorytmów.

Metody sortowania prostego, s. 170–179

Funkcja losująca dane

Zanim napiszemy funkcję poszukującą lidera, musimy w odpowiedni sposób przygotować dane wejściowe, czyli tablicę z losowymi liczbami. Nie wystarczy w tym przypadku zwykłe wylosowanie tablicy, ponieważ prawdopodobieństwo, że w tak utworzonym ciągu jakiś element wystąpi ponad $n/2$ razy (czyli będzie liderem), jest bliskie zeru.

Dobra rada

Zwykłe losowanie tablicy możesz zastosować w przypadku, gdyby tablica miała się składać wyłącznie z zer i jedynek.

Dobra rada

Dane możesz przygotować ręcznie, np. w pliku tekstowym. Program powinien na początku wczytać je do tablicy.

Wylosujmy liczby w taki sposób, żeby istniała realna szansa, że wśród nich będzie lider. W tym celu najpierw wylosujemy jedną liczbę i zapamiętamy ją w zmiennej `x`. Następnie dla każdego elementu tablicy będziemy losować wartość zero-jedynkową, która będzie decydować o tym, czy dany element tablicy wypełnić wartością losową czy wartością zmiennej `x`.

Oto kod źródłowy funkcji losującej tablicę:

Kod źródłowy funkcji losującej tablicę potencjalnie zawierającą lidera

```
1.  void Losuj(int A[])
2.  {
3.      int x=rand()%100;
4.      for (int i=0;i<N;i++)
5.          if (rand()%2==0) A[i]=rand()%100;
6.          else A[i]=x;
7.  }
```

W linii 3 zmiennej `x` przypisana jest losowa wartość od 0 do 99. Będzie to wartość potencjalnego lidera w zbiorze. W pętli (linie 4–6) podejmowana jest losowa decyzja (z prawdopodobieństwem 50%), czy w aktualnym polu tablicy ma się pojawić wartość `x` czy losowa liczba z przedziału [0; 99].

Ćwiczenie 3

Napisz program realizujący pierwszy algorytm znajdowania lidera (sprawdzanie każdego elementu oddzielnie). Algorytm zaimplementuj w postaci funkcji. Wykorzystaj powyższą funkcję `Losuj`, uzupełnij także program o funkcję wypisującą tablicę.

Dobra rada

Kiedy losujesz tablicę, pamiętaj, aby dodać do programu biblioteki `ctime` oraz `cstdlib`.

Funkcja szukająca lidera

Zaimplementujemy teraz funkcję poszukującą lidera w wylosowanej tablicy zgodnie z drugim algorytmem (z uporządkowaną tablicą). Do posortowania elementów tablicy możemy wykorzystać jeden z poznanych wcześniej algorytmów sortowania.

Biblioteka STL

Możemy też skorzystać z **biblioteki STL** (ang. *Standard Template Library* – standardowa biblioteka szablonów), która zawiera zbiór szablonów wielu algorytmów i struktur danych, w tym **funkcji sortującej `sort`**. Funkcję tę wywołujemy w następujący sposób:

Funkcja `sort`

```
sort(tablica, tablica + liczba elementów do sortowania)
```

Warto wiedzieć

Za pomocą funkcji `sort` można posortować fragment tablicy oraz zdefiniować własny porządek sortowania (domyślnie stosowana jest relacja <, zapewniająca porządek rosnący).

Pierwszym parametrem funkcji `sort` jest nazwa tablicy, a drugim – nazwa tablicy, znak + oraz liczba elementów do posortowania. Na przykład sortowanie `N` elementów tablicy `T` z wykorzystaniem funkcji `sort` zapisujemy następująco: `sort(T,T+N)`. Aby skorzystać z funkcji `sort` w języku C++, należy na początku kodu programu dołączyć bibliotekę `algorithm`.

Oto zapis funkcji SzukajLidera, realizującej drugi z przedstawionych wcześniej algorytmów:

Kod źródłowy funkcji poszukującej lidera w zbiorze uporządkowanym

```
1.  int SzukajLidera(int A[])
2.  {
3.      int i, kandydat, ile=0;
4.      sort(A,A+N);
5.      kandydat=A[N/2];
6.      for (i=0;i<N;i++)
7.          if (A[i]==kandydat) ile++;
8.      if (ile>N/2) return kandydat;
9.      else return -1;
10. }
```

W linii 4 wywoływana jest funkcja **sort**. Stała N określa rozmiar tablicy, dlatego drugim parametrem jest A+N. W linii 5 wskazywany jest kandydat na lidera – element środkowy tablicy A.

Ćwiczenie 4

Napisz program wykorzystujący powyższą funkcję szukającą lidera, w której używa się funkcji **sort** z biblioteki STL.

16.2. Efektywny algorytm poszukiwania lidera

Omówimy jeszcze jeden algorytm, który najpierw wyłoni kandydata na lidera bez porządkowania zbioru, a potem sprawdzi, czy rzeczywiście występuje on odpowiednią liczbę razy.

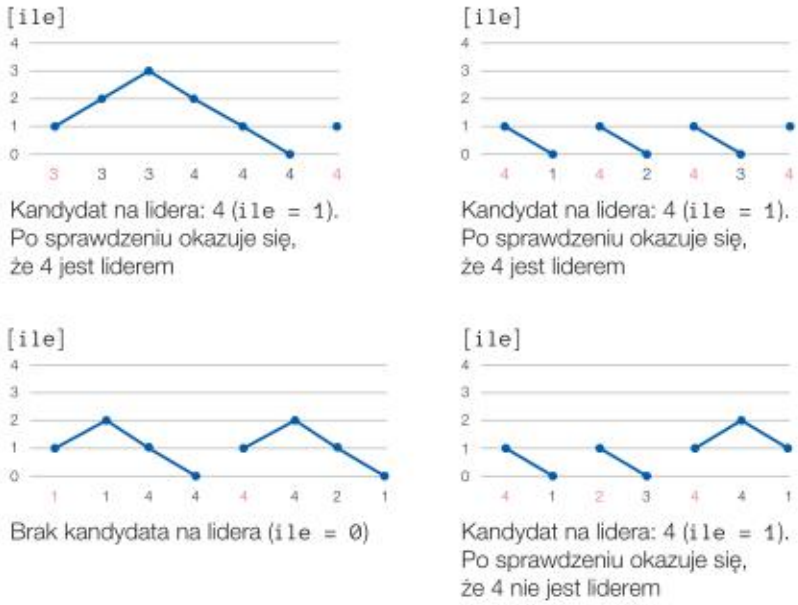
Zauważ, że jeżeli ze zbioru mającego lidera usuniemy dwie różne liczby, to lider się nie zmienia. Dzieje się tak, ponieważ:

- jeśli żadna z usuwanych liczb nie jest liderem, to liczba wystąpień lidera się nie zmienia, a zbiór liczy o dwa elementy mniej,
- jeśli jedna z usuwanych liczb jest liderem, to liczba wystąpień lidera zmniejszy się o jeden, jednocześnie liczebność zbioru zmniejszy się o dwa elementy, więc nadal lider występuje więcej niż $n/2$ razy.

Nie będziemy usuwać liczb z tablicy, ale zliczać wystąpienia kandydata na lidera. Najpierw za kandydata przyjmiemy pierwszy element i ustawimy licznik jego wystąpień na 1. Jeśli kolejny element jest taki sam, to zwiększamy licznik wystąpień, w przeciwnym przypadku go zmniejszamy. Gdy licznik osiągnie wartość 0, kolejna liczba staje się kandydatem.

Jeśli po przejrzeniu całego zbioru licznik będzie równy 0, to lider nie występuje w zbiorze. Dodatnia wartość licznika nie oznacza jeszcze, że wyłoniony kandydat jest liderem. Należy to sprawdzić, czyli policzyć jego wystąpienia.

Działanie algorytmu poszukiwania lidera dla różnych zbiorów danych ilustruje rysunek 16.1. Widać na nim zmianę wartości zmiennej `ile`. Ta wartość wskazuje, czy brana pod uwagę liczba jest kandydatem na lidera. Liczby zaznaczone na czerwono są rozpatrywane jako kandydaci na lidera w zbiorze.



Rys. 16.1. Schematy objaśniające, jak się zlicza wystąpienia kandydatów na lidera w zbiorze

Zauważ, że jeśli wartość zmiennej `ile` po rozpatrzeniu ostatniego elementu zbioru jest większa od zera (na rys. 16.1 występują trzy takie sytuacje), to został ustalony kandydat na lidera. Jest nim ostatnia liczba zaznaczona na czerwono. Kandydata na lidera należy jeszcze sprawdzić, czyli zliczyć jego wystąpienia w zbiorze.

Oto zapis algorytmu w pseudokodzie:

Warto wiedzieć

W pseudokodach funkcji należy podawać parametry, na których operują. Robi się to podobnie jak w językach programowania. Jeśli parametrem jest tablica `T`, to zapisuje się ją jako `T[]`.

```
funkcja SzukajLidera(A[])
    ile ← 0
    dla i ← 0, 1, ..., n - 1 wykonuj
        jeśli ile = 0 to
            kandydat ← A[i]
            ile ← 1
        w przeciwnym przypadku
            jeśli A[i] = kandydat to ile ← ile + 1
            w przeciwnym przypadku ile ← ile - 1
    jeśli ile = 0 to zwróć -1 i zakończ
    ile ← 0
    dla i ← 0, 1, ..., n - 1 wykonuj
        jeśli A[i] = kandydat to ile ← ile + 1
    jeśli ile > n div 2 to zwróć kandydat i zakończ
    w przeciwnym przypadku zwróć -1 i zakończ
```

Kod źródłowy funkcji `SzukajLidera`, realizującej powyższy algorytm, może być następujący:

Kod źródłowy funkcji poszukującej lidera w zbiorze

```
1.  int SzukajLidera(int A[])
2.  {
3.      int i, kandydat, ile=0;
4.      for (i=0;i<N;i++)
5.          if (ile==0)
6.          {
7.              ile=1;
8.              kandydat=A[i];
9.          }
10.         else if (A[i]==kandydat) ile++;
11.              else ile--;
12.     if (ile==0) return -1;
13.     ile=0;
14.     for (i=0;i<N;i++)
15.         if (A[i]==kandydat) ile++;
16.     if (ile>N/2) return kandydat;
17.     else return -1;
18. }
```

Pętla w liniach 4–11 wyznacza kandydata na lidera. Jeśli go nie znajdzie (wartość zmiennej `ile` będzie równa 0), to wynikiem funkcji jest –1 (linia 12). W pętli w liniach 14–15 następuje sprawdzenie, czy wyłoniony kandydat rzeczywiście jest liderem.

Rysunek 16.2 pokazuje przykładowe wywołanie programu dla zbioru 10-elementowego.

```
58 32 32 11 32 32 10 32 32 32
Lider = 32
```

Rys. 16.2. Przykładowe wywołanie programu poszukującego lidera w zbiorze

Ćwiczenie 5

Napisz program wykorzystujący powyższą funkcję i sprawdź jego działanie.

Zapamiętaj

Biblioteka STL dostarcza wielu użytecznych szablonów struktur danych i algorytmów, np. funkcję `sort`, umożliwiającą porządkowanie danych. Żeby korzystać z określonej struktury danych lub algorytmów, należy dołączyć do programu odpowiednią bibliotekę – dla funkcji `sort` jest to biblioteka `algorithm`.

16.3. Porównanie złożoności czasowej algorytmów poszukiwania lidera

Warto wiedzieć

Pierwszy algorytm można nieznacznie poprawić przez ograniczenie poszukiwania lidera tylko do połowy tablicy. Algorytm wykona wtedy dwa razy mniej operacji, ale jego złożoność czasowa nadal będzie wynosić $O(n^2)$.

W pierwszym z omówionych algorytmów jeśli nie ma lidera, to dla każdego elementu należy przejrzeć całą tablicę, a więc wykonać n^2 operacji porównania elementów tablicy.

Złożoność czasowa drugiego algorytmu jest uzależniona od zastosowanej metody sortowania. Po posortowaniu tablica jest przeglądana jeden raz, a więc wykonywanych jest dodatkowo n operacji. Jeśli zastosujemy jedną z poznanych wcześniej metod sortowania prostego o złożoności czasowej $O(n^2)$, nie uzyskamy poprawy względem pierwszego algorytmu. Użyta w przykładzie funkcja `sort` z biblioteki STL działa znacznie szybciej niż metody sortowania prostego – jej złożoność czasowa wynosi $O(n \cdot \log n)$.

Trzeci algorytm wykonuje o wiele mniej operacji. Przegląda on tablicę dwa razy: pierwszy raz w celu wyznaczenia kandydata na lidera, a drugi raz, żeby sprawdzić, czy kandydat rzeczywiście jest liderem. Złożoność czasowa tego algorytmu jest liniowa – wynosi $O(n)$.

Liczba elementów zbioru (n)	Przybliżona liczba operacji dominujących		
	Algorytm 1 $O(n^2)$	Algorytm 2 $O(n \cdot \log n)$	Algorytm 3 $O(n)$, dokładniej $2 \cdot n$
100	10 000	700	200
1 000	1 000 000	10 000	2 000
10 000	100 000 000	132 000	20 000

Tabela 16.1. Porównanie złożoności czasowej algorytmów znajdowania lidera

16.4. Jak znaleźć idola w zbiorze?

Idol

Z algorytmicznego punktu widzenia **idolem** w grupie osób nazywamy taką osobę, którą znają wszyscy w tej grupie, ale ona nie zna nikogo. Zastanówmy się, jaka struktura danych może reprezentować informację o znajomości dwóch osób.

Załóżmy, że grupa liczy n osób. Ile par można utworzyć w obrębie takiej grupy? Pierwsza osoba może być w parze ze wszystkimi pozostałymi, czyli $n - 1$ osobami, druga z $n - 2$ (bo policzyliśmy już parę z pierwszą osobą) itd. Łączna liczba par wynosi:

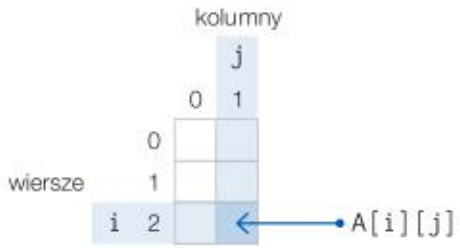
$$(n-1)+(n-2)+\ldots+2+1=\frac{n\cdot(n-1)}{2}$$

Dla każdej pary należy pamiętać dwie informacje, ponieważ relacja znajomości między dwoma osobami nie jest symetryczna, tzn. jedna osoba może znać drugą, ale druga nie musi znać pierwszej. Należy więc przechować $n \cdot (n - 1)$ informacji o znajomości poszczególnych par.

Informacje te zapamiętamy w **tablicy dwuwymiarowej** o n wierszach i n kolumnach. W takiej tablicy mamy n^2 elementów, czyli możemy zapisać wszystkie informacje o znajomościach w grupie n osób.

• Tablica dwuwymiarowa

Liczba wierszy i kolumn w tablicy dwuwymiarowej nie musi być równa. Można deklarować tablicę o n wierszach i m kolumnach. Rysunek 16.3 przedstawia tablicę A o 3 wierszach i 2 kolumnach.



Rys. 16.3. Przykład tablicy dwuwymiarowej 3 na 2 elementy

Do poszczególnych elementów odwołujemy się, podając dwa indeksy – numer wiersza i numer kolumny, np. `A[i][j]`.

Informacja o tym, czy osoba i zna osobę j, jest informacją logiczną, więc w tablicy będą przechowywane wartości **`prawda`** lub **`fałsz`**. Jeśli element `A[i][j]` przyjmuje wartość **`prawda`**, oznacza to, że osoba i zna osobę j. W przeciwnym razie (gdy `A[i][j]` ma wartość **`fałsz`**) osoba i nie zna osoby j. Oto specyfikacja problemu znajdowania idola:

Warto wiedzieć

Można także deklarować tablice wielowymiarowe (o więcej niż dwóch wymiarach), ale w praktyce stosuje się je bardzo rzadko.

Specyfikacja

Dane: `n` – liczba całkowita większa od 1 określająca liczebność grupy, `A[0..n-1][0..n-1]` – tablica `n` na `n` elementów zawierająca wartości logiczne określające znajomości w grupie. Jeśli `A[i][j]` przyjmuje wartość **`prawda`**, to osoba `i` zna osobę `j`, $i \neq j, 0 \leq i < n, 0 \leq j < n$. Osoba `i` nie zna osoby `j`, gdy `A[i][j]` przyjmuje wartość **`fałsz`**.

Wynik: `idol` – liczba, która określa numer idola w grupie, lub –1, gdy idola nie ma w zbiorze.

Rysunek 16.4 przedstawia przykładową tablicę znajomości dla grupy 5 osób. Wartości na głównej przekątnej (znajomość osoby samej ze sobą) nie mają znaczenia, dlatego pola te są szare.

Osoba o numerze 0 zna osoby 2 i 3, a nie zna osób 1 i 4. Osoba 1 zna 0 i 3, a nie zna 2 i 4 itd. W powyższej grupie znajduje się idol – jest to osoba o numerze 3, którą znają wszyscy, a ona nie zna nikogo.

	0	1	2	3	4
0		fałsz	prawda	prawda	fałsz
1	prawda		fałsz	prawda	fałsz
2	prawda	fałsz		prawda	prawda
3	fałsz	fałsz	fałsz		fałsz
4	fałsz	fałsz	prawda	prawda	

Rys. 16.4. Przykładowa tablica znajomości dla 5 osób

Warto wiedzieć

Elementy na głównej przekątnej oznaczają, że dana osoba zna samą siebie.

Algorytm poszukiwania idola

W grupie osób nie może być dwóch idoli. Pierwszy musiałby znać drugiego, a drugi pierwszego, podczas gdy idol nie zna nikogo. Szukanie idola sprowadza się do znalezienia wiersza złożonego z samych wartości **fałsz** (kandydat na idola nie zna nikogo) i sprawdzenia, czy kolumna o indeksie będącym numerem kandydata składa się z samych wartości **prawda** (każdy zna idola). Jeśli takiego wiersza i takiej kolumny nie znajdziemy, to w grupie nie ma idola.

Najprostszą metodą znalezienia kandydata na idola jest przeglądanie kolejnych wierszy, a następnie sprawdzenie właściwej kolumny. Pamiętajmy, że nieważne są wartości na głównej przekątnej (taki sam indeks wiersza i kolumny). Możemy w zależności od potrzeb wpisać do tych komórek wartość **fałsz** lub **prawda**, żeby nie sprawdzać dodatkowo, czy indeksy są różne. Algorytm znajdowania idola można zapisać w pseudokodzie następująco:

```
funkcja SzukajIdola(A[][])
    kandydat ← 0
    jest_kandydat ← fałsz
    dopóki kandydat<n oraz nie jest_kandydat wykonuj
        i ← 0
        A[kandydat][kandydat] ← fałsz
        dopóki i<n oraz nie A[kandydat][i] wykonuj
            i ← i + 1
        jeśli i = n to jest_kandydat ← prawda
        w przeciwnym przypadku kandydat ← kandydat + 1
    jeśli nie jest_kandydat to zwróć -1 i zakończ
    // sprawdzenie kolumny o indeksie kandydat
    i ← 0
    A[kandydat][kandydat] ← prawda
    dopóki i<n oraz A[i][kandydat] wykonuj i ← i + 1
    jeśli i = n to zwróć kandydat i zakończ
    w przeciwnym przypadku zwróć -1 i zakończ
```

Dobra rada

Jeśli przynajmniej dwie osoby nie znają nikogo, to w grupie nie ma idola. Gdyby w grupie był idol, to musiałby być znany przez wszystkich, także przez te osoby. Dlatego wystarczy, że sprawdzisz tylko pierwszy znaleziony wiersz wypełniony wartościami **fałsz**.

Dobra rada

Elementy na przekątnej nie mają znaczenia przy naszych rozważaniach, dlatego możesz nadawać im wartości wygodne do sprawdzania kluczowych warunków w algorytmie.

Dobra rada

Jeśli po wykonaniu wewnętrznej pętli zmienna `i` ma wartość `n`, to przejrzane zostały wszystkie elementy wiersza i wszystkie miały wartość **fałsz**. Oznacza to, że znaleźliśmy kandydata.

A to ciekawe

Czy algorytmy wybierają nam muzycznych idoli?

Historie dojścia gwiazd muzyki na szczyty sławy często są barwne i niepowtarzalne. Jednak dzięki rosnącej popularności serwisów strumieniowych oraz inteligentnych głośników to wbudowane w nie algorytmy mogą promować nowych idoli. Technologie te na podstawie metadanych często samodzielnie decydują, które utwory zagrać. Jak donosi firma BuzzAngle, zajmująca się analizą rynku muzycznego, w 2017 r. za 99% wszystkich odtworzeń w serwisach strumieniowych odpowiadało tylko 10% utworów.

Ćwiczenie 6

Zapisz algorytm znajdowania idola tak, żeby najpierw poszukiwał kolumny złożonej z wartości **prawda**, a potem sprawdzał odpowiedni wiersz.

Implementacja algorytmu poszukiwania idola

Podobnie jak podczas poszukiwania lidera, aby znaleźć idola, należy odpowiednio przygotować dane – w tym wypadku tablicę wartości logicznych. Ręczne wprowadzanie danych do tablicy dwuwymiarowej byłoby pracochłonne. Ponadto trzeba zadbać, aby w grupie mógł znajdować się idol.

Stworzymy funkcję losującą wartości tablicy. Wypełnimy ją liczbami 0 i 1, reprezentującymi wartości **fałsz** i **prawda**. Prawdopodobieństwo, że w tak wylosowanej tablicy znajdzie się idol, jest bliskie zeru. Dlatego po wypełnieniu tablicy wylosujemy jeszcze informację, czy w grupie ma się znajdować idol. Jeśli tak, to wylosujemy numer idola i ponownie wypełnimy odpowiednie wiersz i kolumnę tablicy znajomości. Oto kod źródłowy funkcji losującej oraz wypisującej na ekranie utworzoną tablicę:

Dobra rada

Dane możesz przygotować ręcznie, np. w pliku tekstowym. Program powinien na początku wczytać je do tablicy.

Kod źródłowy funkcji losującej tablicę oraz wypisującej jej wartości na ekranie

```
1.  const int N=5;
2.
3.  void Losuj(bool A[][N])
4.  {
5.      int i, j, idol;
6.      for (i=0;i<N;i++)
7.          for (j=0;j<N;j++) A[i][j]=rand()%2;
8.      if (rand()%2==1)
9.      {
10.         idol=rand()%N;
11.         for (i=0;i<N;i++) A[i][idol]=true;
12.         for (j=0;j<N;j++) A[idol][j]=false;
13.     }
14. }
15.
16. void Wypisz(bool A[][N])
17. {
18.     for (int i=0;i<N;i++)
19.     {
20.         for (int j=0;j<N;j++)
21.             if (i!=j) cout<<A[i][j]<<" ";
22.             else cout<<"  ";
23.         cout<<endl;
24.     }
25. }
```

Zwróć uwagę na nagłówki funkcji `Losuj` i `Wypisz` (wiersze 3 i 16). W języku C++, gdy parametrem funkcji jest tablica dwuwymiarowa, nie można pominąć w nawiasach jej drugiego rozmiaru (liczby kolumn). W liniach 6 i 7 losowana jest tablica wartości logicznych (0 reprezentuje fałsz, 1 – prawdę). W wierszu 8 losujemy, czy do tablicy znajomości wpisać idola. Jeśli tak, to jest losowany numer idola (linia 10), a następnie są wypełniane określony wiersz i określona kolumna (linie 11–12).

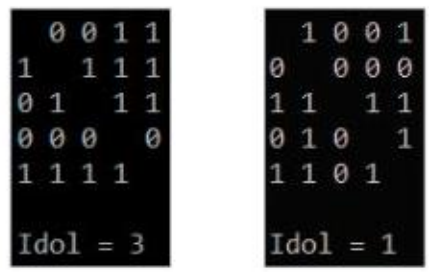
W funkcji `Wypisz` nie są wypisywane elementy leżące na głównej przekątnej, zamiast nich wypisywane są spacje (linie 21–22).

Kod funkcji `SzukajIdola` może być następujący:

Kod źródłowy funkcji poszukującej idola w zbiorze

```
1.  int SzukajIdola(bool A[][N])
2.  {
3.      int i, kandydat=0;
4.      bool jest_kandydat=false;
5.      while (kandydat<N && !jest_kandydat)
6.      {
7.          i=0;
8.          A[kandydat][kandydat]=false;
9.          while (i<N && !A[kandydat][i]) i++;
10.         if (i==N) jest_kandydat=true;
11.         else kandydat++;
12.     }
13.     if (!jest_kandydat) return -1;
14.     i=0; A[kandydat][kandydat]=true;
15.     while (i<N && A[i][kandydat]) i++;
16.     if (i==N) return kandydat;
17.     else return -1;
18. }
```

Pętla w liniach 5–12 poszukuje kandydata na idola, czyli wiersza złożonego z samych wartości **fałsz**. Pętla w linii 15 sprawdza, czy znaleziony kandydat rzeczywiście jest idolem (kolumna złożona z samych wartości **prawda**). Rysunek 16.5 przedstawia przykładowe wywołania programu.



Rys. 16.5. Dwa przykładowe wywołania programu szukającego idola

Ćwiczenie 7

Napisz program znajdujący idola za pomocą omówionego algorytmu z wykorzystaniem funkcji `Losuj`, `Wypisz`, `SzukajIdola`.

16.5. Efektywny algorytm poszukiwania idola

Poprzedni algorytm w najgorszym wypadku musiał przejrzeć raz całą kwadratową tablicę. Poszukajmy efektywniejszego algorytmu.

Zauważ, że przy każdym porównaniu dwóch osób jedną z nich można wykluczyć, ponieważ tylko jedna ma szasnę być idolem. Wystarczy więc wykonać $n - 1$ porównań. Przeanalizujmy znajdowanie kandydata na idola dla tablicy znajomości z rysunku 16.6.

1. Porównujemy osoby o numerach 0 i 1. Kandydatem zostaje 0, 1 nie może być idolem.
2. Porównujemy osoby o numerach 0 i 2. Kandydatem zostaje 2, 0 nie może być idolem.
3. Porównujemy osoby o numerach 2 i 3. Kandydatem zostaje 3, 2 nie może być idolem.
4. Porównujemy osoby o numerach 3 i 4. Kandydatem pozostaje 3, 4 nie może być idolem.

	0	1	2	3	4
0		fałsz	prawda	prawda	fałsz
1	prawda		fałsz	prawda	fałsz
2	prawda	fałsz		prawda	prawda
3	fałsz	fałsz	fałsz		fałsz
4	fałsz	fałsz	prawda	prawda	

Rys. 16.6. Tablica znajomości

Dobra rada

Porównanie znajomości osób *i* oraz *j* możesz rozumieć jako sprawdzenie wartości w *i*-tym wierszu i *j*-tej kolumnie. Jeśli wartością tą jest **fałsz**, to odrzucamy *j*-tą osobę, jeśli **prawda** – odrzucamy *i*-tą osobę.

Następnie należy sprawdzić jeszcze kandydata – zarówno jego wiersz, jak i kolumnę. Oto zapis algorytmu w pseudokodzie:

```
funkcja SzukajIdola(A[][])
    kandydat ← 0
    i ← 1
    // pętla znajdująca kandydata
    dopóki i < n wykonuj
        jeśli A[kandydat][i] to kandydat ← i
        i ← i + 1
    i ← 0
    A[kandydat][kandydat] ← fałsz
    // sprawdzenie wiersza
    dopóki i < n oraz nie A[kandydat][i] wykonuj i ← i + 1
    jeśli i < n to zwróć -1 i zakończ
    i ← 0
    A[kandydat][kandydat] ← prawda
    // sprawdzenie kolumny
    dopóki i < n oraz A[i][kandydat] wykonuj i ← i + 1
    jeśli i = n to zwróć kandydat i zakończ
    w przeciwnym przypadku zwróć -1 i zakończ
```

W zmiennych `kandydat` oraz `i` pamiętane są aktualne numery porównywanych osób. Jeśli element tablicy `A[kandydat][i]` przyjmuje wartość **prawda**, to kandydat zna osobę `i`, więc nie może być idolem. Kandydatem staje się osoba o numerze `i`. Następną osobą do porównania jest osoba o numerze `i + 1`, niezależnie od tego, czy nastąpiła zmiana kandydata.

Oto kod źródłowy funkcji poszukującej idola:

Kod źródłowy funkcji poszukującej idola w zbiorze zgodnie z drugim algorytmem

```
1.  int SzukajIdola(bool A[][N])
2.  {
3.      int kandydat=0, i=1;
4.      while (i<N)
5.      {
6.          if (A[kandydat][i]) kandydat=i;
7.          i++;
8.      }
9.      i=0; A[kandydat][kandydat]=false;
10.     while (i<N && !A[kandydat][i]) i++;
11.     if (i<N) return -1;
12.     i=0; A[kandydat][kandydat]=true;
13.     while (i<N && A[i][kandydat]) i++;
14.     if (i==N) return kandydat;
15.     else return -1;
16. }
```

Pętla w liniach 4–8 wyznacza kandydata na idola. Dwie kolejne pętle (linie 10 i 13) sprawdzają, czy kandydat rzeczywiście jest idolem, czyli czy odpowiedni wiersz (o indeksie określonym wartością zmiennej `kandydat`) jest złożony z samych wartości **fałsz**, a odpowiednia kolumna – z samych wartości **prawda**.

Ćwiczenie 8

Napisz program poszukujący idola, korzystający z powyższej funkcji.

16.6. Porównanie złożoności algorytmów poszukiwania idola

W przypadku pierwszego z omówionych algorytmów jeśli idol ma numer 0, sprawdzamy tylko jeden wiersz i jedną kolumnę, czyli wykonujemy $2 \cdot n$ operacji. Jeśli w zbiorze nie ma idola, przejrzymy wszystkie elementy tablicy, których jest n^2. Jeśli w grupie jest idol, możemy oczekiwać, że prawdopodobnie należy przejrzeć połowę wierszy i dodatkowo jedną kolumnę, a więc wykonać około $n^2/2 + n$ operacji. Złożoność czasowa tego algorytmu, zarówno pesymistyczna, jak i oczekiwana, jest więc kwadratowa: $O(n^2)$.

W drugim algorytmie aby wyłonić kandydata, należy wykonać $n - 1$ porównań, a następnie przejrzeć jeden wiersz i jedną kolumnę. Ulepszony algorytm wykona więc około $3 \cdot n$ operacji. Jego złożoność czasowa jest zatem liniowa: $O(n)$. Dlatego algorytm można uznać za efektywny.

Podsumowanie

- Lider w n-elementowym zbiorze danych to element, który występuje więcej niż $n/2$ razy.
- Idol w grupie osób to osoba, która nie zna w tej grupie nikogo, ale wszyscy inni ją znają.
- Istnieją szybkie algorytmy o złożoności liniowej, które znajdują lidera i idola w zbiorze danych.
- STL to standardowa biblioteka szablonów wielu algorytmów i struktur danych. Jest uniwersalna, tzn. algorytmy można wykonywać dla różnych typów danych, a w strukturach danych mogą być pamiętane dane różnych typów.
- Jedną z często używanych funkcji z biblioteki STL jest funkcja `sort`. Porządkuje ona dane i ma złożoność czasową $O(n \cdot \log n)$. Aby z niej skorzystać, należy w kodzie źródłowym dołączyć bibliotekę `algorithm`.
- Oprócz tablic jednowymiarowych możemy także korzystać z tablic wielowymiarowych. W praktyce mają zastosowanie głównie tablice dwuwymiarowe.

Zadania

* **1** Napisz program, który wylosuje i wypisze tablicę dwuwymiarową liczb całkowitych (o n wierszach i n kolumnach), a następnie znajdzie w niej maksimum.

* **2** Napisz program, który wylosuje i wypisze tablicę dwuwymiarową liczb całkowitych, następnie posortuje każdy wiersz tablicy i ponownie wypisze tablicę.

* **3** Napisz program, który wylosuje i wypisze tablicę dwuwymiarową liczb całkowitych (o n wierszach i n kolumnach), następnie posortuje każdą kolumnę tablicy i ponownie wypisze tablicę.

** **4** Napisz program, który wylosuje i wypisze tablicę dwuwymiarową liczb całkowitych (o n wierszach i n kolumnach), a następnie sprawdzi, czy w tablicy znajdują się wiersz i kolumna o tej samej sumie liczb. Program powinien wypisać numer wiersza i kolumny lub słowo „BRAK".

** **5** Utwórz n-elementową tablicę liczb z zakresu od 0 do $n - 1$. Przygotuj ją tak, żeby z prawdopodobieństwem 50% żaden element tablicy się nie powtarzał. Napisz program, który sprawdzi, czy w tablicy wszystkie elementy są różne. Program powinien wypisać odpowiedni komunikat słowny.

*** **6** Napisz program sprawdzający, czy w tablicy n-elementowej znajduje się liczba, która występuje więcej niż $n/4$ razy. Przygotuj tablicę tak, aby była szansa wystąpienia w niej takiej liczby. Przyjmij, że n jest wielokrotnością 4. Program powinien wykonywać jak najmniej operacji.

Sposób na zadania

Zadanie 1

W pliku, który otrzymasz od nauczyciela (np. *WUZ2_zad1_liczby.txt*), w kolejnych 100 wierszach są zapisane po dwie liczby oddzielone spacją. Pierwsza liczba (z zakresu od 2 do 10) oznacza podstawę pozycyjnego systemu liczbowego, w którym zapisana jest druga liczba (wartość liczby w systemie dziesiętnym mieści się w zakresie od 0 do $2^{16} - 1$). Oto pięć początkowych wierszy pliku:

```
5 2341
2 1111111100000000
9 3333
6 550012
4 3333
```

Napisz program lub programy rozwiązujące poniższe zadania. Odpowiedzi zapisz w sposób wskazany przez nauczyciela.

Zadanie 1.1 (0–3)

Oblicz sumę wszystkich liczb z drugiej kolumny i zapisz wynik w systemie dziesiętnym. Suma pięciu początkowych liczb wynosi 113 709.

Rozwiązanie

Tagi: systemy liczbowe, zamiana reprezentacji liczb między systemami liczbowymi, działania na liczbach w różnych systemach pozycyjnych, kod ASCII

Krok 1
Aby rozwiązać to zadanie, wystarczy odczytywać liczby z kolejnych wierszy pliku, zamieniać je na postać dziesiętną i sumować.

Krok 2
Każda z liczb do sumowania jest nie większa niż 65 535. Ponieważ jest ich 100, suma nie będzie większa niż 6 553 500. Zmienna przechowująca sumę może więc być typu `int`.

Krok 3
Zamianę liczb na postać dziesiętną możemy zrealizować z wykorzystaniem funkcji. Nazwijmy ją `ToDec`. Pobierze ona z pliku podstawę systemu liczbowego `p` i napis `s`, przechowujący liczbę. Podczas przeliczania wartości liczby na postać dziesiętną należy uwzględnić kody ASCII cyfr od 0 do 9, które zaczynają się od 48 (patrz s. 37).

Krok 4
Dane z pliku tekstowego odczytamy z wykorzystaniem zmiennej plikowej `we` typu `ifstream` (patrz s. 336–338).

Odpowiedź: Kod źródłowy programu rozwiązującego to zadanie może być następujący:

```
1. #include <iostream>
2. #include <fstream>
3. #include <string>
4.
5. using namespace std;
6.
```

```
7.  int ToDec(int p, string s)
8.  // wartość dziesiętna liczby s, zapisanej w systemie o podstawie p
9.  {
10.     int w=0;
11.     for (int i=0;i<s.size();i++) w=w*p+s[i]-48;
12.     return w;
13. }
14.
15. int main()
16. {
17.     int p,suma=0;
18.     string s;
19.     ifstream we("WUZ2_zad1_liczby.txt");
20.     for (int i=0;i<100;i++)
21.     {
22.         we>>p>>s;
23.         suma=suma+ToDec(p,s);
24.     }
25.     we.close();
26.     cout<<"Suma: "<<suma;
27.     return 0;
28. }
```

Zadanie 1.2 (0–3)

Podaj najmniejszą i największą liczbę znajdującą się w pliku oraz podstawy systemów, w których te liczby zostały zapisane.

Dla pięciu początkowych liczb z pliku wynikiem jest:

```
Minimum: 4 3333
Maksimum: 2 1111111100000000
```

Rozwiązanie

Tagi: systemy liczbowe, zamiana reprezentacji liczb między systemami liczbowymi, znajdowanie minimum i maksimum

Krok 1

Żeby rozwiązać to zadanie, wystarczy odczytywać kolejne liczby z pliku i porównywać je z aktualną wartością zmiennych pomocniczych przechowujących minimum i maksimum. Porównujemy wartości dziesiętne wszystkich liczb.

Krok 2

Jako wartość początkową zarówno minimum, jak i maksimum możemy przyjąć liczbę z pierwszego wiersza pliku – wówczas porównywanie rozpoczynamy od drugiego wiersza. Możemy też przyjąć wartości graniczne: minimum równe 65 535, maksimum równe 0. Wykorzystamy pierwszy z podanych sposobów.

Krok 3

Aby porównać liczbę odczytaną z pliku z aktualnymi wartościami minimum i maksimum, należy ją zamienić na liczbę w systemie dziesiętnym. Wykorzystamy w tym celu funkcję `ToDec`.

Krok 4

Aktualne wartości minimum i maksimum program powinien także pamiętać w tych systemach liczbowych, w których zostały zapisane w pliku.

Odpowiedź: Kod źródłowy programu może wyglądać następująco:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int ToDec(int p, string s)
// wartość dziesiętna liczby s, zapisanej w systemie o podstawie p
{
    int w=0;
    for (int i=0;i<s.size();i++) w=w*p+s[i]-48;
    return w;
}

int main()
{
    int d,mini,maks,p,pmini,pmaks;
    string s,smini,smaks;
    ifstream we("WUZ2_zad1_liczby.txt");
    we>>p>>s;
    mini=ToDec(p,s); maks=mini;
    smini=s; smaks=s; pmini=p; pmaks=p;
    for (int i=1;i<100;i++)
    {
        we>>p>>s;
        d=ToDec(p,s);
        if (d<mini)
        {
            mini=d; smini=s; pmini=p;
        }
        else if (d>maks)
            {
                maks=d; smaks=s; pmaks=p;
            }
    }
    we.close();
    cout<<"Minimum: "<<pmini<<" "<<smini<<endl;
    cout<<"Maksimum: "<<pmaks<<" "<<smaks;
    return 0;
}
```

Zadanie 1.3 (0–6)

Posortuj wszystkie liczby z drugiej kolumny od najmniejszej do największej, zachowując zapis każdej liczby razem z podstawą systemu, w którym została ona zapisana.

Oto pięć pierwszych liczb po uporządkowaniu:

```
4 3333
5 2341
9 3333
6 550012
2 1111111100000000
```

Rozwiązanie

Tagi: systemy liczbowe, zamiana reprezentacji liczb między systemami liczbowymi, algorytmy sortowania, zapis do pliku

Sposób I. Implementacja własnej funkcji sortującej

Krok 1

Ponieważ sortujemy niewiele liczb, możemy zastosować jedną z metod sortowania prostego, np. przez wybieranie (patrz s. 174). Program powinien wczytać i zapamiętać wszystkie liczby wraz z podstawami systemów, w których zostały zapisane. Wykorzystamy w tym celu tablicę `p` (do przechowania podstaw kolejnych liczb) oraz tablicę `s` elementów typu `string` (do przechowania kolejnych liczb zapisanych w różnych systemach).

Krok 2

Liczby należy porównywać w tym samym systemie. Wygodnie będzie zastosować funkcję `ToDec`, która zapisze liczby w postaci dziesiętnej. Liczby te zapamiętamy w tablicy `d`, aby uniknąć ich wielokrotnego przeliczania przy każdym porównaniu podczas sortowania.

Krok 3

W czasie sortowania, podczas zamiany dwóch liczb dziesiętnych, trzeba pamiętać również o zamianie elementów o tych samych indeksach w tablicach `p` i `s`. Wyniki wygodnie będzie zapisać do pliku z wykorzystaniem zmiennej plikowej `wy` typu `ofstream`.

Odpowiedź: Kod źródłowy programu realizującego opisany algorytm wygląda następująco:

```
1.  #include <iostream>
2.  #include <fstream>
3.  #include <string>
4.
5.  using namespace std;
6.
7.  int ToDec(int p, string s)
8.  // wartość dziesiętna liczby s, zapisanej w systemie o podstawie p
9.  {
10.     int w=0;
11.     for (int i=0;i<s.size();i++) w=w*p+s[i]-48;
12.     return w;
13. }
14.
```

```
15. void sortuj(int p[], string s[], int d[], int n)
16. // sortowanie tablic metodą przez wybieranie
17. // porównywane są elementy tablicy d
18. {
19.     int i,j,k;
20.     for (i=0;i<n-1;i++)
21.     {
22.         k=i;
23.         for (j=i+1;j<n;j++)
24.             if (d[j]<d[k]) k=j;
25.         swap(p[i],p[k]);
26.         swap(s[i],s[k]);
27.         swap(d[i],d[k]);
28.     }
29. }
30.
31. int main()
32. {
33.     const int N=100;
34.     int p[N],d[N];
35.     string s[N];
36.     ifstream we("WUZ2_zad1_liczby.txt");
37.     for (int i=0;i<N;i++)
38.     {
39.         we>>p[i]>>s[i];
40.         d[i]=ToDec(p[i],s[i]);
41.     }
42.     we.close();
43.     sortuj(p,s,d,N);
44.     ofstream wy("zadanie_1_3_I.txt");
45.     for (int i=0;i<N;i++) wy<<p[i]<<" "<<s[i]<<endl;
46.     wy.close();
47.     cout<<"Wyniki w pliku zadanie_1_3_I.txt";
48.     return 0;
49. }
```

Sposób II. Wykorzystanie funkcji `sort` z biblioteki STL

Krok 1

Do sortowania można także wykorzystać funkcję `sort` z biblioteki STL. Zdefiniujemy strukturę o nazwie `liczba`, która będzie zawierała następujące informacje: podstawę systemu liczbowego, zapis liczby w systemie o danej podstawie oraz wartość dziesiętną tej liczby.

Krok 2

Zdefiniujemy własną funkcję o nazwie `porownaj`, której zadaniem będzie porównanie dwóch elementów typu `liczba`.

Krok 3

Dane przechowamy w tablicy struktur o nazwie `t`. Dwie pierwsze informacje: podstawa systemu i zapis liczby zostaną pobrane wprost z pliku, zapis dziesiętny liczby otrzymamy po zastosowaniu funkcji `ToDec`, która oblicza wartość dziesiętną liczby z pliku.

Krok 4

Po posortowaniu liczb za pomocą funkcji **`sort`** z parametrem `porownaj` dane zapiszemy do pliku z wykorzystaniem zmiennej plikowej typu **`ofstream`**.

Odpowiedź: Kod źródłowy programu realizującego opisany algorytm wygląda następująco:

```
1.  #include <iostream>
2.  #include <fstream>
3.  #include <string>
4.  #include <algorithm>
5.
6.  using namespace std;
7.
8.  struct liczba
9.  {
10.     int p;     // p - podstawa systemu liczbowego
11.     string s; // s - zapis liczby w systemie o podstawie p
12.     int d;     // d - wartość dziesiętna liczby s
13. };
14.
15. bool porownaj(liczba a, liczba b)
16. {
17.     return (a.d<b.d);
18. }
19.
20. int ToDec(int p, string s)
21. {
22.     int w=0;
23.     for (int i=0;i<s.size();i++) w=w*p+s[i]-48;
24.     return w;
25. }
26.
27. int main()
28. {
29.     const int N=100;
30.     liczba t[N];
31.     ifstream we("WUZ2_zad1_liczby.txt");
32.     for (int i=0;i<N;i++)
33.     {
34.         we>>t[i].p>>t[i].s;
35.         t[i].d=ToDec(t[i].p,t[i].s);
36.     }
37.     we.close();
38.     sort(t,t+N,porownaj);
```

```
39.     ofstream wy("zadanie_1_3_II.txt");
40.     for (int i=0;i<N;i++) wy<<t[i].p<<" "<<t[i].s<<endl;
41.     wy.close();
42.     cout<<"Wyniki w pliku zadanie_1_3_II.txt";
43.     return 0;
44. }
```

Powyższe zadania można rozwiązać z wykorzystaniem jednego programu, w którym tylko raz korzysta się m.in. z operacji wczytywania danych. Rozwiązanie zadania 1.2 można uzyskać z rozwiązania zadania 1.3 – minimum i maksimum wystarczy odczytać z końców posortowanej tablicy.

Trening

ROZWIĄŻ NA KOMPUTERZE

Zadanie 1

W pliku, który otrzymasz od nauczyciela (np. *WUZ2_zad1_obraz.txt*), znajduje się opis obrazu o wymiarach 96 na 96 pikseli w 16 odcieniach szarości (opis jednego piksela zajmuje 4 bity).

- Wartość `0` (0_{16}) oznacza kolor czarny, a wartość `15` (F_{16}) – kolor biały.
- Pośrednie wartości są określeniami odcieni szarości.
- W pliku znajduje się 96 wierszy, w każdym 24 liczby z zakresu od 0 do $2^{16} - 1$ oddzielone spacjami. Każda liczba opisuje 4 kolejne piksele od lewej do prawej.

W pierwszym wierszu znajdują się następujące liczby:

```
255 65280 0 65535 4080 291 17767 35243 52719 65244 47768 30292 12816 291
17767 35243 52719 65244 47768 30292 12816 255 65280 65535
```

Cztery pierwsze liczby reprezentują piksele w następujących kolorach: pierwsza – 2 piksele czarne i 2 białe ($00FF_{16}$), druga – 2 białe i 2 czarne ($FF00_{16}$), trzecia – 4 czarne (0000_{16}), czwarta – 4 białe ($FFFF_{16}$).

Napisz program lub programy rozwiązujące poniższe zadania. Odpowiedzi zapisz w sposób wskazany przez nauczyciela.

Zadanie 1.1 (0–3)

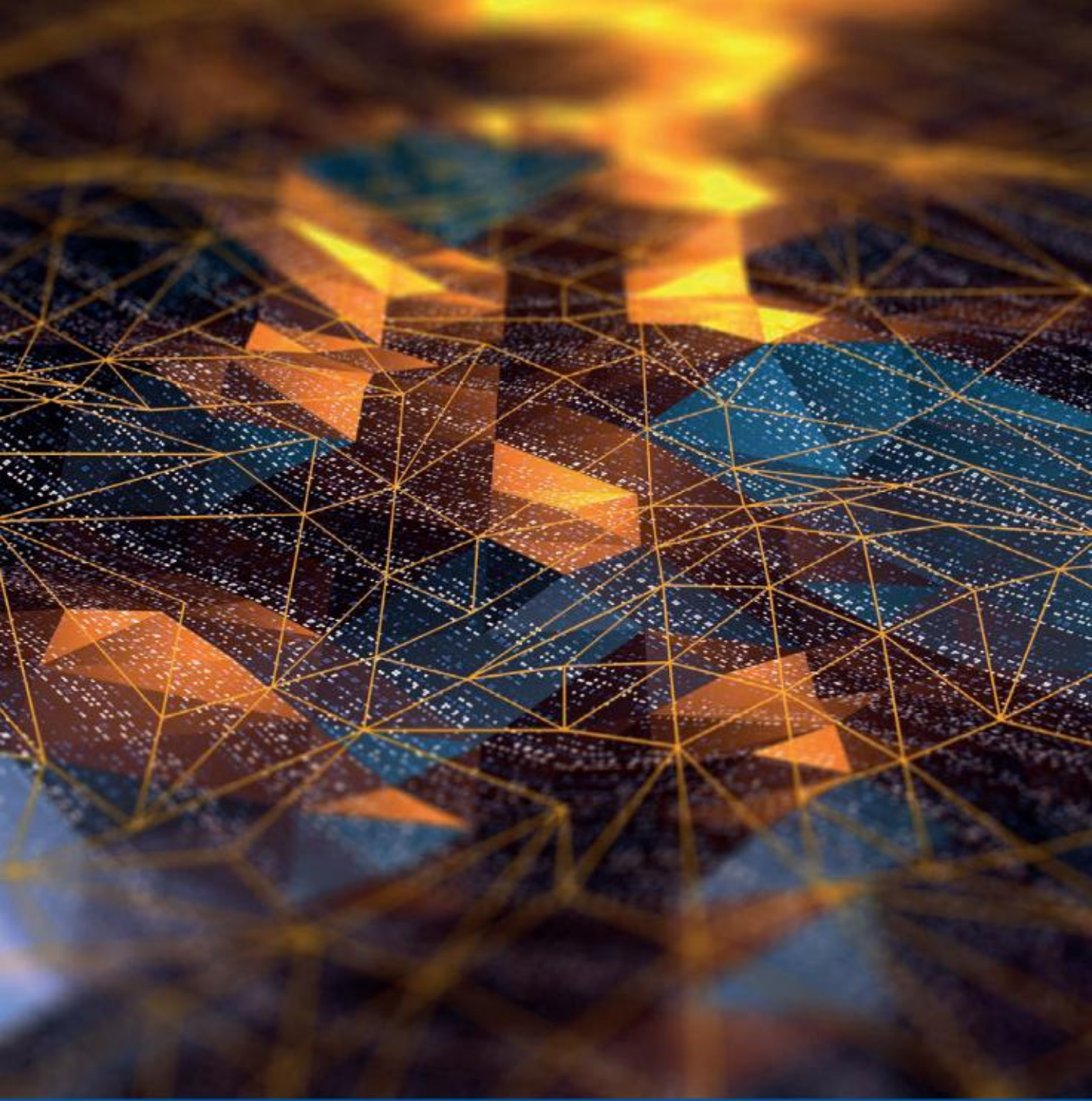
Policz piksele czarne i białe znajdujące się w obrazie. W pierwszej linii (pierwszym wierszu opisującym obraz) znajduje się 18 pikseli czarnych i 22 białe.

Zadanie 1.2 (0–4)

Wyznacz długość najdłuższego poziomego odcinka, czyli ciągu kolejnych pikseli w tym samym kolorze występujących poziomo obok siebie. W pierwszym wierszu taki odcinek jest złożony z 6 pikseli (piksele czarne reprezentowane przez drugą i trzecią liczbę).

Zadanie 1.3 (0–5)

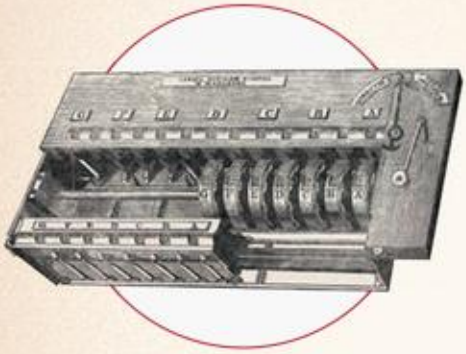
Wyznacz długość najdłuższego pionowego odcinka, czyli ciągu kolejnych pikseli w tym samym kolorze występujących pionowo obok siebie.



3 Metody algorytmiczne

17. Iteracja a rekurencja
18. Metoda zachłanna
19. Programowanie dynamiczne
20. Dziel i zwyciężaj, czyli sortujemy sprawniej
Wiesz, umiesz, zdasz

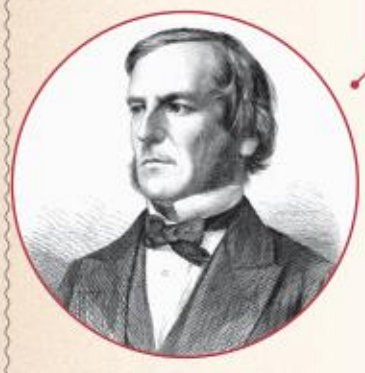
Algorytmika i programowanie



Abraham Izrael Staffel, polski zegarmistrz i wynalazca pochodzenia żydowskiego, w 1845 r. skonstruował **maszynę liczącą** wykonującą sześć działań arytmetycznych. Oprócz dodawania, odejmowania, mnożenia, dzielenia i pierwiastkowania nową funkcjonalnością było potęgowanie. Wynalazek Staffela otrzymał główną nagrodę na wystawie światowej w Londynie w 1851 r.

1845 r.

1847 r.



Angielski matematyk George Boole w pracy *Mathematical Analysis of Logic*, opublikowanej w 1847 r., wprowadził wyrażenia logiczne w formie matematycznej, znanej obecnie jako **algebra Boole'a**. Na jej podstawie współcześnie konstruowane są układy logiczne w komputerach. W dowód uznania dla Boole'a często w językach programowania typ logiczny zmiennych nazywa się *boolean* lub *bool*.

Na potrzeby spisu powszechnego w USA przeprowadzonego w 1890 r. Herman Hollerith skonstruował **maszynę licząco-analityczną** wykorzystującą karty perforowane. Dzięki niej opracowano dużo większy zakres danych w krótszym czasie. Hollerith założył także firmę Tabulating Machine Company. W 1924 r. połączyła się ona z innymi firmami i przekształciła w korporację IBM, która wniosła ogromny wkład w rozwój komputerów i programowania.

1890 r.

1896 r.

W 1896 r. polski wynalazca Jan Szczepanik opracował nowy **model maszyny tkackiej** wynalezionej przez Jacquarda, w której karty perforowane były sterowane automatycznie. Do przygotowania kart wykorzystał metody fotograficzne, a w maszynie zastosował automatyczny (sterowany elektromagnesami) system odczytu kart. Gobelin, którego przygotowanie dotychczas zajmowało ok. 5 tygodni, można było wykonać w 2 godziny.

W 1920 r. wybitny polski matematyk Jan Łukasiewicz, profesor uniwersytetów we Lwowie, Warszawie (także rektor) oraz Dublinie, zaproponował zapis wyrażeń arytmetycznych bez użycia nawiasów, zwany notacją Łukasiewicza lub **notacją polską**. Znak operacji jest w niej umieszczany przed argumentami. Australijski naukowiec Charles Leonard Hamblin odwrócił zapis (znak operacji jest umieszczany za argumentami), tworząc odwrotną notację polską. Znalazła ona zastosowanie w niektórych językach programowania i kalkulatorach.

W 1945 r. trzech naukowców, John von Neumann, John William Mauchly oraz John Presper Eckert, stworzyło logiczny model komputera zwany **architekturą von Neumanna**. W architekturze tej dane przechowywane są wspólnie z instrukcjami i tak samo kodowane, czyli mogą być różnie interpretowane. Wydzielono także trzy podstawowe części: procesor, pamięć i urządzenia wejścia/wyjścia. Komputery w tym modelu mogły być konstruowane do różnych celów, a nie do konkretnego zastosowania.

1920 r.

1938 r.

1945 r.

1950 r.

Za jednego z wynalazców komputera uważa się niemieckiego inżyniera Konrada Zusego, który w 1938 r. skonstruował **mechaniczny komputer Z1**. Był to pierwszy programowalny komputer wykorzystujący binarne liczby zmiennoprzecinkowe. Jego unowocześniona wersja (Z3), skonstruowana podczas II wojny światowej, była używana przy projektowaniu skrzydeł samolotów.

W ramach rozważań nad sztuczną inteligencją Alan Turing zaproponował test sprawdzający, czy maszyna może przejawiać inteligentne zachowania tak jak człowiek. Dziś nazywany jest on **testem Turinga**. Polega on na tym, że człowiek sędzia rozmawia (pisemnie) równocześnie z innym człowiekiem oraz z programem komputerowym. Jeśli człowiek nie zdemaskuje komputera, to taki program zalicza test. Dzisiaj odwrócony test Turinga wykorzystywany jest w technice CAPTCHA: w roli sędziego jest komputer, który daje użytkownikowi do rozwiązania zadanie – łatwe dla człowieka, ale trudne dla maszyny.

17. Iteracja a rekurencja

Do rozwiązywania wielu zagadnień algorytmicznych stosuje się iterację, czyli instrukcję polegającą na powtarzaniu wskazanych operacji. Istnieje jednak inna technika, która również wiąże się z powtarzaniem, ale nie wykorzystuje pętli. Jest nią rekurencja. W tym temacie dowiesz się, czym jest rekurencja, jakie ma zastosowania, a także dlaczego należy jej używać z rozwagą.

Cele lekcji

- Zrozumiesz zasadę działania rekurencji.
- Zaprogramujesz w języku C++ wybrane algorytmy rekurencyjne.
- Porównasz wersje iteracyjne i rekurencyjne wybranych algorytmów.
- Poznasz zasady poprawnego konstruowania algorytmów rekurencyjnych.
- Dowiesz się, czym się charakteryzują liczby Fibonacciego, oraz poznasz zasadę złotego podziału.
- Zrozumiesz rozszerzony algorytm Euklidesa.

Rekurencja (rekursja)

Rekurencja lub **rekursja** (ang. *recursion*) w informatyce oznacza odwołanie się w rozwiązaniu problemu do tego samego problemu, ale dla mniejszego rozmiaru danych. Innymi słowy, zakładamy, że znamy rozwiązanie problemu np. dla danych rzędu $n - 1$ i wykorzystujemy to, aby sformułować rozwiązanie dla danych rzędu n.

Nazwa „rekurencja" pochodzi od łacińskiego słowa *recurrere*, czyli „przybiec z powrotem". Kluczem do zrozumienia działania rekurencji są właśnie powroty z wywołania rekurencyjnego.

Warto wiedzieć

Efekt podobny do rekurencyjnego można zauważyć, stojąc plecami do dużego lustra i robiąc zdjęcie przednim aparatem telefonu (tzw. selfie).

17.1. Rekurencja w matematyce

Na lekcjach matematyki spotkaliście się lub spotkacie ze wzorami rekurencyjnymi, służącymi m.in. do obliczania wartości n-tego wyrazu ciągu. Rozważmy ciąg a_n dany wzorem:

$$a_n = \begin{cases} 2 & \text{dla } n = 1 \\ a_{n-1} + 3 & \text{dla } n > 1 \end{cases}$$

Pierwszy element tego ciągu – oznaczony jako a_1 – ma wartość 2. Aby wyznaczyć wartość a_2, należy znać wartość elementu a_1 i wykorzystać ją do obliczenia wartości według wzoru: $a_2 = a_1 + 3$. Wartość a_2 wynosi więc 5. Kolejne wyrazy ciągu a_n wyznacza się analogicznie.

Z drugiej strony, aby obliczyć wartość a_n dla danego n, należy wcześniej wyznaczyć wyraz a_{n-1}, ale żeby obliczyć tę wartość, należy znać wyraz a_{n-2} itd. Ciąg a_n jest **ciągiem arytmetycznym**, ponieważ różnica między wszystkimi kolejnymi jego wyrazami jest stała (w tym przypadku wynosi 3). Aby to wykazać, wystarczy skorzystać ze wzoru dla każdego n i sprawdzić różnicę między a_n a a_{n-1}.

Ciąg arytmetyczny

Zwróć uwagę, że odwoływanie się do poprzednich wyrazów ciągu w celu obliczenia wartości dla danego n powinno się kiedyś skończyć. We wzorze określona jest wartość dla pierwszego elementu ciągu ($n = 1$) – w informatyce nazywa się to **warunkiem początkowym**. Dzięki niemu mamy pewność, że rekurencja się zakończy.

Wartości wyrazów ciągów zdefiniowanych rekurencyjnie można łatwo obliczyć w arkuszu kalkulacyjnym. Dla powyższego ciągu wystarczy wpisać w komórce A1 wartość wyrazu początkowego, czyli 2, a w komórce A2 – formułę opisującą zależność rekurencyjną: =A1+3, i skopiować formułę w dół. Efekt widać na rysunku 17.1.

	A
1	2
2	=A1+3
3	8
4	11
5	14
6	17
7	20
8	23
9	26

Rys. 17.1. Ciąg rekurencyjny w arkuszu kalkulacyjnym

Warunek początkowy

Ćwiczenie 1

a. Zapisz wzór ogólny ciągu a_n.

b. Zapisz wzór rekurencyjny ciągu o początkowych wyrazach: $b_1 = 2$, $b_2 = 7$, $b_3 = 22$, $b_4 = 67$, w którym każdy wyraz zależy od wyrazu poprzedniego. Zastosuj ten wzór do wyznaczenia wartości b_7.

Obliczanie silni

Silnię liczby całkowitej nieujemnej oznaczamy symbolem $n!$ i opisujemy następującym wzorem:

Silnia

$$n! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot \ldots \cdot (n-1) \cdot n$$

Silnię możemy łatwo zapisać w sposób rekurencyjny:

$$n! = \begin{cases} 1 & \text{dla } n < 2 \\ (n-1)! \cdot n & \text{dla } n \geqslant 2 \end{cases}$$

Warto wiedzieć

Choć może się to wydawać nienaturalne, przyjmuje się, że $0! = 1$.

Zapiszmy w pseudokodzie funkcję obliczającą rekurencyjnie wartość $n!$.

Specyfikacja

Dane: n – liczba całkowita, $0 \leqslant n \leqslant 20$.

Wynik: $n!$

Dobra rada

Wartość silni bardzo szybko rośnie. Dlatego w programach, w których wykorzystujesz silnię, ogranicz zakres danych.

```
funkcja SilniaRek(n)
    jeśli n < 2 zwróć 1 i zakończ
    w przeciwnym przypadku zwróć SilniaRek(n–1) * n i zakończ
```

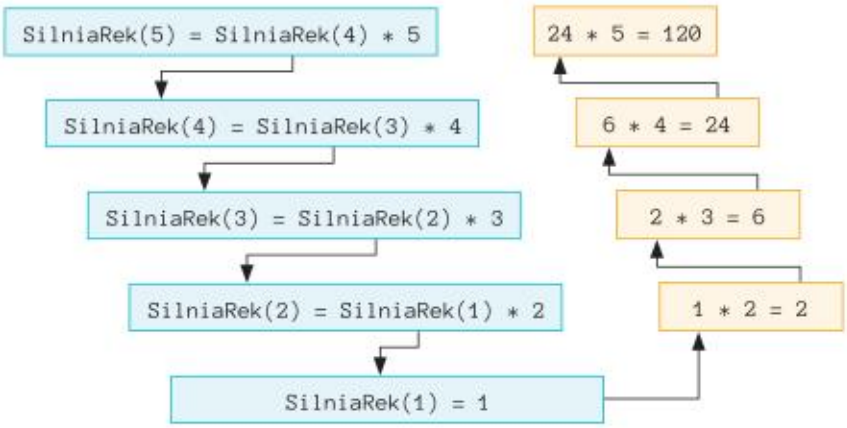
Zwróć uwagę, że powyższa funkcja wywołuje samą siebie z inną wartością parametru. Takie funkcje nazywamy **funkcjami rekurencyjnymi**.

Funkcja rekurencyjna

W językach programowania rekurencja jest realizowana właśnie z wykorzystaniem funkcji. Gdyby zabrakło sprawdzenia warunku początkowego (pierwsza linia bloku instrukcji funkcji w pseudokodzie), mielibyśmy do czynienia z **rekurencją nieskończoną**. Funkcja wywoływałaby samą siebie w nieskończoność.

Rekurencja nieskończona

Rysunek 17.2 przedstawia sposób obliczenia wartości 5! z wykorzystaniem funkcji `SilniaRek`.



Rys. 17.2. Graficzna interpretacja obliczania wartości 5! z wykorzystaniem funkcji rekurencyjnej

Wartość 5! zostanie obliczona po wywołaniu funkcji `SilniaRek(5)`. Ponieważ parametr nie jest mniejszy od 2, zostanie wywołana kopia funkcji z parametrem 4. Takie wywołania będą powtarzane do momentu wykonania funkcji z parametrem 1 (warunek początkowy), którego wynikiem będzie 1. Obliczona wartość jest zwracana do miejsca wywołania, a więc do kopii wywołania funkcji z parametrem 2, wynik pomnożony przez 2 wraca do kopii wywołania funkcji z parametrem 3 itd., aż do wywołania funkcji z parametrem 5. Po pomnożeniu przez 5 otrzymujemy końcową wartość funkcji. W pomarańczowych prostokątach znajdują się wartości generowane przy powrotach z rekurencji.

A to ciekawe

Efekt Droste – rekurencja w projektowaniu

Około 1900 r. na puszce kakao produkowanego przez holenderską firmę cukierniczą Droste pojawił się wizerunek pielęgniarki, która trzymała w ręku tacę z filiżanką oraz puszką kakao Droste. Na tej puszce widać pielęgniarkę trzymającą w ręku tacę z filiżanką oraz puszką kakao Droste itd. Ten sposób prezentacji grafiki został pod koniec lat 70. XX w. nazwany efektem Droste. Jest to rodzaj rekurencyjnego obrazu. Również dziś można zaobserwować ten zabieg na opakowaniach produktów spożywczych, m.in. amerykańskiego proszku do pieczenia i polskiego mleka zagęszczonego.

Ćwiczenie 2

Przygotuj arkusz kalkulacyjny obliczający wartości silni dla kolejnych liczb całkowitych nieujemnych mniejszych od 15.

Oto kod źródłowy programu, który oblicza wartość silni dla podanej wcześniej specyfikacji z wykorzystaniem funkcji rekurencyjnej. Wartością funkcji jest liczba **typu `long long`**. Typ ten pozwala obsługiwać liczby całkowite o dużych wartościach.

Typ `long long`, s. 328

Kod źródłowy programu obliczającego rekurencyjnie silnię

```
1.  #include<iostream>
2.
3.  using namespace std;
4.
5.  long long SilniaRek(short n)
6.  {
7.      if (n<2) return 1;
8.      return SilniaRek(n-1)*n;
9.  }
10.
11. int main ()
12. {
13.     short n;
14.     cout<<"n = "; cin>>n;
15.     cout<<n<<"! = "<<SilniaRek(n);
16.     return 0;
17. }
```

Rodzaj rekurencji, który zastosowano w powyższych przykładach do obliczania wartości według wzorów matematycznych, nazywa się **rekurencją ogonową** (ang. *tail call*). Występuje w niej dokładnie jedno wywołanie rekurencyjne – jest to ostatnia instrukcja funkcji.

Rekurencja ogonowa

Zapiszmy teraz w pseudokodzie funkcję `SilniaIter`, obliczającą iteracyjnie wartość $n!$.

```
funkcja SilniaIter(n)
    silnia ← 1
    dla i ← 2, 3, ..., n wykonuj
        silnia ← silnia * i
    zwróć silnia i zakończ
```

Podobnie jak silnię, każdy algorytm wykorzystujący rekurencję ogonową można zapisać iteracyjnie za pomocą jednej pętli.

Warto wiedzieć

W praktyce rzadko spotyka się algorytmy wykorzystujące rekurencję ogonową. Zastępuje się ją iteracją.

Ćwiczenie 3

Zapisz w języku C++ program obliczający iteracyjnie wartość $n!$.

17.2. Podział liczby na cyfry z wykorzystaniem rekurencji

Do tej pory stosowaliśmy algorytmy, które przeglądały cyfry liczby od najmniej znaczącej. Napiszemy teraz program, który z wykorzystaniem rekurencji wypisze cyfry liczby dziesiętnej od lewej strony do prawej, czyli od cyfry najbardziej znaczącej do cyfry jedności. Program nie będzie zapamiętywać cyfr w zmiennej pomocniczej.

Warto wiedzieć

W algorytmach wyodrębniania cyfr oraz szybkiego podnoszenia do potęgi bez wykorzystania rekurencji cyfry danej liczby przeglądane są od prawej do lewej.

Sformułujmy warunek zakończenia algorytmu (czyli warunek początkowy): jeśli liczba jest mniejsza od 10, to program powinien ją po prostu wypisać. Jeśli liczba jest większa niż 10, to można rekurencyjnie rozwiązać problem dla liczby podzielonej całkowicie przez 10 (czyli bez ostatniej cyfry). W ten sposób wypiszemy wszystkie cyfry liczby z wyjątkiem cyfry jedności. Następnie należy wypisać tę cyfrę, a więc resztę z dzielenia liczby przez 10. Oto specyfikacja i pseudokod:

Specyfikacja

Dane: n – liczba całkowita, $n \geqslant 0$.

Wynik: cyfry liczby n wypisane od najbardziej znaczącej do cyfry jedności, czyli od lewej do prawej.

```
funkcja CyfryRek(n)
    jeśli n < 10 to wypisz n
    w przeciwnym przypadku
        CyfryRek(n div 10)
        wypisz n mod 10
    zakończ
```

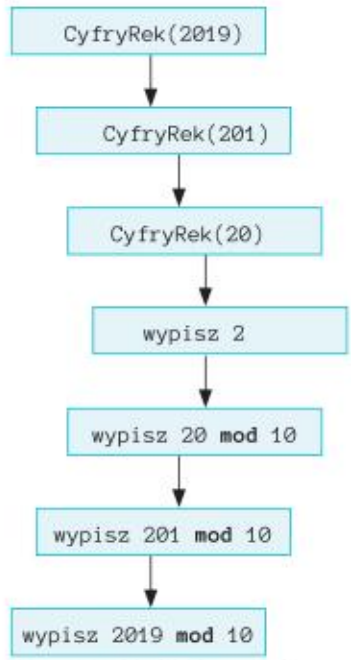
Zwróć uwagę, że wypisywanie cyfr następuje po wywołaniu rekurencyjnym, a więc w trakcie powrotów.

Wywołania rekurencyjne funkcji `CyfryRek` prześledzimy na przykładzie liczby 2019. Wartości pośrednie: 2019, 201, 20 i 2 nie są pamiętane jawnie w zmiennej – powyższa funkcja nie korzysta z żadnej zmiennej. Wartości te jednak muszą być gdzieś przechowywane. Zapewnia to mechanizm rekurencji.

Rysunek 17.3 pokazuje układ kolejnych wywołań funkcji rekurencyjnej `CyfryRek`.

Dla każdego wywołania funkcji pamiętane są aktualne wartości parametrów oraz tzw. **adres powrotu**, który określa, gdzie należy powrócić po zakończeniu danego wywołania rekurencyjnego.

Adres powrotu



Rys. 17.3. Funkcja wypisująca cyfry od najbardziej znaczących

Oto kod źródłowy funkcji CyfryRek:

```
1.  void CyfryRek(int n)
2.  {
3.      if (n<10) cout<<n<<endl;
4.      else
5.      {
6.          CyfryRek(n/10);
7.          cout<<n%10<<endl;
8.      }
9.  }
```

Kod źródłowy funkcji wyznaczającej rekurencyjnie cyfry liczby od najbardziej znaczącej

Zauważ, że zapis warunku początkowego znajduje się w linii 3 kodu źródłowego i jest sprawdzany na samym początku działania funkcji.

Ćwiczenie 4

Napisz funkcję rekurencyjną `void DecToBinRek(int n)`, wypisującą cyfry binarne dziesiętnej liczby całkowitej nieujemnej od najbardziej znaczącej do najmniej znaczącej, czyli od lewej do prawej.

Zapamiętaj

Stosowanie rekurencji powoduje duże zużycie pamięci komputera. Im więcej zagłębień rekurencyjnych, tym większe obciążenie pamięci. Dlatego jeśli można rozwiązać dany problem iteracyjnie, bez wykorzystywania dodatkowych struktur danych, warto to zrobić.

17.3. Rekurencyjny algorytm szybkiego podnoszenia do potęgi

W temacie 3 omówiliśmy **iteracyjny algorytm szybkiego podnoszenia do potęgi**, wykorzystujący rozwinięcie binarne wykładnika.

Algorytm szybkiego podnoszenia do potęgi, s. 57–58

Na przykład:

$$x^{13} = x^{1101_2} = x^{8+4+1} = x^8 \cdot x^4 \cdot x^1$$

Żeby wyznaczyć wartość x^{13}, wystarczyło obliczyć kolejno:

$$x^2 = x \cdot x,\ x^4 = x^2 \cdot x^2,\ x^8 = x^4 \cdot x^4$$

Łącznie wykonaliśmy pięć mnożeń. Wartość x^{13} możemy też zapisać następująco:

$$x^{13} = (x^6)^2 x = ((x^3)^2)^2 x = ((x^2 x)^2)^2 x$$

Ćwiczenie 5

Wyznacz liczbę operacji mnożenia w ostatniej równości powyższego wzoru.

Jeśli wykładnik potęgi jest parzysty, to obliczamy potęgę o wykładniku podzielonym przez 2 i podnosimy ją do kwadratu. Dla wykładnika nieparzystego dodatkowo mnożymy wynik przez podstawę potęgi.

Ten sposób postępowania można wyrazić wzorem rekurencyjnym:

$$x^n = \begin{cases} x & \text{dla } n = 1 \\ (x^{n \text{ div } 2})^2 & \text{dla } n \text{ parzystego} \geqslant 2 \\ x \cdot (x^{n \text{ div } 2})^2 & \text{dla } n \text{ nieparzystego} \geqslant 3 \end{cases}$$

Wzór ten możemy zapisać wprost jako funkcję w pseudokodzie:

Specyfikacja

Dane: n – liczba całkowita, $n \geqslant 1$, x – liczba całkowita.

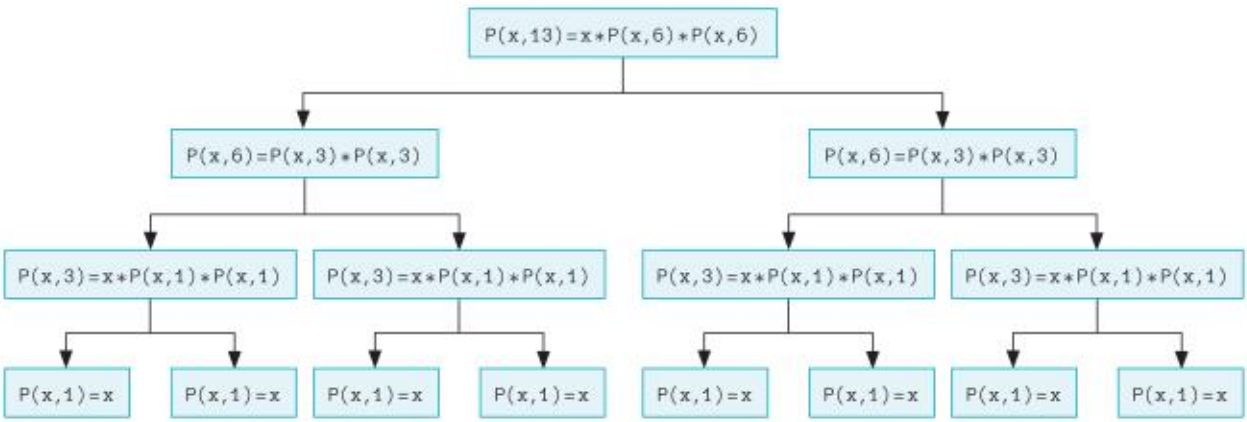
Wynik: x^n

Warto wiedzieć

Przedstawiony algorytm podnoszenia do potęgi można zastosować także wtedy, gdy podstawa jest liczbą rzeczywistą.

```
funkcja PotegaRek(x,n)
    jeśli n = 1 to zwróć x i zakończ
    jeśli n mod 2 = 0 to
        zwróć PotegaRek(x,n div 2)*PotegaRek(x,n div 2)
        i zakończ
    zwróć x*PotegaRek(x,n div 2)*PotegaRek(x,n div 2)
    i zakończ
```

Sprawdźmy, ile operacji mnożenia wykona powyższa funkcja dla $n = 13$. Rysunek 17.4 przedstawia schemat wywołań rekurencyjnych funkcji PotegaRek (na rysunku oznaczonej skrótem P) dla potęgi o wykładniku 13.



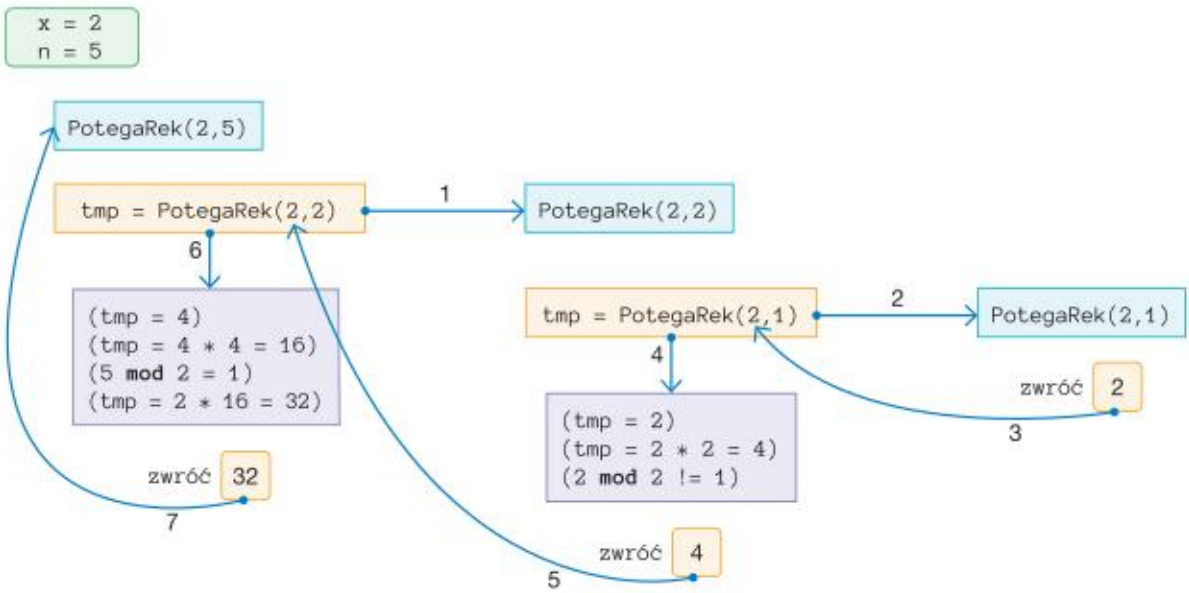
Rys. 17.4. Schemat rekurencyjnego wywoływania funkcji P, obliczającej potęgę liczby x o wykładniku 13

Zauważ, że duża liczba operacji mnożenia wynika z wielokrotnego wyliczania potęgi dla tego samego wykładnika, czyli rekurencyjnego wywoływania funkcji dla tych samych parametrów. Funkcja P zostanie wywołana dla parametrów (x,1) aż osiem razy.

Funkcję PotegaRek można zmodyfikować w taki sposób, aby zapamiętywać wynik wywołania rekurencyjnego w zmiennej pomocniczej, np. o nazwie tmp. Poniżej znajduje się pseudokod, w którym wykorzystano zmienną tmp do pojedynczego obliczania powtarzających się wywołań rekurencyjnych.

```
funkcja PotegaRek(x,n)
    jeśli n = 1 to zwróć x i zakończ
    tmp ← PotegaRek(x,n div 2)
    tmp ← tmp * tmp
    jeśli n mod 2 = 1 to tmp ← x * tmp
    zwróć tmp i zakończ
```

Rysunek 17.5 pokazuje, w jaki sposób realizowane są kolejne wywołania rekurencyjne oraz jak wykorzystywane są wartości obliczane przy powrotach z rekurencji. Funkcja PotegaRek została użyta do obliczenia wartości 2^5. Numery przy strzałkach oznaczają kolejność wywołań rekurencyjnych i obliczeń.



Rys. 17.5. Schemat kolejnych wywołań rekurencyjnych funkcji PotegaRek

Oto kod źródłowy funkcji PotegaRek:

Kod źródłowy funkcji obliczającej potęgę liczby

```
1. long long PotegaRek(int x, int n)
2. {
3.     if (n==1) return x;
4.     long long tmp=PotegaRek(x,n/2);
5.     tmp=tmp*tmp;
6.     if (n%2==1) tmp=x*tmp;
7.     return tmp;
8. }
```

Typ `long long`, s. 328

Zwróć uwagę na użyte typy danych. Ponieważ wartość potęgi szybko rośnie, został użyty **typ** `long long`. W linii 4 następuje wywołanie rekurencyjne funkcji `PotegaRek` z parametrami `x` oraz `n/2`, której wynik jest przypisywany zmiennej `tmp`.

W wersji rekurencyjnej algorytmu przyjęliśmy, że $n \geqslant 1$. Dla $n = 0$ funkcja nie będzie działać prawidłowo – nastąpią nieskończone wywołania rekurencyjne i w efekcie pojawi się błąd – przekroczenie pamięci dostępnej dla programu. Warunek końca można zastąpić warunkiem:

```
jeśli n = 0 to zwróć 1 i zakończ
```

Spowoduje to jednak wykonywanie dodatkowych operacji mnożenia przy obliczaniu każdej potęgi (podnoszenie 1 do kwadratu i mnożenie przez x). W takiej sytuacji warto ten szczególny przypadek rozpatrywać poza funkcją rekurencyjną.

Zapamiętaj

Jeśli wzór zdefiniowany rekurencyjnie zamienimy wprost na funkcję rekurencyjną, może to doprowadzić do wielokrotnego wyliczania tych samych wartości, a wtedy algorytm będzie nieefektywny.

17.4. Jak wejść na wieżę Eiffla, czyli liczby Fibonacciego

Rozważmy następujący problem. Wybraliście się na wycieczkę do Paryża i próbujecie wejść schodami na wieżę Eiffla. Załóżmy, że jednym krokiem można pokonać jeden lub dwa schodki. Na ile sposobów da się wejść na n-ty schodek wieży? Pokazuje to rysunek 17.6.

Rys. 17.6. Sposób wejścia na n-ty schodek wieży

Istnieje tylko jeden sposób wejścia na pierwszy schodek wieży. Na drugi schodek można wejść już na dwa sposoby: trzeba zrobić dwa kroki po jednym schodku lub jeden krok przez dwa schodki.

Warto wiedzieć

Leonardo z Pizy (1175–1250), zwany Fibonaccim, był włoskim matematykiem. Opracował wiele dzieł, ale do najważniejszych należy traktat *Liber abaci* (*Księga liczydła* lub *Księga rachunków*), w którym opisał system pozycyjny liczb oraz podstawy arytmetyki.

Na n-ty schodek można wejść ze schodka $n-1$ (jednym krokiem pokonujemy jeden schodek) lub ze schodka $n-2$ (jednym krokiem pokonujemy dwa schodki). Zatem możliwości wejścia na n-ty schodek jest tyle, ile jest łącznie możliwości wejścia na $n-1$ oraz $n-2$ schodek.

Możemy to zapisać wzorem rekurencyjnym:

$$s_n = \begin{cases} 1 & \text{dla } n = 1 \\ 2 & \text{dla } n = 2 \\ s_{n-1} + s_{n-2} & \text{dla } n > 2 \end{cases}$$

W 1202 r. włoski matematyk Leonardo z Pizy, zwany Fibonaccim, w swoim dziele *Liber abaci* opisał następujący ciąg liczbowy: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ..., w którym każdy kolejny wyraz ciągu jest sumą dwóch poprzednich. Ciąg ten nazywany jest **ciągiem Fibonacciego** i określony jest wzorem:

Ciąg Fibonacciego

$$f_n = \begin{cases} 0 & \text{dla } n = 0 \\ 1 & \text{dla } n = 1 \\ f_{n-1} + f_{n-2} & \text{dla } n > 1 \end{cases}$$

Ćwiczenie 6

a. Porównaj ciągi s_n i f_n. Jaka jest zależność między nimi?

b. Przygotuj arkusz kalkulacyjny obliczający wartości kolejnych liczb Fibonacciego.

Podobnie jak w przypadku innych wzorów rekurencyjnych, powyższy wzór można zapisać w pseudokodzie jako funkcję rekurencyjną. Będzie ona wyznaczać n-tą liczbę Fibonacciego.

Przyjmijmy, że ciąg Fibonacciego rozpoczyna się od 1.

Warto wiedzieć

Ciąg Fibonacciego często definiuje się od 1:

$$f_n = \begin{cases} 1 & \text{dla } n = 1 \text{ lub } n = 2 \\ f_{n-1} + f_{n-2} & \text{dla } n \geqslant 3 \end{cases}$$

Specyfikacja

Dane: n – liczba całkowita, $1 \leqslant n \leqslant 50$.

Wynik: f_n – n-ta liczba Fibonacciego: $f_n = \begin{cases} 1 & \text{dla } n < 3 \\ f_{n-1} + f_{n-2} & \text{dla } n \geqslant 3 \end{cases}$

```
funkcja FibRek(n)
    jeśli n < 3 to zwróć 1 i zakończ
    zwróć FibRek(n-1) + FibRek(n-2) i zakończ
```

Dobra rada

Liczby w ciągu Fibonacciego rosną bardzo szybko. Dlatego w programach wykorzystujących ten ciąg ogranicz zakres danych.

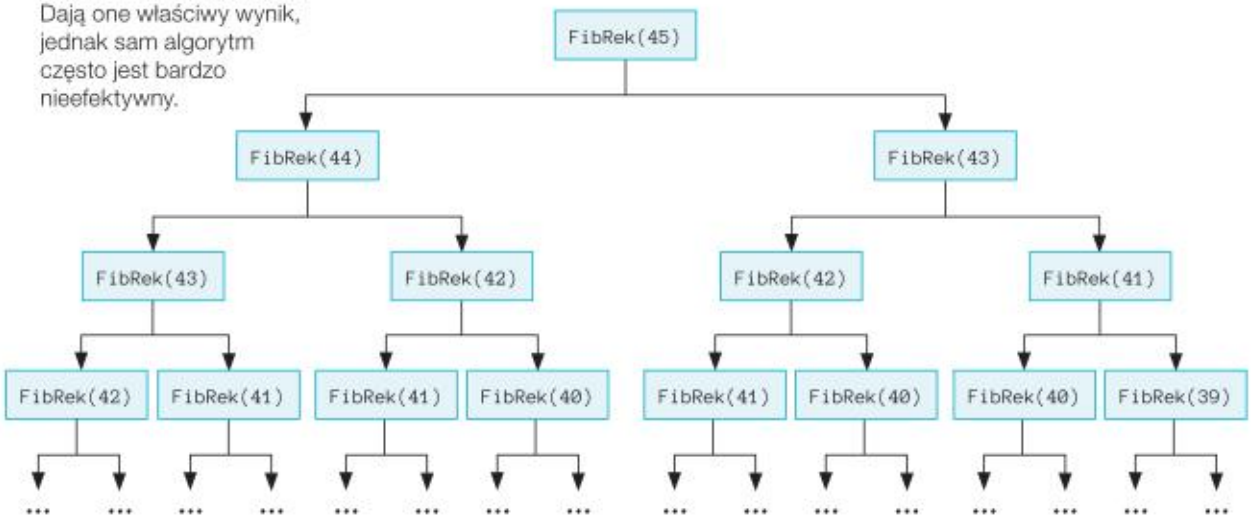
Ćwiczenie 7

Napisz program według powyższej specyfikacji z wykorzystaniem funkcji FibRek. Sprawdź działanie programu dla liczb: 5, 10 i 45.

Dobra rada

Unikaj przepisywania wprost rekurencyjnych wzorów matematycznych. Dają one właściwy wynik, jednak sam algorytm często jest bardzo nieefektywny.

Zauważ, że oczekiwanie na 45. liczbę Fibonacciego trwa długo. Przyczyna, podobnie jak w przykładzie z obliczaniem potęgi, leży w wielokrotnym wyznaczaniu tych samych liczb Fibonacciego (rys. 17.7).



Rys. 17.7. Schemat wywołań funkcji `FibRek` dla $n = 45$

Ćwiczenie 8

a. Ile razy zostanie wywołana funkcja FibRek z rysunku 17.7 dla kolejnych liczb mniejszych od 45?

b. Ustal, ile operacji dodawania wystarczy wykonać, aby wyznaczyć n-tą liczbę Fibonacciego ręcznie na kartce papieru lub w arkuszu kalkulacyjnym.

Liczby Fibonacciego – wersja iteracyjna algorytmu

Zauważ, że aby wyznaczyć kolejną liczbę Fibonacciego, wystarczy w danym momencie pamiętać dwie ostatnie liczby. Obliczanie wartości kolejnych liczb można zapisać za pomocą pojedynczej pętli. Oto zapis funkcji w pseudokodzie:

Warto wiedzieć

Kolejną liczbę Fibonacciego na podstawie dwóch poprzednich można także wyznaczyć bez użycia zmiennej pomocniczej. Wystarczy w tym celu wykonać dodatkowe odejmowanie. Realizuje to sekwencja instrukcji:
`f2 ← f1 + f2`
`f1 ← f2 – f1`

```
funkcja FibIter(n)
    f1 ← 1
    f2 ← 1
    dla i ← 3, 4, ..., n wykonuj
        tmp ← f2
        f2 ← f1 + f2
        f1 ← tmp
    zwróć f2 i zakończ
```

W zmiennych `f1` i `f2` pamiętane są dwie ostatnie liczby Fibonacciego na danym etapie obliczeń. Kiedy program wyznacza kolejną liczbę, w zmiennej pomocniczej `tmp` przechowywana jest aktualnie ostatnia liczba, która w kolejnym kroku będzie liczbą przedostatnią.

W zmiennej `f2` wyliczana jest wartość nowej liczby jako suma dwóch ostatnich. Dotychczasowa wartość `f1` nie jest już potrzebna, ponieważ jest to w danym momencie trzecia liczba od końca. Dlatego zmienna `f1` przyjmuje wartość przedostatniej liczby pamiętanej tymczasowo w zmiennej `tmp`.

Oto kod źródłowy programu wyznaczającego *n*-ty wyraz ciągu Fibonacciego z wykorzystaniem funkcji `FibIter`:

Kod źródłowy programu wyznaczającego *n*-ty wyraz ciągu Fibonacciego

```
#include<iostream>

using namespace std;

long long FibIter(short n)
{
    long long f1=1, f2=1, tmp;
    for (int i=3;i<=n;i++)
        {
            tmp=f2;
            f2=f1+f2;
            f1=tmp;
        }
    return f2;
}

int main ()
{
    short n;
    cout<<"n = "; cin>>n;
    cout<<"f("<<n<<") = "<<FibIter(n);
    return 0;
}
```

W nagłówku funkcji `FibIter` użyto typu **`long long`**, ponieważ wynikiem działania funkcji może być duża liczba całkowita. Parametrami funkcji będą stosunkowo małe liczby, dlatego można w ich wypadku zastosować **typ `short`**.

Typ `short`, s. 328

Ćwiczenie 9

Przygotuj arkusz kalkulacyjny, który obliczy ilorazy kolejnych liczb Fibonacciego $\frac{f_n}{f_{n-1}}$. Do jakiej wartości zbliżają się kolejne ilorazy?

Fenomen liczb Fibonacciego polega na tym, że bardzo często występują wokół nas. Odnajdziemy je w przyrodzie, architekturze i sztuce. Ciekawym zagadnieniem związanym z liczbami Fibonacciego jest tzw. **złota proporcja**, o której możesz przeczytać na s. 248–249.

Złota proporcja

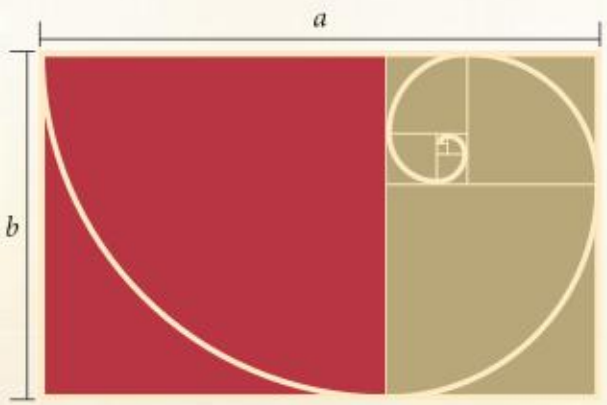
Złota proporcja

Złota proporcja, nazywana też złotym podziałem, znana jest od wieków. Przyrodnicy zachwycają się jej pięknem, którego przykłady można odnaleźć w naturze. Zainspirowało to wiele osób do poszukiwania i stosowania złotego podziału w różnych dziedzinach życia.

Czym jest złota proporcja?

Złota proporcja wyraża podział odcinka na dwie części w taki sposób, że stosunek jego dłuższej części (x) do krótszej (y) jest taki sam jak stosunek długości całego odcinka ($x + y$) do części dłuższej (x). Stosunek ten oznacza się grecką literą φ (czytaj: fi). Co ciekawe, przybliżeniami liczby φ są ilorazy kolejnych liczb ciągu Fibonacciego: 3/2, 5/3, 8/5 itd.

$$\varphi = \frac{x}{y} = \frac{x+y}{x} = \frac{1+\sqrt{5}}{2} = 1{,}618...$$



Jeżeli w prostokąt o bokach a, b będących w złotej proporcji wpiszemy kwadrat (zaznaczony na czerwono), to uzyskamy mniejszy prostokąt (na rysunku złoty) o bokach pozostających w złotej proporcji. Jeżeli powtórzymy tę procedurę wpisywania kwadratów w kolejne mniejsze prostokąty i w każdy kwadrat wpiszemy ćwiartkę okręgu, otrzymamy złotą spiralę.

Złota proporcja w naturze

Zaskakująco wiele przykładów złotego podziału odnajdziemy w przyrodzie. Zbliżone do niego są m.in. proporcje w ciele człowieka. Badacze zauważyli też występowanie złotej proporcji na poziomie atomowym.

Huragan obserwowany z kosmosu ma kształt zbliżony do złotej spirali

Złota proporcja w projektowaniu

Również graficy i projektanci wykorzystują złotą proporcję. Odnajdziemy ją w logotypach wielu słynnych marek, takich jak Toyota czy Nissan. W złotej proporcji pozostają także wymiary kart płatniczych.

Na stronie wyszukiwarki Google złoty podział występuje zarówno w logotypie, jak i w układzie strony

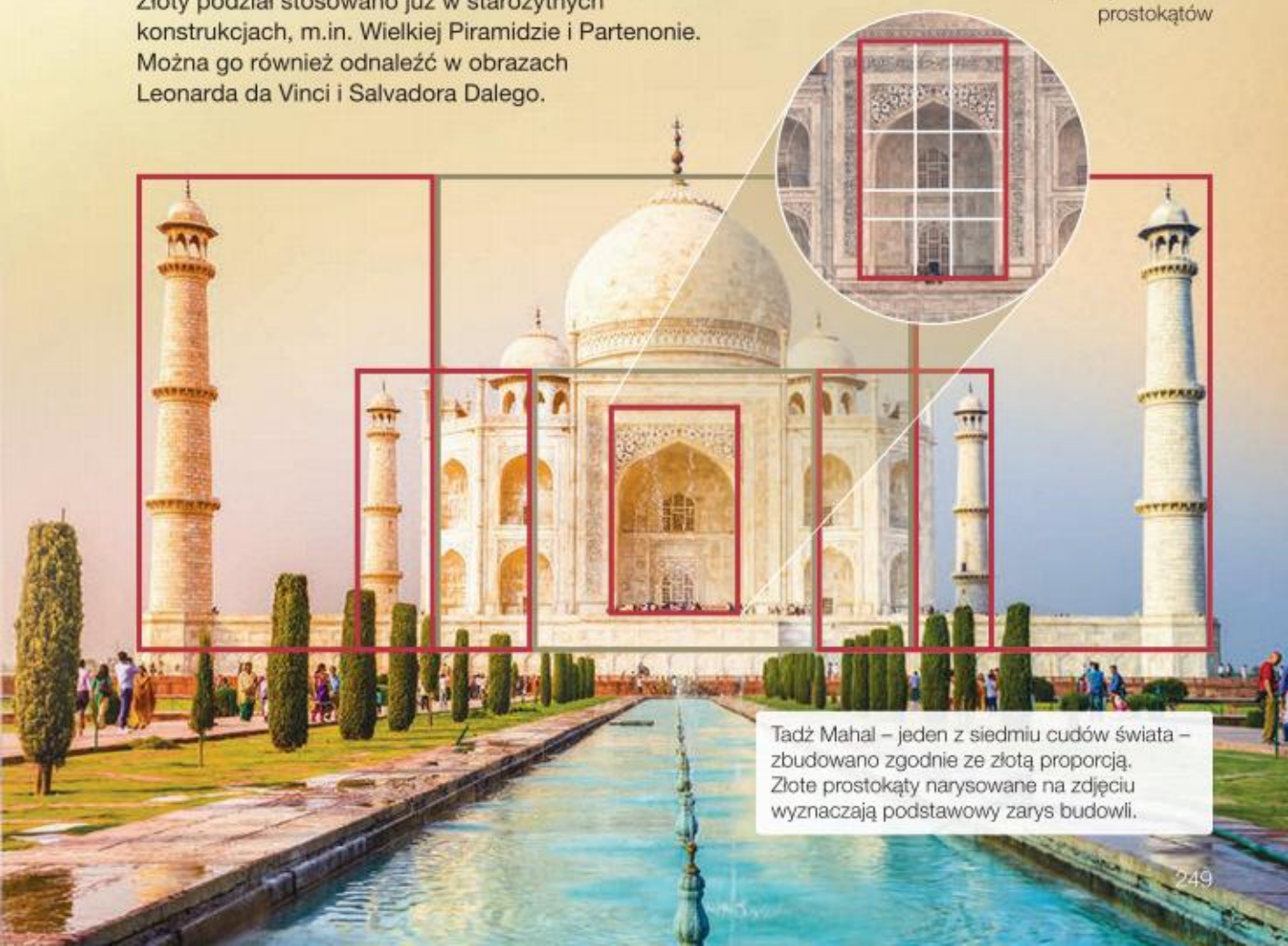
$$\frac{c}{a} \approx \frac{a}{b} \approx \varphi$$

Wewnętrzny prostokąt w logo National Geographic zachowuje złotą proporcję

Złota proporcja w architekturze i sztuce

Złoty podział stosowano już w starożytnych konstrukcjach, m.in. Wielkiej Piramidzie i Partenonie. Można go również odnaleźć w obrazach Leonarda da Vinci i Salvadora Dalego.

Okna i drzwi wyznaczają linie przecięcia złotych prostokątów



Tadż Mahal – jeden z siedmiu cudów świata – zbudowano zgodnie ze złotą proporcją. Złote prostokąty narysowane na zdjęciu wyznaczają podstawowy zarys budowli.

17.5. Rozszerzony algorytm Euklidesa

Algorytm Euklidesa, s. 102

Poprawność **algorytmu Euklidesa** wykazywaliśmy na podstawie zależności rekurencyjnej: $NWD(a, b) = NWD(a - b, b)$ przy założeniu, że $a > b$.

Obliczanie NWD można zatem wyrazić prostym wzorem rekurencyjnym, który ma zastosowanie zarówno w wersji algorytmu z odejmowaniem, jak i z dzieleniem.

$$NWD(a,b) = \begin{cases} a & \text{dla } a = b \\ NWD(a-b,b) & \text{dla } a > b \\ NWD(a,b-a) & \text{dla } b > a \end{cases}$$

$$NWD(a,b) = \begin{cases} a+b & \text{dla } a = 0 \text{ lub } b = 0 \\ NWD(a \bmod b, b) & \text{dla } a > b \\ NWD(a, b \bmod a) & \text{dla } b > a \end{cases}$$

Powyższy wzór można zapisać prościej:

$$NWD(a,b) = \begin{cases} a & \text{dla } b = 0 \\ NWD(b, a \bmod b) & \text{dla } b > 0 \end{cases}$$

Zauważ, że większa z dwóch liczb jest podczas obliczeń pamiętana w zmiennej a. Jeśli na początku a nie jest większą liczbą, pierwsze wywołanie rekurencyjne będzie polegało na zamianie wartości zmiennych a i b.

Warto wiedzieć

Jeśli $a < b$ i $b \neq 0$, to:
$NWD(a,b) =$
$= NWD(b, a \bmod b) =$
$= NWD(b,a)$

Funkcję rekurencyjną obliczającą największy wspólny dzielnik według powyższego wzoru można zapisać następująco:

```
funkcja NWDRek(a,b)
    jeśli b = 0 to zwróć a i zakończ
    zwróć NWDRek(b,a mod b) i zakończ
```

Ćwiczenie 10

Zapisz w języku C++ funkcję NWDRek i sprawdź jej działanie.

Rozszerzony algorytm Euklidesa

W niektórych zastosowaniach informatycznych, np. w kryptografii, wykorzystuje się **rozszerzony algorytm Euklidesa**, który największy wspólny dzielnik pozwala wyrazić wzorem:

$$NWD(a, b) = x \cdot a + y \cdot b$$

gdzie x i y są liczbami całkowitymi (mogą być także ujemne).

Poszukujemy zatem nie jednej liczby, ale pary liczb (x, y) spełniającej powyższe równanie dla danych liczb całkowitych a i b. Oznacza to, że algorytm nie tylko wyznacza NWD podanych liczb a i b, lecz także przedstawia ten dzielnik w postaci sumy liczb a i b pomnożonych przez współczynniki – odpowiednio x i y. Zapiszemy teraz specyfikację tak sformułowanego problemu.

Specyfikacja

Dane: a, b – liczby całkowite, $a > 0$, $b > 0$.

Wynik: x, y – liczby całkowite spełniające równanie:

$\mathrm{NWD}(a, b) = x \cdot a + y \cdot b$.

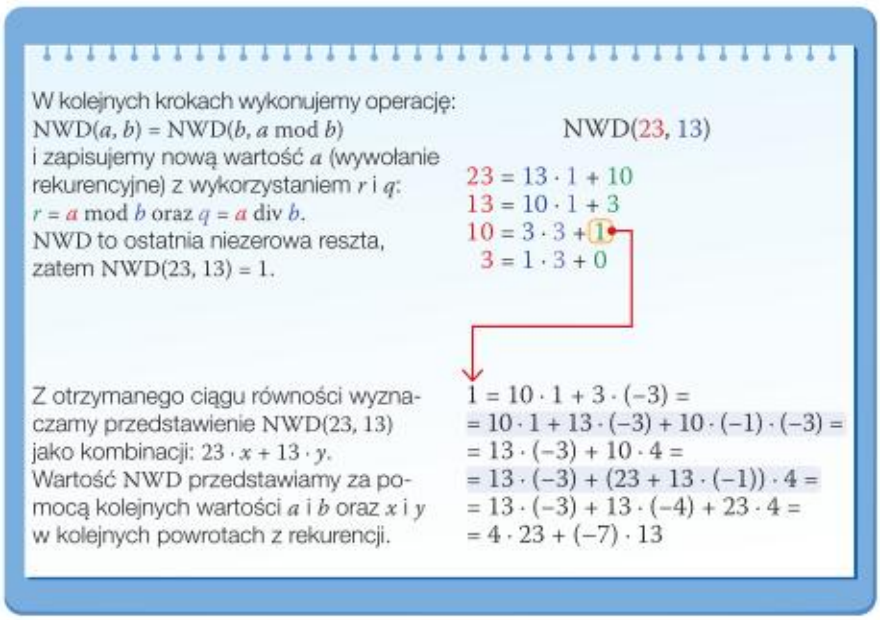
Algorytm wyznaczania współczynników x i y będzie polegał na modyfikacji funkcji `NWDRek` i wykorzystaniu prostych zależności arytmetycznych. Oznaczmy przez q wynik dzielenia całkowitego a przez b, a przez r – resztę z tego dzielenia:

$$q = a \text{ div } b$$
$$r = a \bmod b$$

Prześledźmy, jak się zmieniają wartości a, b, q i r (oznaczono je kolorami) podczas obliczania wartości NWD(23, 13) oraz jak wyznaczane są wartości x i y (rys. 17.8).

Warto wiedzieć

NWD(a,b) zawsze można przedstawić w postaci $x \cdot a + y \cdot b$. Nie jest to jedyne przedstawienie, ponieważ dla dowolnej liczby całkowitej k poprawne jest również przedstawienie: $(x - k \cdot b) \cdot a + (y + k \cdot a) \cdot b$.



Rys. 17.8. Schemat postępowania przy obliczaniu NWD(23, 13)

W górnej części schematu znajdują się kolejne wywołania rekurencyjne, a poniżej zostały rozpisane kolejne przedstawienia liczby 1 z wykorzystaniem aktualnych wartości a i b oraz x i y. W ostatnim kroku otrzymujemy przedstawienie dla wyjściowych wartości a i b:

$$\mathrm{NWD}(23, 13) = 4 \cdot 23 + (-7) \cdot 13$$

Zatem poszukiwaną parą liczb jest $(4, -7)$.

Zauważ, że w każdym kroku wartość x jest równa wartości y z poprzedniego kroku – oznaczmy ją przez y'. Wartość y jest równa $x' + (-q) \cdot y'$, gdzie x' to wartość x z poprzedniego kroku. Widać to na rysunku 17.8 w równościach zaznaczonych kolorem niebieskim.

Warto wiedzieć

Aby reprezentować dwie liczby całkowite, można zdefiniować własny typ danych. Może nim być struktura złożona z dwóch pól typu całkowitego. Oto definicja takiej struktury:

```
struct wspolczynniki
{
    int x,y;
};
```

Typ pair, s. 332

Aby zapisać algorytm rekurencyjny, za warunek początkowy przyjmiemy stan, w którym b będzie równe 0.

Możemy już zapisać w pseudokodzie funkcję NWDRozRek, obliczającą wartości x i y.

```
funkcja NWDRozRek(a,b)
    jeśli b = 0 to zwróć (1,0) i zakończ
    (x,y) ← NWDRozRek(b,a mod b)
    zwróć (y,x-(a div b) * y) i zakończ
```

Wartością funkcji jest para liczb całkowitych. Należy zatem użyć typu danych, który reprezentuje dwie liczby całkowite. Wygodnie będzie wykorzystać **typ** `pair` z biblioteki STL.

Oto kod źródłowy programu realizującego rozszerzony algorytm Euklidesa z wykorzystaniem funkcji NWDRozRek:

Kod źródłowy programu realizującego rozszerzony algorytm Euklidesa

```
1.  #include<iostream>
2.
3.  using namespace std;
4.
5.  pair<int,int> NWDRozRek(int a, int b)
6.  {
7.      if (b==0) return make_pair(1,0);
8.      pair<int,int> pom=NWDRozRek(b,a%b);
9.      return make_pair(pom.second,pom.first-(a/b)*pom.second);
10. }
11.
12. int main ()
13. {
14.     int a,b;
15.     cout<<"a="; cin>>a;
16.     cout<<"b="; cin>>b;
17.     pair<int,int> w=NWDRozRek(a,b);
18.     cout<<"x="<<w.first<<" y="<<w.second;
19.     return 0;
20. }
```

W liniach 5–10 zdefiniowana jest funkcja `NWDRozRek` typu `pair<int,int>`. W linii 7 tworzona jest para złożona z liczb 1 i 0, która jest wartością funkcji dla warunku początkowego. Zmienna pomocnicza `pom`, deklarowana jako para (linia 8), przechowuje wynik rekurencyjnego wywołania funkcji NWDRozRek. W linii 9 tworzona jest para zgodnie z zasadą tworzenia nowych wartości zmiennych `x` i `y`. Wykorzystujemy do tego celu **funkcję** `make_pair`. W linii 17 deklarowana jest zmienna `w` typu `pair<int,int>`, która przechowuje wynik działania funkcji NWDRozRek dla wczytanych danych. W linii 18 wypisywane są elementy pary `w`.

Funkcja make_pair, s. 332

Podsumowanie

- Rekurencja to odwołanie się w rozwiązaniu problemu do tego samego problemu, ale dla mniejszego rozmiaru danych.
- Algorytm rekurencyjny powinien mieć sformułowany warunek końca, nazywany warunkiem początkowym, dla którego nie ma już wywołania rekurencyjnego.
- Bezpośrednie przełożenie rekurencyjnych wzorów matematycznych na funkcje często prowadzi do bardzo nieefektywnych algorytmów.
- Korzystanie z rekurencji znacząco obciąża pamięć komputera.
- Jeśli dany problem można rozwiązać zarówno iteracyjnie, jak i rekurencyjnie, lepiej zrobić to iteracyjnie.
- W rekurencji ogonowej występuje dokładnie jedno wywołanie rekurencyjne – jest to ostatnia instrukcja. Można ją zastąpić algorytmem iteracyjnym z zastosowaniem pojedynczej pętli.
- Każdy algorytm rekurencyjny można sprowadzić do iteracji. Często jest to jednak skomplikowane i wymaga użycia dodatkowych struktur danych.

Zadania

* **1** Dany jest ciąg a_n, określony następującym wzorem rekurencyjnym:

$$a_n = \begin{cases} 1 & \text{dla } n = 0 \\ 2 \cdot a_{n-1} & \text{dla } n > 0 \end{cases}$$

a. Przygotuj arkusz kalkulacyjny obliczający kolejne wyrazy ciągu.

b. Jak nazywamy ciąg, w którym kolejny wyraz powstaje przez pomnożenie poprzedniego przez określoną wartość? Jakie wartości reprezentują wyrazy powyższego ciągu?

c. Napisz w pseudokodzie dwie funkcje, które będą obliczać n-ty wyraz ciągu – jedna w sposób rekurencyjny, druga – w iteracyjny.

* **2** Napisz program obliczający największy wspólny dzielnik n liczb całkowitych dodatnich. Problem rozwiąż iteracyjnie.
Uwaga: Możesz skorzystać z poniższej zależności rekurencyjnej.
$\text{NWD}(a_1, a_2, a_3, \ldots, a_n) = \text{NWD}(a_1, \text{NWD}(a_2, a_3, \ldots, a_n))$

* **3** W pliku otrzymanym od nauczyciela (np. *ciag_rekurencyjny.xlsx*) znajduje się ciąg liczbowy wraz z formułą go tworzącą. Na tej podstawie zapisz pseudokod funkcji rekurencyjnej.

* **4** Zapisz w pseudokodzie funkcję wyznaczającą n-ty wyraz ciągu, którego początkowe wyrazy mają wartości: $a_1 = -2$, $a_2 = 4$, $a_3 = -8$, $a_4 = 16$, $a_5 = -32$ i w którym wartość każdego następnego wyrazu zależy od wyrazu poprzedniego.

** **5** Dany jest ciąg f_n, określony wzorem rekurencyjnym:

$$f_n = \begin{cases} -1 & \text{dla } n = 1 \text{ lub } n = 2 \\ f_{n-1} - f_{n-2} & \text{dla } n > 2 \end{cases}$$

a. Jaki ciąg opisuje powyższy wzór?
b. Podaj wzór ogólny powyższego ciągu oraz napisz funkcję obliczającą n-ty wyraz ciągu.

** **6** Dane są: trzy liczby całkowite p, k, n spełniające warunek $p \leqslant k < n$, tablica uporządkowanych liczb całkowitych `A[0..n-1]`, liczba całkowita x oraz funkcja rekurencyjna `F`:

```
funkcja F(A[],p,k,x)
    jeśli p = k to zwróć A[p] = x i zakończ
    s ← (p + k) div 2
    jeśli A[s] < x to zwróć F(A[],s+1,k,x) i zakończ
    zwróć F(A[],p,s,x) i zakończ
```

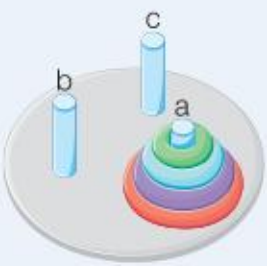
a. Jaki jest wynik działania powyższej funkcji?
b. Oszacuj liczbę porównań wykonywanych przez funkcję.
c. Napisz program, który wylosuje, posortuje i wypisze tablicę `A[0..n-1]` liczb całkowitych, następnie wczyta do zmiennej `x` dowolną liczbę całkowitą i wypisze wynik powyższej funkcji wywołanej z parametrami `F(A,0,n-1,x)`.

** **7** W pliku, który otrzymasz od nauczyciela (np. *Hanoi.pdf*), znajduje się opis rekurencyjnego rozwiązania problemu wież Hanoi. Napisz w pseudokodzie funkcję rekurencyjną rozwiązującą ten problem. Polecenie `Przenies(i,j)`, gdzie `i` oraz `j` oznaczają wieże, potraktuj jako elementarne.

** **8** W pliku, który otrzymasz od nauczyciela (np. *Hanoi.pdf*), znajduje się opis rekurencyjnego rozwiązania problemu wież Hanoi. Napisz funkcję zliczającą operacje przenoszenia pojedynczego krążka w problemie wież Hanoi dla n krążków. Zapisz rozwiązanie rekurencyjnie, a następnie iteracyjnie.

*** **9** Zapisz iteracyjnie rozszerzony algorytm Euklidesa.

*** **10** Korzystając z pliku, który udostępni ci nauczyciel (np. *Hanoi.pdf*), rozwiąż problem wież Hanoi w sposób iteracyjny.
Uwaga: Wieże ustaw na okręgu i przekładaj krążek o najmniejszej średnicy zgodnie z ruchem wskazówek zegara lub przeciwnie do tego ruchu w zależności od tego, czy liczba krążków jest parzysta czy nieparzysta.



18. Metoda zachłanna

Prawdopodobnie nie raz zdarzyło ci się korzystać z różnych automatów z napojami i słodyczami lub biletomatów. Zazwyczaj można w nich płacić kartą albo gotówką. Przy płatności gotówką wiele automatów ma możliwość wydania reszty. Czy wiesz, według jakiego algorytmu działają? Czy w ten sam sposób postępuje sprzedawca, gdy wydaje resztę w sklepie? Odpowiedzi na te pytania znajdziesz w tym temacie. Poznasz przy tym nową metodę algorytmiczną nazywaną metodą zachłanną.

Cele lekcji

- Poznasz metodę zachłanną, stosowaną do rozwiązywania problemów algorytmicznych.
- Opracujesz algorytm wydawania reszty najmniejszą liczbą monet i banknotów.
- Przygotujesz optymalny harmonogram wykorzystania sali.
- Poszukasz drogi o największej sumie pól na kwadratowej planszy wypełnionej liczbami.
- Przeanalizujesz problem pakowania plecaka.
- Wykorzystasz funkcję `sort` z biblioteki STL i napiszesz własną funkcję porównującą elementy.

18.1. Wydawanie reszty metodą zachłanną

Wyobraź sobie, że podczas zakupów kasjer wydaje ci kwotę 7,25 zł wyłącznie monetami jednogroszowymi. Z perspektywy klienta byłoby to bardzo uciążliwe, a z perspektywy sklepu nieopłacalne – wkrótce kasjerowi zabrakłoby monet jednogroszowych do wydawania kwot wymagających zwrócenia jednego grosza.

Warto więc zastanowić się nad algorytmem minimalizującym liczbę wydawanych banknotów i monet. Zastosujemy w nim **metodę zachłanną**, która polega na realizowaniu przez algorytm najlepszego wyboru w danym kroku rozwiązywania problemu. W algorytmie zachłannym nie ocenia się, jak decyzja podjęta w danej chwili wpłynie na kolejne kroki i całe rozwiązanie. Kontynuuje się rozwiązywanie problemu zgodnie z wyborem dokonanym w danym kroku – oczekuje się, że rozwiązanie globalne też będzie najlepsze.

Metoda zachłanna

Zauważ, że na początku wydawania reszty można wybrać największy nominał, który jest mniejszy od kwoty reszty lub równy tej kwocie, i wydać go tyle razy, ile to możliwe, aby nie przekroczyć reszty. Następnie można powtarzać ten krok, pamiętając, aby zmniejszać za każdym razem kwotę reszty do wydania o łączną wartość już wybranych nominałów.

Algorytm wydawania reszty

Kasjerzy i automaty przyjmujące gotówkę korzystają z prostego sposobu wydawania reszty najmniejszą liczbą banknotów lub monet. Jest to przykład podejścia zachłannego w algorytmice.

Jak wydać resztę najmniejszą liczbą banknotów lub monet?

1. Szukamy największego dostępnego nominału nie większego od reszty.
2. Wydajemy nominał tyle razy, ile można, aby nie przekroczyć kwoty reszty.
3. Pomniejszamy resztę do wydania o łączną wartość już wybranych nominałów, a następnie powtarzamy kroki 1 i 2.

Jak zwrócić resztę wynoszącą 142,43 zł?

Kwota do wydania	Nominał	Liczba banknotów lub monet
142,43 zł	100 zł	1
42,43 zł	20 zł	2
2,43 zł	2 zł	1
0,43 zł	0,20 zł	2
0,03 zł	0,02 zł	1
0,01 zł	0,01 zł	1

Kolejne kwoty są wyznaczane przez resztę z dzielenia:

```
kwota = kwota mod nominał
```

Kolejne liczby banknotów lub monet są wyznaczane przez dzielenie całkowite:

```
liczba = kwota div nominał
```

Polskie nominały

W programie realizującym ten algorytm wszystkie nominały można wyrazić w groszach i przechować jako liczby całkowite w tablicy o stałych wartościach, np.:

```
Nominały[15] = {50000, 20000,
10000, 5000, 2000, 1000, 500,
200, 100, 50, 20, 10, 5, 2, 1}
```

Oto specyfikacja problemu wydawania reszty oraz zapis algorytmu:

Specyfikacja

Dane: `kwota` – reszta do wydania wyrażona w groszach, liczba całkowita dodatnia, `n` – liczba dostępnych nominałów, `Nominaly[0..n-1]` – tablica dostępnych nominałów, wyrażonych w groszach, uporządkowana malejąco.

Wynik: `Reszta[0..n-1]` – tablica z liczbą poszczególnych banknotów i monet o łącznej wartości `kwota`. Kolejność nominałów w tablicy `Reszta` jest zgodna z kolejnością nominałów w tablicy `Nominaly`.

Dobra rada

Zawsze dokładnie czytaj specyfikację problemu. Często narzuca ona typ danych konieczny do zastosowania w programie.

```
dla i ← 0, 1, …, n - 1 wykonuj
    Reszta[i] ← kwota div Nominaly[i]
    kwota ← kwota mod Nominaly[i]
```

W pętli wyliczamy liczbę banknotów lub monet o wartości `Nominaly[i]`, korzystając z operacji dzielenia całkowitego. Kwota pozostała do wydania to reszta z dzielenia przez dany nominał.

Ćwiczenie 1

W pliku otrzymanym od nauczyciela (np. *reszta.xlsx*) zrealizuj algorytm wydawania reszty najmniejszą liczbą banknotów lub monet.

Oto fragment kodu źródłowego programu realizującego algorytm:

```
1.  const int N=15;
2.  const int NOMINALY[]={50000,20000,10000,5000,2000,1000,
3.                        500,200,100,50,20,10,5,2,1};
4.  void WydajReszte(int kwota, int Reszta[])
5.  {
6.      for (int i=0;i<N;i++)
7.      {
8.          Reszta[i]=kwota/NOMINALY[i];
9.          kwota=kwota%NOMINALY[i];
10.     }
11. }
12. int main()
13. {
14.     int kwota, Reszta[N];
15.     cout<<"Podaj kwote w groszach: "; cin>>kwota;
16.     WydajReszte(kwota,Reszta);
17.     for (int i=0;i<N;i++)
18.         if (Reszta[i]>0)
19.             cout<<NOMINALY[i]<<": "<<Reszta[i]<<endl;
20.     return 0;
21. }
```

Fragment kodu źródłowego programu wyznaczającego liczbę poszczególnych banknotów lub monet potrzebnych do wydania reszty

W linii 1 została zadeklarowana stała `N`, określająca liczbę dostępnych nominałów, a w liniach 2–3 **stała tablicowa** o nazwie `NOMINALY`, czyli tablica wypełniona stałymi (niezmiennymi) wartościami nominałów. **Tablicom** można nadawać wartości początkowe podczas ich deklaracji – wpisujemy je w nawiasach klamrowych, oddzielając przecinkami. Rozmiar tablicy wynika z liczby elementów w nawiasach klamrowych, więc nie trzeba go podawać.

Stała tablicowa

Tablice, s. 330

Zwróć uwagę na nagłówek funkcji w wierszu 4. Tablice jako parametry są **przekazywane przez wskaźnik**, więc zmiany wprowadzone w tablicy `Reszta` zostaną zachowane. Pętla w wierszach 17–19 powoduje wypisanie liczby tych nominałów, które są wykorzystywane do wydania reszty. Rysunek 18.1 przedstawia efekt działania programu.

Przekazanie parametru przez wskaźnik, s. 152

```
Podaj kwote w groszach: 13579
10000: 1
2000: 1
1000: 1
500: 1
50: 1
20: 1
5: 1
2: 2
```

Rys. 18.1. Przykładowe wywołanie programu wyznaczającego resztę wydawaną najmniejszą liczbą banknotów lub monet

Warto wiedzieć

Rozmiar tablicy można także podać podczas określania wartości początkowych. Jeśli podamy mniejszy rozmiar niż liczba elementów w nawiasach klamrowych, będzie to błąd. W przypadku podania większego rozmiaru pozostałe elementy będą miały wartość 0.

Ćwiczenie 2

a. Uruchom program ze s. 257 i sprawdź jego działanie.
b. Popraw program tak, aby wypisywał wartości w złotówkach i groszach.

Warto wiedzieć

Optymalizacja rozwiązania danego problemu może oznaczać coś innego dla różnych problemów. W przypadku wydawania reszty oznacza ona wydanie możliwie najmniejszej liczby banknotów lub monet.

Można podać przykłady hipotetycznych systemów monetarnych, dla których metoda zachłanna nie znajdzie optymalnej (najmniejszej) liczby banknotów i monet. Załóżmy, że w polskim systemie monetarnym monetę 2 zł zastępujemy monetą 4 zł i chcemy wydać 8 zł reszty. Metoda zachłanna wyznaczy wówczas jako resztę jedną monetę 5 zł i trzy monety 1 zł, czyli razem cztery monety. Kwotę 8 zł można by jednak wydać za pomocą dwóch monet 4 zł, czyli w tej sytuacji dla podanych nominałów istnieje lepsze rozwiązanie.

A to ciekawe

Systemy monetarne a wydawanie reszty

Systemy monetarne wielu krajów są tak skonstruowane, aby metoda zachłanna działała w nich optymalnie, tzn. żeby można było wydać resztę najmniejszą liczbą banknotów i monet. Polski system monetarny jest zbudowany na bazie liczb: 5, 2, 1. Wartości banknotów i monet są wielokrotnościami tych liczb.

Choć omówiony algorytm rozwiązuje przedstawiony w specyfikacji problem, nie zawsze wskazuje najmniejszą liczbę monet, którymi można wypłacić resztę. Takie rozwiązania z wykorzystaniem metody zachłannej, o ile są łatwe w implementacji, mogą wystarczać, gdy można zaakceptować przybliżone rozwiązanie danego problemu.

Ćwiczenie 3

Podaj wymyślony przez siebie przykład systemu monetarnego, w którym wydawanie reszty metodą zachłanną nie jest optymalne.

Zapamiętaj

Metoda zachłanna polega na podejmowaniu decyzji optymalnej w danym kroku. Nie rozważa się w tym momencie, jak wybór wpłynie na kolejne kroki. Dlatego podejście zachłanne nie zawsze prowadzi do znalezienia optymalnego rozwiązania. Można je jednak zastosować, gdy wystarczy znaleźć rozwiązanie przybliżone.

18.2. Optymalne wykorzystanie sali

Rozważmy następujący problem: w szkole jest organizowany festiwal teatralny, na który każda z klas przygotowuje przedstawienie. W sali gimnastycznej została ustawiona scena na próby. Czas trwania przedstawień jest różny, klasy rozpoczynają też próby o różnych godzinach. Zadanie polega na ustaleniu harmonogramu, według którego w sali gimnastycznej odbędzie się możliwie jak najwięcej prób przedstawień. Zakładamy, że dwie próby nie mogą się odbywać w sali jednocześnie, a klasy, które nie znajdą się na liście, ćwiczą w innej sali. Tabela 18.1 przedstawia przykładowe dane.

Warto wiedzieć

Optymalizacja oznacza dążenie do znalezienia rozwiązania, które zgodnie z ustalonym kryterium jest oceniane jako najlepsze z dopuszczalnych rozwiązań.

Klasa	Godzina rozpoczęcia	Czas trwania (min)
1a	9	150
1b	10	50
2a	11	60
2b	12	50
3a	13	60
3b	14	80
4a	15	90
4b	16	100

Tabela 18.1. Przykładowe zestawienie godzin rozpoczęcia i czasu trwania prób

Warto wiedzieć

Podczas rozwiązywania problemu wykorzystania sali można wskazać maksymalną liczbę prób, które mogą się w tej sali odbyć, ale dla pewnych układów danych sam wybór klas może być różny. Na przykład jeśli dwie klasy miałyby tej samej długości próbę o tej samej porze, to można byłoby wybrać pierwszą lub drugą z nich.

Maksymalna liczba prób w sali gimnastycznej dla danych zamieszczonych w tabeli 18.1 na s. 259 wynosi 6 (klasy: 1b, 2a, 2b, 3a, 3b i 4b). Zauważ, że gdyby jako pierwsza zaczęła próbę klasa 1a, wówczas próby mogłoby odbyć tylko 5 klas (1a, 2b, 3a, 3b, 4b). W tej sytuacji uwzględnienie w harmonogramie klasy, która najwcześniej rozpoczyna próbę, nie jest dobrym wyborem.

Metoda zachłanna w tym przypadku polega na wyborze w danym kroku próby, która najwcześniej się kończy i nie koliduje z już wybranymi próbami (tymi, które zakończyły się wcześniej). Strategia ta prowadzi to wyboru optymalnego rozwiązania, czyli ustalenia maksymalnej liczby prób, które mogą się odbyć w sali gimnastycznej.

Oto specyfikacja problemu:

Specyfikacja

Dane: `n` – liczba klas, `P[0..n-1]` – tablica prób, próba reprezentowana jest przez dwie liczby całkowite dodatnie `pocz` i `czas`, określające godzinę rozpoczęcia próby i czas jej trwania w minutach.

Wynik: `lp` – maksymalna liczba prób niekolidujących ze sobą.

Aby określić maksymalną liczbę prób, które mogą się odbyć w sali gimnastycznej, wygodnie będzie uporządkować przedstawienia według czasu ich zakończenia.

Dobra rada

Próbę możesz reprezentować z wykorzystaniem tablicy dwuwymiarowej. W pierwszej kolumnie mogą być pamiętane godziny rozpoczęcia prób, a w drugiej – czas ich trwania.

Algorytm znajdujący maksymalną liczbę prób, przy założeniu uporządkowania przedstawień według czasu ich zakończenia, można zapisać w pseudokodzie następująco:

```
lp ← 1  // liczba prób wybranych do sali
ost ← 0 // indeks w tablicy ostatniej wybranej próby
dla i ← 1, 2, ..., n - 1 wykonuj
    jeśli P[i].pocz * 60 ≥ P[ost].pocz * 60 + P[ost].czas to
        lp ← lp + 1
        ost ← i
```

Zauważ, że godzina rozpoczęcia próby jest mnożona przez 60, aby otrzymana wartość była wyrażona w minutach (czas trwania próby podajemy w minutach). Do reprezentacji próby (godziny rozpoczęcia i czasu trwania) wykorzystamy **strukturę**. Dzięki temu będziemy mogli skorzystać z **funkcji** `sort` z biblioteki STL w celu uporządkowania tablicy struktur zawierającej informacje o próbach.

Struktura, s. 109

Funkcja `sort`, s. 214

Oto definicja struktury proba:

```
struct proba
{
    int pocz;
    int czas;
};
```

Aby funkcja **sort** mogła porównać dwie próby, należy zdefiniować funkcję logiczną określającą, która z dwóch prób kończy się wcześniej. Oto kod źródłowy struktury proba i funkcji logicznej Porownaj:

Kod źródłowy struktury proba i funkcji logicznej Porownaj

```
1.  struct proba
2.  {
3.      int pocz, czas;
4.  };
5.
6.  bool Porownaj(proba a, proba b)
7.  {
8.      return (a.pocz*60+a.czas<b.pocz*60+b.czas);
9.  }
```

Dobra rada

Dane dotyczące czasu sprowadzaj do jednolitej postaci, np. zamieniając na wartości minutowe.

Dane odczytamy z **pliku tekstowego**. Przykład takich danych przedstawia rysunek 18.2. W każdym wierszu znajdują się dwie liczby oddzielone spacją, które określają godzinę rozpoczęcia i czas trwania kolejnych prób (tablica P). W pliku znajduje się tyle wierszy, ile wynosi wartość stałej N, określającej rozmiar tablicy P w programie. Oto kod źródłowy funkcji MLP (maksymalna liczba prób) oraz funkcji main:

```
9 150
10 50
11 60
12 50
13 60
```

Rys. 18.2. Fragment zestawu danych z pliku *dane.txt*

Obsługa plików tekstowych, s. 336–338

Fragment kodu źródłowego programu wyznaczającego maksymalną liczbę prób

```
1.  int MLP(proba P[])
2.  // Maksymalna Liczba Prób
3.  {
4.      int lp=1, ost=0;
5.      for (int i=1;i<N;i++)
6.          if (P[i].pocz*60>=P[ost].pocz*60+P[ost].czas)
7.          {
8.              lp++;
9.              ost=i;
10.         }
11.     return lp;
12. }
13.
14. int main()
15. {
16.     proba P[N];
17.     ifstream we("dane.txt");
18.     for (int i=0;i<N;i++)
19.         we>>P[i].pocz>>P[i].czas;
20.     we.close();
21.     sort(P,P+N,Porownaj);
22.     cout<<"Maksymalna liczba prob: "<<MLP(P);
23.     return 0;
24. }
```

Zmienna plikowa, s. 122

Dane liczbowe z pliku tekstowego wczytujemy tak samo jak z klawiatury. Zamiast standardowego wejścia `cin` należy użyć nazwy **zmiennej plikowej** powiązanej z danym plikiem. W linii 16 deklarowana jest *N*-elementowa tablica typu `proba`. Jeden element tablicy stanowi strukturę pamiętającą dwie liczby opisujące próbę. W wierszu 17 deklarowana jest zmienna `we` typu `ifstream`, a z nią skojarzony jest plik *dane.txt*, który jest otwarty w trybie do odczytu. Pętla w wierszach 18–19 odczytuje godziny rozpoczęcia i czas trwania prób. W następnym wierszu plik jest zamykany, ponieważ nie będzie już potrzebny. Aby skorzystać z obsługi plików, należy dołączyć bibliotekę `fstream`.

Dobra rada

Pamiętaj, żeby plik z danymi umieścić w tym samym katalogu (folderze), w którym znajduje się plik z programem.

W wierszu 21 wywołana jest funkcja sortująca `sort`. Jej trzeci parametr (funkcja `Porownaj`) określa sposób porównania dwóch elementów. W linii 22 wywoływana jest funkcja `MLP`, a na ekranie wypisywana jest jej wartość, czyli maksymalna liczba prób. Aby skorzystać z funkcji `sort`, należy dołączyć bibliotekę `algorithm`.

Ćwiczenie 4

Uruchom program i sprawdź jego działanie dla pliku:

a. przekazanego przez nauczyciela (np. *proby.txt*),

b. zawierającego przygotowane przez ciebie dane.

18.3. Maksymalna suma w kwadracie – metoda zachłanna

Szukamy podciągu spójnego o maksymalnej sumie, s. 207–210

Zajmowaliśmy się już problemem znalezienia **podciągu o maksymalnej sumie**. Teraz zapiszemy algorytm, który będzie poszukiwał maksymalnej sumy na kwadratowej planszy z liczbami.

Zadanie polega na przejściu od lewego górnego rogu planszy do prawego dolnego tak, aby suma liczb znajdujących się w poszczególnych polach była jak największa. Jedynymi dozwolonymi ruchami są przejścia na sąsiednie pola w prawo lub w dół. Liczby w polach, w omawianym przez nas przykładzie, będą tylko dodatnie.

Rysunek 18.3 przedstawia przykładową planszę z zaznaczonym na zielono podciągiem o maksymalnej sumie, która wynosi 60.

	0	1	2	3	4
0	5	5	1	3	9
1	8	6	4	9	8
2	5	1	3	9	4
3	5	6	2	8	3
4	5	6	3	6	5

Rys. 18.3. Przykładowa plansza z zaznaczonym podciągiem o maksymalnej sumie

Oto specyfikacja problemu:

Specyfikacja

Dane: n – liczba wierszy i kolumn planszy,
P[0..n–1][0..n–1] – tablica liczb całkowitych dodatnich.

Wynik: suma – maksymalna suma liczb w polach od P[0][0] do P[n–1][n–1], dozwolone ruchy to przejście na sąsiednie pole w prawo lub w dół.

Zastosowanie strategii zachłannej w tym przypadku polega na wyborze najlepszego sąsiedniego pola, czyli pola z większą liczbą. Jeśli obydwa pola mają taką samą wartość, nie ma znaczenia, które z nich zostanie wybrane.

Na poszukiwaną ścieżkę składa się dokładnie $2n - 1$ pól. Po dotarciu do dolnego wiersza można się już poruszać tylko w prawo. Analogicznie po osiągnięciu ostatniej kolumny można już wykonywać tylko ruchy w dół. Aby nie sprawdzać, czy indeks nie przekracza zakresu tablicy, można dodać do niej **wartowników**, jak na rysunku 18.4.

Warto wiedzieć

Algorytmy działające według strategii zachłannej nie biorą pod uwagę kolejnych kroków, wybierają najlepszy ruch tylko w danym kroku.

Wartownik, s. 64

	0	1	2	3	4	5
0	5	5	1	3	9	0
1	8	6	4	9	8	0
2	5	1	3	9	4	0
3	5	6	2	8	3	0
4	5	6	3	6	5	0
5	0	0	0	0	0	0

Rys. 18.4. Plansza z zaznaczonymi wartownikami

Oto zapis algorytmu zachłannego w pseudokodzie:

```
suma ← A[0][0]
w ← 0 // wiersz, w którym się znajdujemy
k ← 0 // kolumna, w której się znajdujemy
dla i ← 1, 2, …, 2 * n – 2 wykonuj
    jeśli A[w+1][k] > A[w][k+1] to
        suma ← suma + A[w+1][k]
        w ← w + 1
    w przeciwnym przypadku
        suma ← suma + A[w][k+1]
        k ← k + 1
```

Dobra rada

Algorytm poszukiwania maksymalnej sumy w kwadracie możesz również zapisać za pomocą następującej pętli:

```
dopóki (w < n – 1 lub
        k < n – 1)
wykonuj
```

Wartością początkową zmiennej suma jest liczba pamiętana w polu startowym. Zmienne w i k przechowują współrzędne pola, w którym się aktualnie znajdujemy (ostatnio wybrane pole). W pętli wybierane jest sąsiednie pole z większą liczbą oraz modyfikowane są wartości zmiennych suma oraz w lub k.

Poniżej znajduje się kod źródłowy funkcji Losuj, losującej planszę (wraz z ustawieniem wartowników), oraz funkcji MaksSuma, realizującej omówiony algorytm.

Kod źródłowy funkcji Losuj, **generującej dane w tablicy, funkcji** Wypisz, **wypisującej liczby z planszy, oraz funkcji** MaksSuma, **która wyznacza maksymalną sumę liczb**

```
1.  const int N=5;
2.  void Losuj(int A[][N+1])
3.  {
4.      for (int i=0;i<N;i++)
5.          for (int j=0;j<N;j++) A[i][j]=1+rand()%9;
6.      for (int i=0;i<N;i++)
7.      {
8.          A[N][i]=0; A[i][N]=0;
9.      }
10. }
11. void Wypisz(int A[][N+1])
12. {
13.     for (int i=0;i<N;i++)
14.     {
15.         for (int j=0;j<N;j++) cout<<A[i][j]<<" ";
16.         cout<<endl;
17.     }
18. }
19. int MaksSuma(int A[][N+1])
20. {
21.     int suma=A[0][0], w=0, k=0;
22.     for (int i=1;i<2*N-1;i++)
23.         if (A[w+1][k]>A[w][k+1])
24.         {
25.             suma+=A[w+1][k]; w++;
26.         }
27.         else
28.         {
29.             suma+=A[w][k+1]; k++;
30.         }
31.     return suma;
32. }
```

Dobra rada

Jeśli chcesz wylosować liczbę c z przedziału [a,b], możesz zastosować zapis: `c = a + rand()%(b-a+1)`

Parametrem każdej z funkcji Losuj, Wypisz i MaksSuma jest **tablica dwuwymiarowa**. W języku C++ konieczne jest podanie liczby elementów drugiego wymiaru (i kolejnych wymiarów w przypadku tablic wielowymiarowych). W linii 23 sprawdzane jest, czy przejście w dół planszy daje większą sumę częściową niż przejście w prawo. Przykładowe wywołanie programu i wyznaczenie sumy dla wylosowanej tablicy liczb przedstawia rysunek 18.5.

Tablica dwuwymiarowa, s. 219

```
3 2 2 8 9
4 7 1 5 1
2 2 7 6 2
8 5 1 5 1
7 9 2 8 4
Suma: 46
```

Rys. 18.5. Przykładowe wywołanie programu

Ćwiczenie 5

Napisz program poszukujący maksymalnej sumy pól w kwadracie. Wykorzystaj funkcje `Losuj` oraz `MaksSuma`. Czy za każdym razem otrzymujesz maksymalną sumę?

Zwróć uwagę, że metoda zachłanna nie zawsze znajduje maksymalną sumę pól. W przykładzie danych z tego zagadnienia (rys. 18.4, s. 263) wystarczy w polu (2,3) zamienić 9 np. na 7, aby algorytm zachłanny nie wyznaczył maksymalnej sumy.

Program działa więc zgodnie ze specyfikacją (celem jest znalezienie maksymalnej sumy) tylko dla pewnych zestawów liczb. Dla pozostałych zestawów znajduje jedynie rozwiązanie przybliżone, a więc nie realizuje specyfikacji. Podobnie jak w przypadku **problemu wydawania reszty** program zawsze znajduje rozwiązanie, ale nie zawsze jest ono optymalne.

Wydawanie reszty metodą zachłanną, s. 255–259

Ćwiczenie 6

a. Wyznacz maksymalną sumę w kwadracie z rysunku 18.4 na s. 263 po zmianie wartości w polu (2,3) z 9 na 5.

b. Zastosuj metodę zachłanną do wyznaczenia maksymalnej sumy w kwadracie z rysunku 18.4 na s. 263 po zmianie wartości w polu (2,3) z 9 na 5.

18.4. Pakowanie plecaka metodą zachłanną

Pakowanie plecaka to dla wielu z nas duże wyzwanie. Na przykład do samolotu często można zabrać tylko jedną sztukę bagażu i to o ograniczonej wadze. Na wycieczkę górską pakuje się plecak tak, aby nie ważył on zbyt wiele. Spośród przedmiotów, które powinno się zabrać, trzeba wybrać te najbardziej potrzebne tak, aby maksymalnie wykorzystać dopuszczalną wagę.

Takie zagadnienie w ujęciu informatycznym nazywamy **problemem plecakowym**. Można go sformułować w następujący sposób: mamy n rodzajów przedmiotów, każdy o określonej wadze i wartości (informującej o jego przydatności). Chcemy wybrać te przedmioty, które w sumie będą miały największą wartość, ale łącznie ich waga nie przekroczy wyznaczonego limitu. Zakładamy przy tym, że dysponujemy nieograniczoną liczbą przedmiotów każdego rodzaju oraz że są one niepodzielne, to znaczy nie możemy zapakować do plecaka fragmentu żadnego przedmiotu. Tak sformułowany problem nazywamy **ogólnym problemem plecakowym**.

Problem plecakowy

Ogólny problem plecakowy

Jeśli mamy do dyspozycji tylko jeden przedmiot każdego rodzaju (pakowanie ogranicza się do decyzji, czy dany przedmiot spakować czy nie), to tak sformułowany problem nazywamy **decyzyjnym problemem plecakowym**.

Decyzyjny problem plecakowy

Pakowanie plecaka

Aby możliwie najlepiej zapakować plecak tak, aby ważył maksymalnie 7 kg, każdemu przedmiotowi przypisujemy jego wagę oraz wartość, która określa, jak bardzo jest przydatny.

Jak wybrać przedmioty do spakowania?

Przedmiot	Waga [g]	Wartość	Iloraz	Łączna waga przedmiotów
Kompas	20	30	1,5	20
Telefon	200	100	0,5	220
Rękawiczki	100	30	0,3	320
Mapa	200	54	0,27	520
Powerbank	400	85	0,21	920
Obiektyw	500	100	0,2	1420
Butelka z wodą	1000	150	0,15	2420
Notes	200	29	0,145	2620
Aparat	900	100	0,11	3520
Laptop	1800	200	0,11	5320
Portfel	150	16	0,107	5470
Książka	200	20	0,1	5670
Zasilacz	800	68	0,085	6470
Linka	500	34	0,068	6970
Sweter	650	40	0,062	7620
Rondel	350	15	0,043	7970
Okulary	50	2	0,04	8020

Obliczamy stosunek wartości do wagi. Im wyższy iloraz, tym korzystniej jest spakować dany przedmiot

Przedmioty porządkujemy malejąco według ilorazów i wyliczamy narastająco łączną wagę przedmiotów. Na zielono zaznaczono najistotniejsze przedmioty, których łączna waga nie przekracza 7 kg

W dalszej części tematu zajmiemy się ogólnym problemem plecakowym z nieograniczoną liczbą przedmiotów do zapakowania.

Specyfikacja

Dane: `n` – liczba rodzajów przedmiotów, `P[0..n-1]` – tablica przedmiotów, przedmiot reprezentowany jest przez dwie liczby całkowite dodatnie `waga` i `wart`, określające jego wagę i wartość, `makswaga` – maksymalna łączna waga przedmiotów, które można spakować. Dysponujemy nieograniczoną liczbą egzemplarzy każdego rodzaju przedmiotu.

Wynik: `K[0..n-1]` – tablica liczb całkowitych nieujemnych określających, ile egzemplarzy każdego przedmiotu należy spakować tak, aby łączna wartość była jak największa, a łączna waga była mniejsza lub równa `makswaga`; `makswart` – wartość spakowanych przedmiotów.

Podobnie jak w poprzednich przykładach zastosujemy strategię zachłanną. Należy się zastanowić, według jakiego kryterium wybierać kolejne przedmioty, aby w każdym kroku podejmować najkorzystniejszą decyzję. Najlepszym wyborem wydaje się uwzględnienie zarówno wartości, jak i wagi przedmiotów. Kryterium wyboru będzie więc stosunek wartości do wagi – im wyższy, tym korzystniej będzie spakować dany przedmiot. Przedmioty porządkujemy malejąco według tego stosunku.

Oto zapis algorytmu w pseudokodzie:

```
makswart ← 0  // łączna wartość spakowanych przedmiotów
dla i ← 0, 1, ..., n - 1 wykonuj
    K[i] ← makswaga div P[i].waga
    makswaga ← makswaga mod P[i].waga
    makswart ← makswart + K[i] * P[i].wart
```

Do reprezentacji pojedynczego przedmiotu (jego wartości i wagi) wykorzystamy **strukturę**. Umożliwi to uporządkowanie tablicy według określonego kryterium za pomocą **funkcji `sort`** z biblioteki STL. Oto kod źródłowy definiujący strukturę `przedmiot` i funkcję `Porownaj`:

```
1.  struct przedmiot
2.  {
3.      int wart, waga;
4.  };
5.
6.  bool Porownaj(przedmiot a, przedmiot b)
7.  {
8.      return (float(a.wart)/a.waga > float(b.wart)/b.waga);
9.  }
```

Warto wiedzieć

Stosunek ceny do jakości jest wykorzystywany przy tworzeniu różnego rodzaju rankingów, np. urządzeń elektronicznych.

Warto wiedzieć

Algorytm wyboru przedmiotów jest bardzo podobny do algorytmu wydawania reszty.

Dobra rada

Zauważ, że ze struktury korzystaliśmy też przy tworzeniu harmonogramu wykorzystania sali.

Struktura, s. 109

Funkcja `sort`, s. 214

Kod źródłowy struktury `przedmiot` i funkcji `Porownaj`, porównującej dwa przedmioty

W linii 8 używamy znaku większości, ponieważ chcemy posortować wartości od największej do najmniejszej. Interesuje nas też wynik dzielenia rzeczywistego, a nie całkowitego. Dlatego następuje **konwersja typu** na liczby zmiennoprzecinkowe – typ `float` oznacza takie właśnie liczby. Oto kod źródłowy funkcji `Plecak`, realizującej omówiony algorytm:

Konwersja (rzutowanie typu), s. 54

Kod źródłowy funkcji zwracającej maksymalną wartość przedmiotów, które można zapakować do plecaka

```
1.  int Plecak(przedmiot P[], int makswaga, int K[])
2.  {
3.      int makswart=0;
4.      for (int i=0;i<N;i++)
5.      {
6.          K[i]=makswaga/P[i].waga;
7.          makswaga=makswaga%P[i].waga;
8.          makswart+=K[i]*P[i].wart;
9.      }
10.     return makswart;
11. }
```

Warto wiedzieć

Tablice domyślnie przekazywane są przez wskaźnik. To znaczy, że wszystkie zmiany, które funkcja wprowadza w tablicy, zostają zachowane po zakończeniu działania funkcji.

Stała `N` oznacza liczbę rodzajów przedmiotów. Wartością funkcji jest łączna wartość spakowanych przedmiotów. Informacja o liczbie poszczególnych przedmiotów jest zwracana w tablicy `K`, przekazywanej przez wskaźnik.

Algorytm nie zawsze znajduje optymalną wartość, a odpowiedzi, które zwraca, należy traktować jak przybliżenia optymalnego rozwiązania. Dla danych z tabeli 18.2 i maksymalnej wagi 17 algorytm znajduje wartość 54, choć optymalnym rozwiązaniem jest 55.

Wartość	10	15	11	4	3
Waga	3	5	4	2	2
Iloraz	3,33...	3	2,75	2	1,5

Tabela 18.2. Przykładowy zestaw danych dla ogólnego problemu plecakowego

Ćwiczenie 7

a. Napisz program rozwiązujący w przybliżony sposób ogólny problem plecakowy. Dane wczytaj z pliku tekstowego, który otrzymasz od nauczyciela (np. *do_plecaka.txt*). Oto przykład wywołania programu:

```
Wartosc: 54
Liczby poszczegolnych przedmiotow:
5 o wartosci 10 i wadze 3
0 o wartosci 15 i wadze 5
0 o wartosci 11 i wadze 4
1 o wartosci 4 i wadze 2
0 o wartosci 3 i wadze 2
```

b. Które przedmioty i w jakiej liczbie należy spakować, aby dla danych z tabeli 18.2 otrzymać wartość 55?

Podsumowanie

- Metoda zachłanna polega na podejmowaniu najkorzystniejszej decyzji w danym kroku rozwiązania.
- Problem wydawania reszty polega na przedstawieniu danej kwoty reszty za pomocą jak najmniejszej liczby banknotów i monet. Aby rozwiązać ten problem metodą zachłanną, należy wybierać w danym kroku jak największy nominał nie większy od kwoty, która pozostała jeszcze do wydania.
- Problem ustalenia optymalnego harmonogramu wykorzystania sali polega na takim przypisaniu zajęć do sali, aby odbyło się ich jak najwięcej. Strategia zachłanna umożliwia optymalne rozwiązanie tego problemu: w danym kroku należy wybierać zajęcia, które kończą się najwcześniej.
- Problem plecakowy polega na takim wyborze przedmiotów do spakowania, aby ich łączna wartość była jak największa, a łączna waga nie przekroczyła dopuszczalnej.
- W ogólnym problemie plecakowym mamy do dyspozycji nieograniczoną liczbę przedmiotów każdego rodzaju. W decyzyjnym problemie plecakowym możemy zabrać tylko jeden przedmiot każdego rodzaju. Problem sprowadza się do decyzji, czy dany przedmiot spakować czy nie.
- Rozwiązując problem plecakowy metodą zachłanną, możemy nie otrzymać optymalnego rozwiązania (o największej wartości).
- Funkcja `sort` z biblioteki STL pozwala określić własne kryterium porównania dwóch elementów poprzez definicję funkcji o wartości logicznej.

Zadania

* **1** W pliku arkusza kalkulacyjnego, który otrzymasz od nauczyciela (np. o nazwie *wydaj_reszte_zlotowki.xlsx*), rozwiąż problem wydawania reszty tak, aby wyświetlane były kwoty w złotówkach i groszach.

* **2** Program poszukujący maksymalnej sumy w kwadracie zmodyfikuj tak, aby wypisywał także liczby tworzące sumę.

* **3** Zapisz w pseudokodzie funkcję realizującą algorytm wyboru przedmiotów w decyzyjnym problemie plecakowym.

** **4** Oszacuj złożoność obliczeniową algorytmów rozwiązywania problemów omówionych w tym temacie, stosujących metodę zachłanną. Przyjmij, że dane są już uporządkowane zgodnie z najlepszym kryterium dla danego problemu (nie musisz uwzględniać kosztu sortowania w szacowaniu złożoności obliczeniowej).

** **5** Napisz program znajdujący przybliżone rozwiązanie decyzyjnego problemu plecakowego metodą zachłanną.

** **6** W dużych kinach jest wiele sal kinowych, w których wyświetlane są różne filmy rozpoczynające się o różnych godzinach (filmy mają różną długość). Wyobraź sobie, że chcesz obejrzeć jak najwięcej filmów jednego dnia. Jest to tak zwany problem kinomana. Przygotuj specyfikację problemu i potrzebne dane oraz napisz program rozwiązujący ten problem.

** **7** Napisz program znajdujący przybliżone rozwiązanie ogólnego problemu plecakowego metodą zachłanną. Przyjmij, że wartości i wagi przedmiotów są pamiętane w dwóch oddzielnych tablicach liczb całkowitych. Przygotuj własną funkcję sortującą tablice według ilorazu wartości do wagi.

** **8** W zawodach sportowych startują pary. Łączna waga pary nie może przekroczyć określonej wagi. Dana jest liczba zawodników, ich wagi oraz maksymalna waga pary. Napisz program obliczający maksymalną liczbę par, które można utworzyć z tych zawodników.

*** **9** Napisz program poszukujący rozwiązania problemu wydawania reszty, gdy masz ograniczoną liczbę banknotów i monet o poszczególnych nominałach (należy utworzyć dodatkową tablicę pamiętającą dostępną liczbę poszczególnych banknotów i monet). Zwróć uwagę, że nie każdą resztę uda się wydać (program powinien wtedy wypisać odpowiedni komunikat).

19. Programowanie dynamiczne

Rozwiązaliście już wiele problemów, poznaliście różne algorytmy i napisaliście programy je realizujące. Być może zastawiasz się, czy programowanie dynamiczne to jakiś inny sposób programowania niż stosowane do tej pory. Termin ten oznacza nie tyle pisanie programów, ile technikę rozwiązywania problemów. W tym temacie poznasz tę metodę i wykorzystasz ją do znalezienia optymalnych rozwiązań poznanych wcześniej problemów.

Cele lekcji

- Zrozumiesz metodę programowania dynamicznego.
- Skonstruujesz algorytmy, które będą znajdować optymalne rozwiązania następujących problemów: wydawania reszty, poszukiwania drogi na planszy oraz pakowania plecaka.
- Znajdziesz najdłuższy wspólny podciąg dwóch ciągów.
- Porównasz metodę zachłanną i programowanie dynamiczne.

Programowanie dynamiczne to metoda algorytmiczna, w której problem rozwiązuje się poprzez rozwiązanie podproblemów dla mniejszych danych, w kolejności od najmniejszych do największych, i odpowiednie wykorzystanie otrzymanych wyników. Ponieważ często nie wiadomo, które wyniki posłużą do rozwiązania całego problemu, wyniki dla mniejszych danych zapamiętuje się zazwyczaj w tablicy.

Programowanie dynamiczne

Programowanie dynamiczne jest więc jedną z technik rozwiązywania problemów, a z programowaniem wiąże się wtedy, kiedy piszemy programy realizujące algorytmy, które wykorzystują tę metodę.

19.1. Wydawanie reszty metodą programowania dynamicznego

Zastosujemy metodę programowania dynamicznego do rozwiązania problemu **wydawania reszty**. W tym celu będziemy znajdować minimalną liczbę monet dla wszystkich pośrednich kwot reszty. Aby wyznaczyć rozwiązania dla kolejnych wartości, skorzystamy z już znalezionych wyników dla mniejszych wartości. Wyniki będą pamiętane w tablicy jedno- lub dwuwymiarowej.

Wydawanie reszty, s. 255

Ćwiczenie 1

Wyznacz metodą zachłanną liczbę monet potrzebnych do wydania reszty 15 zł, jeśli dysponujesz hipotetycznymi monetami: 1 zł, 3 zł, 5 zł i 8 zł. Czy jest to rozwiązanie optymalne?

Algorytm wykorzystujący programowanie dynamiczne rozpatrzymy na przykładzie wydawania 8 zł reszty, kiedy dysponujemy hipotetycznymi monetami o nominałach: 1 zł, 2 zł, 4 zł i 5 zł. Ograniczymy się do wyznaczenia minimalnej liczby monet potrzebnych do wydania reszty.

Powiedzieliśmy już, że wyniki cząstkowe będziemy przechowywać w tablicy. Dla wszystkich kwot mniejszych lub równych kwocie reszty (kolejne indeksy tablicy) będziemy pamiętać liczbę monet (wartości tablicy). Wypełnijmy tablicę początkowymi danymi.

Żeby wydać kwotę o wartości 0 zł, nie potrzebujemy żadnych monet. Dla pozostałych kwot cząstkowych początkową wartością liczby potrzebnych monet może być dowolna odpowiednio duża liczba, np. wartość o 1 większa od kwoty do wydania. W tablicy na rysunku 19.1 na zielono zaznaczona jest znaleziona już optymalna wartość, a na czerwono – wartości, które będą w kolejnych krokach aktualizowane. Będziemy poszukiwać niższych wartości.

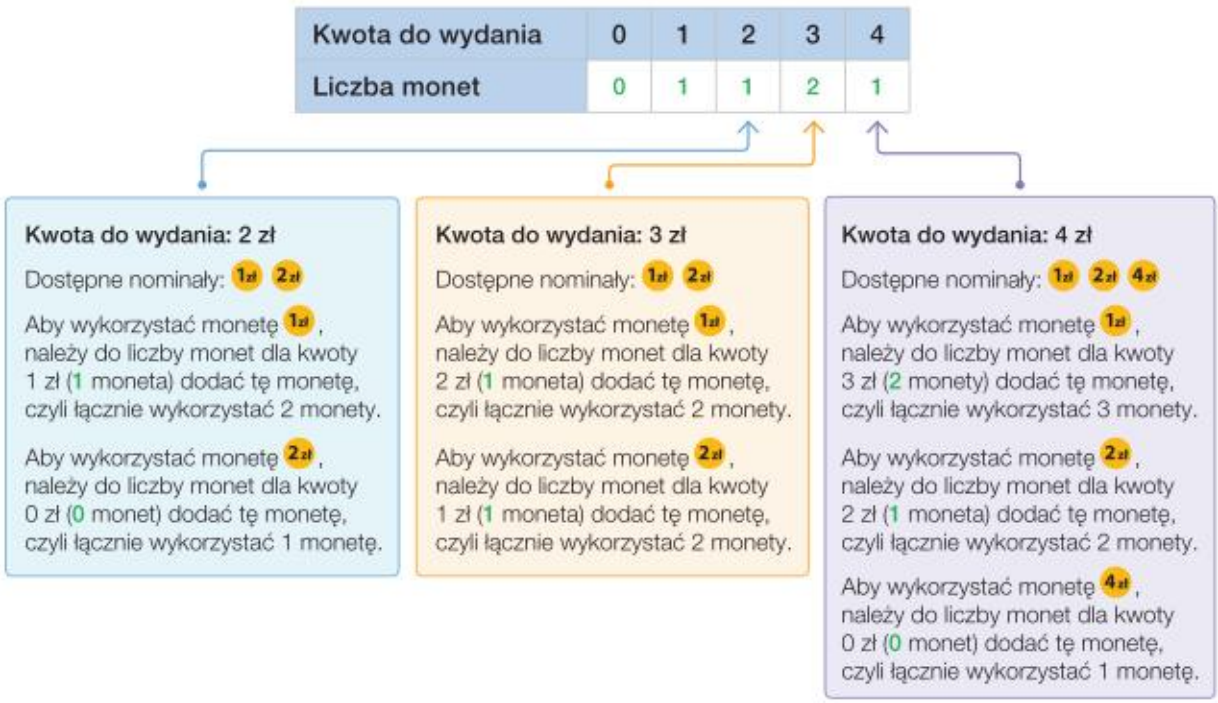
Warto wiedzieć

Przy wykorzystaniu systemu monetarnego z hipotetycznymi nominałami: 1 zł, 2 zł, 4 zł i 5 zł kwotę 8 zł można wydać co najwyżej 8 monetami o wartości 1 zł. Dlatego jako początkową liczbę monet można przyjąć 9.

Kwota do wydania	0	1	2	3	4	5	6	7	8
Liczba monet	0	9	9	9	9	9	9	9	9

Rys. 19.1. Wartości początkowe w tablicy

Zauważ, że kwotę 1 zł można wydać tylko i wyłącznie za pomocą monety 1 zł. Rysunek 19.2 przedstawia kolejne kroki algorytmu wyznaczającego najmniejszą liczbę monet dla kwot od 2 do 4 zł.



Rys. 19.2. Schemat postępowania dla wartości od 2 do 4 zł

Dla kolejnych kwot (od 5 do 8 zł) w podobny sposób analizowane są wartości nominałów mniejsze lub równe aktualnej kwocie. Dla danego nominału sprawdzimy, czy jeśli dołożymy jedną monetę o tym nominale, uzyskamy lepszy wynik (czyli mniejszą liczbę monet niż aktualnie wyznaczona). Uwzględniamy przy tym rozwiązania znalezione wcześniej dla mniejszych kwot.

Podczas obliczania ostatecznej liczby monet dla kwoty 8 zł najpierw zostanie rozpatrzona moneta jednozłotowa (liczba monet dla kwoty 7 zł plus 1 moneta), następnie moneta dwuzłotowa (liczba monet dla kwoty 6 zł plus 1 moneta), a potem moneta czterozłotowa (liczba monet dla kwoty 4 zł plus 1 moneta).

Rysunek 19.3 przedstawia wartości w tablicy po zakończeniu działania algorytmu.

Kwota do wydania	0	1	2	3	4	5	6	7	8
Liczba monet	0	1	1	2	1	1	2	2	2

Rys. 19.3. Wartości końcowe w tablicy po zakończeniu działania algorytmu

Warto wiedzieć

Metoda programowania dynamicznego jest pod pewnymi względami podobna do rekurencji, która też wykorzystuje rozwiązania dla mniejszych danych. Prostym przykładem zastosowania metody programowania dynamicznego jest iteracyjne obliczanie liczb Fibonacciego.

Ćwiczenie 2

a. Wyznacz liczbę monet potrzebnych do wydania reszty 10 zł, jeśli dysponujesz hipotetycznymi monetami: 1 zł, 5 zł i 7 zł. Wykorzystaj metodę programowania dynamicznego.

b. Czy istnieje sposób na wydanie 10 zł mniejszą liczbą monet, niż wskazuje to algorytm wykorzystujący metodę programowania dynamicznego?

Zapiszmy specyfikację problemu obliczenia minimalnej liczby monet potrzebnych do wydania danej kwoty.

Specyfikacja

Dane: `kwota` – reszta do wydania, liczba całkowita dodatnia,
`n` – liczba dostępnych nominałów,
`Nominały[0..n-1]` – tablica dostępnych nominałów.

Wynik: `ile_monet` – minimalna liczba monet potrzebna do wyrażenia wartości `kwota`, jeśli dysponujemy nieograniczoną liczbą monet o nominałach z tablicy `Nominały`.

Warto wiedzieć

Tablica `Nominały` nie musi być uporządkowana. Przy rozwiązywaniu problemu metodą zachłanną nominały były uporządkowane od największego do najmniejszego. W przypadku programowania dynamicznego kolejność, w jakiej przegląda się nominały, nie ma znaczenia, ponieważ są one przeglądane wielokrotnie dla każdej z pośrednich kwot.

W algorytmie użyjemy pomocniczej tablicy `IleM` o rozmiarze `kwota+1`, przechowującej minimalną liczbę monet dla poszczególnych kwot. Zapiszmy algorytm w pseudokodzie oraz kod źródłowy funkcji realizującej ten algorytm.

```
IleM[0] ← 0
dla i ← 1, 2, …, kwota wykonuj IleM[i] ← kwota + 1
dla i ← 1, 2, …, kwota wykonuj
    dla j ← 0, 1, …, n - 1 wykonuj
        jeśli Nominaly[j] ≤ i to
            jeśli IleM[i-Nominaly[j]] + 1 < IleM[i] to
                IleM[i] ← IleM[i-Nominaly[j]] + 1
ile_monet ← IleM[kwota]
```

Fragment kodu źródłowego programu wyznaczającego minimalną liczbę monet przy wydawaniu reszty metodą programowania dynamicznego

```
1.  const int N=4;
2.  const int NOMINALY[]={1,2,4,5};
3.  const int KWOTA=8;
4.
5.  int IleMonet()
6.  {
7.      int i, j;
8.      int IleM[KWOTA+1];
9.      IleM[0]=0;
10.     for (i=1;i<=KWOTA;i++) IleM[i]=KWOTA+1;
11.     for (i=1;i<=KWOTA;i++)
12.         for (j=0;j<N;j++)
13.             if (NOMINALY[j]<=i)
14.                 if (IleM[i-NOMINALY[j]]+1<IleM[i])
15.                     IleM[i]=IleM[i-NOMINALY[j]]+1;
16.     return IleM[KWOTA];
17. }
18.
19. int main()
20. {
21.     cout<<"Minimalna liczba monet dla kwoty ";
22.     cout<<KWOTA<<": "<<IleMonet();
23.     return 0;
24. }
```

Dobra rada

Pamiętaj, że rozmiar deklarowanej tablicy powinien być określony wprost, tzn. przez podanie liczby lub z wykorzystaniem stałej.

W linii 8 deklarowana jest tablica pomocnicza `IleM` o rozmiarze określonym przez stałą `KWOTA` zdefiniowaną w linii 3. W liniach 11–15 dla kolejnych kwot reszty analizowane są kolejne dostępne nominały. Jeżeli dołożenie monety o określonym nominale dla danej kwoty daje w efekcie mniejszą liczbę monet (sprawdzenie następuje w linii 14), to aktualizowana jest liczba monet (linia 15).

Ćwiczenie 3

a. Uruchom powyższy program i sprawdź jego działanie.

b. Zmodyfikuj linie kodu 1–3 tak, aby ustalić inny system monetarny i kwotę reszty, dla których metoda zachłanna nie znajduje optymalnego rozwiązania.

Optymalny rozkład monet

Kiedy rozwiązywaliśmy problem metodą zachłanną, znajdowaliśmy rozkład monet o poszczególnych nominałach. Żeby powyższy algorytm także wyznaczał rozkład monet, należy go uzupełnić tak, aby pamiętał nie tylko ich liczbę, lecz także rozkład dla cząstkowych kwot. Możemy do tego celu wykorzystać **tablicę dwuwymiarową**.

Kwoty	Nominały 1	2	4	5
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	1	1	0	0
4	0	0	1	0
5	0	0	0	1
6	1	0	0	1
7	0	1	0	1
8	0	0	2	0

Rys. 19.4. Rozkład monet o poszczególnych nominałach dla podanych kwot

Tablica dwuwymiarowa, s. 219

Wiersz tablicy (rys. 19.4) określa rozkład dla danej kwoty. Kolejne kolumny zawierają liczbę potrzebnych monet zgodnie z wartościami z tablicy `Nominaly`.

Pierwszy wiersz (dla kwoty 0 zł) jest inicjowany wartościami 0 (nie są potrzebne żadne monety). Każdy kolejny wiersz jest aktualizowany wtedy, gdy aktualizowana jest tablica z liczbą monet. Wówczas kopiowany jest wiersz kwoty mniejszej od aktualnie badanej o dany nominał, a następnie licznik tego nominału jest zwiększany o jeden. Ostatni wiersz tablicy zawiera rozkład monet dla wydawanej kwoty reszty.

Oto zmodyfikowana specyfikacja problemu:

Specyfikacja

Dane: `kwota` – reszta do wydania, liczba całkowita dodatnia, `n` – liczba dostępnych nominałów,
`Nominaly[0..n-1]` – tablica dostępnych nominałów.

Wynik: `ile_monet` – minimalna liczba monet potrzebna do wyrażenia wartości `kwota`, przy założeniu, że dysponujemy nieograniczoną liczbą monet o nominałach z tablicy `Nominaly`, `Reszta[0..n-1]` – tablica z liczbą poszczególnych monet o łącznej wartości `kwota`. Kolejność nominałów reszty jest zgodna z kolejnością w tablicy `Nominaly`.

Warto wiedzieć

Problem optymalnego rozkładu monet można również sformułować w taki sposób, aby uwzględniał ograniczoną liczbę dostępnych monet o poszczególnych nominałach.

W celu wyznaczenia rozkładu monet wykorzystamy dwuwymiarową tablicę pomocniczą o nazwie `ResztaM`, o rozmiarze `kwota+1` na `n`, w której zapiszemy cząstkowe rozkłady monet (patrz rys. 19.4). Ostatni wiersz tej tablicy będzie kopiowany do tablicy `Reszta` jako wynik.

Zapiszmy algorytm w pseudokodzie oraz zmodyfikowaną funkcję `IleMonet` w języku C++.

```
IleM[0] ← 0
dla i ← 1, 2, …, kwota wykonuj IleM[i] ← kwota + 1
dla i ← 0, 1, …, n – 1 wykonuj ResztaM[0][i] ← 0
dla i ← 1, 2, …, kwota wykonuj
    dla j ← 0, 1, …, n – 1 wykonuj
        jeśli Nominaly[j] ≤ i to
            jeśli IleM[i–Nominaly[j]] + 1 < IleM[i] to
                IleM[i] ← IleM[i–Nominaly[j]] + 1
                dla k ← 0, 1, …, n – 1 wykonuj
                    ResztaM[i][k]←ResztaM[i–Nominaly[j]][k]
                ResztaM[i][j] ← ResztaM[i][j] + 1
ile_monet ← IleM[kwota]
dla i ← 0, 1, …, n – 1 wykonuj
    Reszta[i] ← ResztaM[kwota][i]
```

Kod źródłowy funkcji wyznaczającej optymalny rozkład monet potrzebnych do wydania reszty metodą programowania dynamicznego

```
1.  int IleMonet(int Reszta[])
2.  {
3.      int i, j, k;
4.      int IleM[KWOTA+1];
5.      int ResztaM[KWOTA+1][N];
6.      IleM[0]=0;
7.      for (i=0;i<N;i++) ResztaM[0][i]=0;
8.      for (i=1;i<=KWOTA;i++) IleM[i]=KWOTA+1;
9.      for (i=1;i<=KWOTA;i++)
10.         for (j=0;j<N;j++)
11.             if (NOMINALY[j]<=i)
12.                 if (IleM[i-NOMINALY[j]]+1<IleM[i])
13.                 {
14.                     IleM[i]=IleM[i-NOMINALY[j]]+1;
15.                     for (k=0;k<N;k++)
16.                         ResztaM[i][k]=
17.                         ResztaM[i-NOMINALY[j]][k];
18.                     ResztaM[i][j]++;
19.                 }
20.     for (i=0;i<N;i++) Reszta[i]=ResztaM[KWOTA][i];
21.     return IleM[KWOTA];
22. }
```

Zwróć uwagę na modyfikację nagłówka funkcji. Rozkład monet jest zwracany za pośrednictwem parametru `Reszta`, który jest tablicą **przekazywaną przez wskaźnik**. Rozkład monet przechowujemy w tablicy dwuwymiarowej `ResztaM`, którą deklarujemy w linii 5. Rysunek 19.5 przedstawia wywołanie programu dla kwoty 8 zł.

Przekazanie parametru przez wskaźnik, s. 152

```
Minimalna liczba monet dla kwoty 8: 2
1: 0
2: 0
4: 2
5: 0
```

Rys. 19.5. Wywołanie programu znajdującego optymalny rozkład monet

Ćwiczenie 4

Napisz program, który wykorzysta zmodyfikowaną funkcję `IleMonet`. Sprawdź działanie programu dla różnych danych.

Wydawanie reszty – porównanie złożoności

Porównajmy obydwie metody rozwiązania problemu wydawania reszty: zachłanną oraz programowanie dynamiczne.

Metoda zachłanna znajduje rozwiązanie optymalne dla powszechnie stosowanych systemów monetarnych. Dla dowolnego zestawu nominałów często znajduje tylko rozwiązanie przybliżone. Jest prosta i ma stosunkowo krótki kod. Tablica o n elementach jest przeglądana raz, więc algorytm ma **złożoność czasową** rzędu $O(n)$ względem liczby nominałów.

Czasowa złożoność obliczeniowa, s. 164

Metoda programowania dynamicznego znajduje zawsze rozwiązanie optymalne, ale jej złożoność czasowa jest większa. Tablica jest przeglądana dla każdej kwoty pośredniej, więc złożoność czasowa algorytmu wynosi $O(n \cdot k)$, gdzie k oznacza kwotę do wydania, a n – liczbę nominałów. Złożoność ta zależy więc nie tylko od liczby nominałów, lecz także od kwoty do wydania. Dla dużej kwoty metoda staje się mało efektywna. Jej **złożoność pamięciowa** wynosi $O(n \cdot k)$, gdy chcemy poznać rozkład banknotów lub monet, a nie tylko ich liczbę potrzebną do wydania reszty.

Pamięciowa złożoność obliczeniowa, s. 167

19.2. Maksymalna suma w kwadracie – programowanie dynamiczne

W poprzednim temacie poszukiwaliśmy **maksymalnej sumy** w obrębie kwadratowej planszy, przechodząc od lewego górnego rogu do prawego dolnego. Dozwolone były tylko dwa typy ruchów: przejście na sąsiednie pole w prawo lub do dołu.

Maksymalna suma w kwadracie, s. 262

Metoda zachłanna nie zawsze znajdowała największą sumę, czyli nie prowadziła do rozwiązania optymalnego. Pokazuje to rysunek 19.6.

Dobra rada

Pamiętaj, że algorytmy stosujące metodę zachłanną wybierają najlepsze rozwiązanie w danym kroku, bez analizy skutków tej decyzji w kolejnych krokach.

	0	1	2	3	4
0	5	5	1	3	9
1	8	6	4	9	8
2	5	1	3	7	4
3	5	6	2	8	3
4	5	6	3	6	5

Metoda zachłanna: suma = 52

	0	1	2	3	4
0	5	5	1	3	9
1	8	6	4	9	8
2	5	1	3	7	4
3	5	6	2	8	3
4	5	6	3	6	5

Rozwiązanie optymalne: suma = 58

Rys. 19.6. Przykładowa plansza z rozwiązaniem wyznaczonym metodą zachłanną oraz rozwiązanie optymalne

Zastosowanie programowania dynamicznego do rozwiązania tego problemu będzie polegało na wyliczeniu częściowych sum dla każdego pola tak, jakby ono było polem docelowym. Najłatwiej określić tę sumę dla pola startowego – będzie to wartość zapisana w tym polu, ponieważ droga prowadząca do niego składa się tylko z tego pola.

Warto wiedzieć

Zgodnie z zasadami poruszania się po kwadracie do pola w pierwszej kolumnie można dojść wyłącznie z pola powyżej, a do pola w pierwszym wierszu – wyłącznie z pola po lewej stronie.

Łatwo również policzyć sumy dla pierwszego wiersza i pierwszej kolumny. Wypełnimy pierwszy wiersz od lewej do prawej i pierwszą kolumnę od góry do dołu. Na rysunku 19.7 na zielono zaznaczone są sumy dla pól w pierwszym wierszu i pierwszej kolumnie.

	0	1	2	3	4
0	5	5	1	3	9
1	8	6	4	9	8
2	5	1	3	7	4
3	5	6	2	8	3
4	5	6	3	6	5

→

	0	1	2	3	4
0	5	10	11	14	23
1	13	6	4	9	8
2	18	1	3	7	4
3	23	6	2	8	3
4	28	6	3	6	5

Rys. 19.7. Plansza po wypełnieniu sumami pierwszego wiersza i pierwszej kolumny

Dobra rada

Wartości możesz uzupełnić również od lewej do prawej, zaczynając od drugiej kolumny.

Podobnie możemy modyfikować kolejne wiersze tablicy. Zastąpimy istniejącą wartość sumą tej liczby i liczby z poprzedniego pola – wybieramy pole o większej wartości (sąsiednie na lewo lub u góry).

Na przykład do pola (1,1) można dotrzeć z:

- pola (0,1) o wartości 10 – wtedy suma w polu (1,1) wyniosłaby 10 + 6 = 16,
- pola (1,0) o wartości 13 – wtedy suma w polu (1,1) wyniosłaby 13 + 6 = 19.

Korzystniej wybrać drogę o wartości 19, więc taką wartość należy wpisać w polu (1,1). Tak samo postępujemy z kolejnymi polami.

Rysunek 19.8 przedstawia planszę ze wszystkimi policzonymi sumami.

	0	1	2	3	4
0	5	10	11	14	23
1	13	6	4	9	8
2	18	1	3	7	4
3	23	6	2	8	3
4	28	6	3	6	5

→

	0	1	2	3	4
0	5	10	11	14	23
1	13	19	23	32	40
2	18	20	26	39	44
3	23	29	31	47	50
4	28	35	38	53	58

Rys. 19.8. Plansza z wyliczonymi wszystkimi sumami

Ćwiczenie 5

W pliku, który otrzymasz od nauczyciela (np. *suma_w_kwadracie.xlsx*), wyznacz maksymalną wartość sumy oraz ścieżkę, która doprowadzi do tej sumy.

Oto specyfikacja problemu i zapis algorytmu w pseudokodzie:

Specyfikacja

Dane: n – liczba wierszy i kolumn planszy, P[0..n–1][0..n–1] – dwuwymiarowa tablica liczb całkowitych dodatnich.

Wynik: suma – maksymalna suma liczb w polach od P[0][0] do P[n–1][n–1], dozwolone ruchy to przejście na sąsiednie pole w prawo lub w dół.

```
dla i ← 1, 2, ..., n – 1 wykonuj
    P[0][i] ← P[0][i] + P[0][i–1]
    P[i][0] ← P[i][0] + P[i–1][0]
dla i ← 1, 2, ..., n – 1 wykonuj
    dla j ← 1, 2, ..., n – 1 wykonuj
        jeśli P[i–1][j] > P[i][j–1] to
            P[i][j] ← P[i][j] + P[i–1][j]
        w przeciwnym przypadku
            P[i][j] ← P[i][j] + P[i][j–1]
suma ← P[n–1][n–1]
```

Kod funkcji realizującej powyższy algorytm można zapisać następująco:

Kod źródłowy funkcji wyznaczającej maksymalną sumę w kwadracie

```
1.  int MaksSuma(int P[][N])
2.  {
3.      for (int i=1;i<N;i++)
4.      {
5.          P[0][i]+=P[0][i-1];
6.          P[i][0]+=P[i-1][0];
7.      }
8.      for (int i=1;i<N;i++)
9.          for (int j=1;j<N;j++)
10.             P[i][j]+=max(P[i-1][j],P[i][j-1]);
11.     return P[N-1][N-1];
12. }
```

Zwróć uwagę na linie 5–6 oraz 10. Zastosowano w nich skrócony zapis powiększenia zmiennej (w tym przypadku elementu tablicy dwuwymiarowej) o określoną wartość. W linii 10 korzystamy dodatkowo z funkcji `max`, której wynikiem jest większy z dwóch parametrów.

Dobra rada

Instrukcję x=x+y możesz zapisać w skrócie x+=y. Analogiczne skróty możesz stosować dla pozostałych działań arytmetycznych.

Warto wiedzieć

Funkcja `max` znajduje się w bibliotece standardowej.

Dobra rada

Możesz dodatkowo określić trzeci parametr funkcji `max`, który jest nazwą funkcji logicznej wskazującej sposób porównania parametrów. Parametr ten odgrywa taką samą rolę jak w przypadku funkcji `sort`.

Ćwiczenie 6

Napisz program wykorzystujący funkcję MaksSuma. Pamiętaj o funkcjach losującej i wypisującej planszę.

Maksymalna suma w kwadracie – porównanie złożoności

Algorytm zachłanny znajduje rozwiązanie przybliżone, które nie zawsze jest największą sumą, ale jego złożoność czasowa jest liniowa – wynosi $O(n)$.

Algorytm z wykorzystaniem programowania dynamicznego zawsze wyznacza maksymalną sumę, ale przegląda całą planszę, więc jego złożoność czasowa jest kwadratowa ($O(n^2)$).

Zapamiętaj

Metoda programowania dynamicznego polega na znajdowaniu optymalnego rozwiązania na podstawie wcześniej znalezionych najlepszych rozwiązań dla pośrednich wartości danych.

19.3. Pakowanie plecaka metodą programowania dynamicznego

Problem plecakowy, s. 265

Metoda zachłanna stosowana w **problemie plecakowym** pozwalała znaleźć rozwiązanie przybliżone, a nie maksymalną wartość przedmiotów.

Dzięki programowaniu dynamicznemu można znaleźć rozwiązanie optymalne. Polega ono na wyliczaniu najlepszych wartości dla kolejnych wag plecaka od zera do maksymalnej wagi, z uwzględnieniem wcześniej policzonych wartości. Dla każdej wagi należy przejrzeć wszystkie przedmioty, przy czym kolejność ich przeglądania nie ma znaczenia.

Dobra rada

Zauważ, że problemy pakowania plecaka i wydawania reszty są podobne.

Rysunek 19.9 przedstawia dane dotyczące przedmiotów, które należy zapakować do plecaka o maksymalnej wadze 17 kg.

Nr przedmiotu	0	1	2	3	4
Wartość	3	4	10	11	15
Waga	2	2	3	4	5

Rys. 19.9. Przykładowe dane do problemu plecakowego

Utwórzmy tablicę, w której wartości początkowe dla każdej wagi plecaka są zerowe. Będziemy poszukiwać większych wartości. Dla wag 0 i 1 pozostaną one zerowe, ponieważ każdy z przedmiotów ma większą wagę jednostkową (najmniejsza waga przedmiotu to 2).

Rysunek 19.10 przedstawia tablicę z obliczonymi wartościami dla poszczególnych wag. Ostatni element tablicy o wartości 55 stanowi rozwiązanie problemu. Jest to maksymalna wartość upakowania plecaka dla wagi 17 kg.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0	4	10	11	15	20	21	25	30	31	35	40	41	45	50	51	55

Rys. 19.10. Wartości upakowania plecaka dla poszczególnych wag

Podobnie jak w przypadku **algorytmu wydawania reszty** przeglądamy wszystkie przedmioty. Dla aktualnej wagi i wybranego przedmiotu określamy wartość plecaka po dołożeniu do niego tego przedmiotu. W tym celu do wartości plecaka dla wagi mniejszej o wagę tego przedmiotu dodajemy jego wartość. Jeśli otrzymamy większą wartość, to modyfikujemy tablicę.

Pokażemy, w jaki sposób została wyznaczona wartość 55 dla wagi 17 (znane są wartości dla wag plecaka od 0 do 16).

1. Analizujemy przedmiot 0 (wartość: 3, waga: 2). Do wartości 50 dla wagi 15 dodajemy wartość przedmiotu 0 i otrzymujemy wartość 53. Zapamiętujemy tę wartość.
2. Analizujemy przedmiot 1 (wartość: 4, waga: 2). Do wartości 50 dla wagi 15 dodajemy wartość przedmiotu 1 i otrzymujemy wartość 54. Jest ona większa niż w punkcie 1. Zapamiętujemy tę wartość.
3. Analizujemy przedmiot 2 (wartość: 10, waga: 3). Do wartości 45 dla wagi 14 dodajemy wartość przedmiotu 2 i otrzymujemy wartość 55. Jest ona większa niż w punkcie 2. Zapamiętujemy tę wartość.
4. Analizujemy przedmiot 3 (wartość: 11, waga: 4). Do wartości 41 dla wagi 13 dodajemy wartość przedmiotu 3 i otrzymujemy wartość 52. Nie jest ona większa niż wcześniej znaleziona.
5. Analizujemy przedmiot 4 (wartość: 15, waga: 5). Do wartości 40 dla wagi 12 dodajemy wartość przedmiotu 4 i otrzymujemy wartość 55. Nie jest ona większa niż wcześniej znaleziona.

W ten sam sposób zostały wyliczone wszystkie wcześniejsze wartości dla mniejszych wag plecaka.

Wydawanie reszty metodą programowania dynamicznego, s. 271–277

Dobra rada

Zauważ, że jeśli przedmiot, który chcemy dopakować, ma wagę 2, a plecak maksymalną wagę 17, to musimy przedmiot ten dopakować do wagi 15. Wartość dla wagi 15 powiększona o wartość dopakowanego przedmiotu będzie stanowiła nową wartość plecaka o wadze 17.

Ćwiczenie 7

a. Wypisz wszystkie kroki, w których obliczysz wartość dla wagi 8.

b. Opracujcie w grupie prostą animację lub pokaz slajdów, które omówią problem plecakowy oraz algorytm jego rozwiązania z wykorzystaniem metody programowania dynamicznego.

A to ciekawe

Czy dowódca zawsze ma rację?

Metodę programowania dynamicznego opracował w latach 50. XX w. amerykański matematyk Richard Bellman na potrzeby wojska. Skąd nazwa „programowanie dynamiczne"? Według jednej z anegdot Bellman chciał uniknąć terminów matematycznych, ponieważ jeden z dowódców amerykańskiego wojska, dla którego pracował Bellman, był niechętny badaniom z tej dziedziny. Inna anegdota głosi, że nazwa ta powstała przez modyfikację nazwy innej metody – programowania liniowego.

Oto specyfikacja problemu i zapis algorytmu w pseudokodzie:

Specyfikacja

Dane: n – liczba rodzajów przedmiotów, P[0..n–1] – tablica przedmiotów, przedmiot reprezentowany jest przez dwie liczby całkowite dodatnie waga i wart, określające wagę i wartość przedmiotu, makswaga – maksymalna łączna waga przedmiotów, które można spakować. Dysponujemy nieograniczoną liczbą egzemplarzy każdego przedmiotu.

Wynik: makswart – maksymalna wartość spakowanych przedmiotów o łącznej wadze mniejszej lub równej makswaga.

```
dla i ← 0, 1, …, makswaga wykonuj W[i] ← 0
dla i ← 0, 1, …, makswaga wykonuj
    dla j ← 0, 1, …, n – 1 wykonuj
        jeśli P[j].waga ≤ i to
            jeśli W[i–P[j].waga] + P[j].wart > W[i] to
                W[i] ← W[i–P[j].waga] + P[j].wart
makswart ← W[makswaga]
```

W tablicy W przechowamy wartości plecaka dla kolejnych wag pośrednich. Przedmioty mogą być reprezentowane tak jak przy rozwiązaniu problemu metodą zachłanną, jako tablica **struktur**. Nie musimy tej tablicy sortować, więc nie jest potrzebna funkcja porównująca dwa przedmioty. Oto kod źródłowy funkcji realizującej powyższy algorytm z wykorzystaniem programowania dynamicznego:

Struktura, s. 109

Kod źródłowy funkcji wyznaczającej maksymalną wartość przedmiotów, które można zapakować do plecaka

```
1.  int Plecak(przedmiot P[])
2.  {
3.      int W[MAKSWAGA+1];
4.      for (int i=0;i<=MAKSWAGA;i++) W[i]=0;
5.      for (int i=0;i<=MAKSWAGA;i++)
6.          for (int j=0;j<N;j++)
7.              if (P[j].waga<=i)
8.                  if (W[i-P[j].waga]+P[j].wart>W[i])
9.                      W[i]=W[i-P[j].waga]+P[j].wart;
10.     return W[MAKSWAGA];
11. }
```

Ćwiczenie 8

Napisz program rozwiązujący ogólny problem plecakowy metodą programowania dynamicznego z wykorzystaniem funkcji Plecak. Dane wczytaj z pliku tekstowego, który otrzymasz od nauczyciela (np. *do_plecaka.txt*).

Zauważ, że tablica `W` została zadeklarowana z wykorzystaniem stałej `MAKSWAGA`. Określa ona maksymalną wagę plecaka i powinna być zdefiniowana na początku kodu źródłowego.

Optymalny rozkład przedmiotów w plecaku

Uzupełnimy teraz algorytm tak, aby podawał, które przedmioty należy spakować. Rozkład przedmiotów dla poszczególnych wag cząstkowych plecaka będziemy pamiętali w **tablicy dwuwymiarowej** (rys. 19.11).

Wiersz tablicy określa rozkład dla danej wagi plecaka. Kolejne kolumny zawierają liczbę pakowanych przedmiotów. Dany wiersz jest aktualizowany wtedy, gdy aktualizowana jest tablica z maksymalną liczbą przedmiotów. Kopiowany jest wiersz dla wagi plecaka równej różnicy między rozpatrywaną wagą i wagą aktualnie analizowanego przedmiotu. Następnie licznik przedmiotu tego rodzaju jest zwiększany o jeden. Ostatni wiersz tabeli zawiera rozkład przedmiotów dla maksymalnej wagi plecaka.

Oto zaktualizowana specyfikacja problemu:

Maksymalna waga \ Numer przedmiotu	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	1	0	0	0
3	0	0	1	0	0
4	0	0	0	1	0
5	0	0	0	0	1
6	0	0	2	0	0
7	0	0	1	1	0
8	0	0	1	0	1
9	0	0	3	0	0
10	0	0	2	1	0
11	0	0	2	0	1
12	0	0	4	0	0
13	0	0	3	1	0
14	0	0	3	0	1
15	0	0	5	0	0
16	0	0	4	1	0
17	0	0	4	0	1

Rys. 19.11. Rozkład przedmiotów dla poszczególnych wag plecaka

Specyfikacja

Dane: `n` – liczba rodzajów przedmiotów, `P[0..n–1]` – tablica przedmiotów, przedmiot jest reprezentowany przez dwie liczby całkowite dodatnie `waga` i `wart`, określające wagę i wartość przedmiotu, `makswaga` – maksymalna łączna waga przedmiotów, które można spakować. Dysponujemy nieograniczoną liczbą egzemplarzy każdego przedmiotu.

Wynik: `makswart` – maksymalna wartość spakowanych przedmiotów o łącznej wadze mniejszej lub równej `makswaga`, `K[0..n–1]` – tablica liczb całkowitych nieujemnych określających, ile egzemplarzy każdego przedmiotu należy spakować, aby wartość plecaka wynosiła `makswart`.

Tablica dwuwymiarowa, s. 219

Warto wiedzieć

Modyfikacja algorytmu plecakowego, tak aby wskazywał, które przedmioty należy spakować, jest podobna jak dla problemu wydawania reszty, w którym podaje się rozkład monet o wybranych nominałach.

Dobra rada

Zauważ, że indeksami tablicy `W` są kolejne wagi plecaka, dlatego zapis `i-P[j].waga` dla określonego `i` jest prawidłowy.

Zapis algorytmu w pseudokodzie może wyglądać następująco:

```
dla i ← 0, 1, ..., makswaga wykonuj
    W[i] ← 0
    dla j ← 0, 1, ..., n - 1 wykonuj KPom[i][j] ← 0
dla i ← 0, 1, ..., makswaga wykonuj
    dla j ← 0, 1, ..., n - 1 wykonuj
        jeśli P[j].waga ≤ i to
            jeśli W[i-P[j].waga] + P[j].wart > W[i] to
                W[i] ← W[i-P[j].waga] + P[j].wart
                dla m ← 0, 1, ..., n - 1 wykonuj
                    KPom[i][m] ← KPom[i-P[j].waga][m]
                KPom[i][j] ← KPom[i][j] + 1
dla i ← 0, 1, ..., n - 1 wykonuj K[i] ← KPom[makswaga][i]
makswart ← W[makswaga]
```

W pseudokodzie została wykorzystana pomocnicza tablica dwuwymiarowa `KPom`, przechowująca liczbę przedmiotów każdego rodzaju, które należy spakować dla wag od 0 do 17 kg (patrz rys. 19.11, s. 283). Oto kod źródłowy funkcji w języku C++:

Kod źródłowy funkcji zwracającej rozkład przedmiotów, które można zapakować do plecaka

```
1.  int Plecak(przedmiot P[], int K[])
2.  {
3.      int W[MAKSWAGA+1];
4.      int KPom[MAKSWAGA+1][N];
5.      for (int i=0;i<=MAKSWAGA;i++)
6.      {
7.          W[i]=0;
8.          for (int j=0;j<N;j++) KPom[i][j]=0;
9.      }
10.     for (int i=0;i<=MAKSWAGA;i++)
11.         for (int j=0;j<N;j++)
12.             if (P[j].waga<=i)
13.                 if (W[i-P[j].waga]+P[j].wart>W[i])
14.                 {
15.                     W[i]=W[i-P[j].waga]+P[j].wart;
16.                     for (int m=0;m<N;m++)
17.                         KPom[i][m]=
18.                         KPom[i-P[j].waga][m];
19.                     KPom[i][j]++;
20.                 }
21.     for (int i=0;i<N;i++) K[i]=KPom[MAKSWAGA][i];
22.     return W[MAKSWAGA];
23. }
```

Zauważ, że funkcja `Plecak` ma dodatkowy parametr – tablicę `K`, w której po zakończeniu działania funkcji jest pamiętany rozkład przedmiotów do spakowania. W linii 4 deklarowana jest tablica `KPom`, a w pętli w liniach 5–9 jej elementy są zerowane. W liniach 16–18 następuje modyfikacja wiersza tablicy, wtedy gdy modyfikowana jest

również wartość upakowania plecaka. W linii 21 ostatni wiersz tablicy pomocniczej jest kopiowany do parametru K.

Ćwiczenie 9

Program rozwiązujący ogólny problem plecakowy metodą programowania dynamicznego zmodyfikuj tak, aby oprócz maksymalnej wartości wypisywał także informację, które przedmioty i w jakiej liczbie spakować.

Pakowanie plecaka – porównanie złożoności

Algorytm zachłanny znajduje rozwiązanie przybliżone, które nie zawsze jest optymalne, ale jego **złożoność czasowa** jest liniowa ($O(n)$).

Czasowa złożoność obliczeniowa, s. 164

Złożoność czasowa algorytmu wykorzystującego programowanie dynamiczne zależy nie tylko od liczby rodzajów przedmiotów, lecz także od maksymalnej wagi. Złożoność ta wynosi $O(n \cdot w)$, gdzie n określa liczbę różnych przedmiotów, a w maksymalną dopuszczalną wagę. **Złożoność pamięciowa** również wynosi $O(n \cdot w)$, gdy chcemy poznać rozkład przedmiotów.

Pamięciowa złożoność obliczeniowa, s. 167

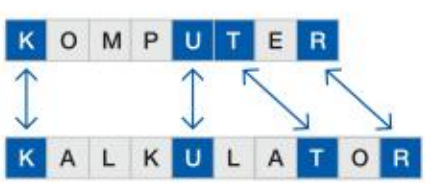
Zapamiętaj

Programowanie dynamiczne pozwala znaleźć optymalne rozwiązania wielu problemów, ale złożoność czasowa oraz pamięciowa algorytmów może być wysoka i zależeć od wartości danych. Złożoność czasowa metody zachłannej jest zwykle liniowa. Metoda ta często znajduje tylko rozwiązanie przybliżone, ale w wielu sytuacjach takie rozwiązania wystarczają.

19.4. Znajdowanie najdłuższego wspólnego podciągu

Metodę programowania dynamicznego wykorzystamy, aby wyznaczyć najdłuższy wspólny podciąg dwóch napisów. Elementy podciągu nie muszą w napisach znajdować się obok siebie, ale muszą występować w tej samej kolejności.

Na przykład dla napisów KOMPUTER i KALKULATOR najdłuższym wspólnym podciągiem jest KUTR. Przedstawia to rysunek 19.12.



Rys. 19.12. Najdłuższy wspólny podciąg słów KOMPUTER i KALKULATOR

Dobra rada

Zwróć uwagę, że pierwsza litera wspólnego podciągu dla słów KOMPUTER i KALKULATOR występuje w drugim z tych słów dwa razy i może być wybrana dowolnie.

Wyznaczanie długości najdłuższego podciągu

Warto wiedzieć

Algorytm znajdowania najdłuższego wspólnego podciągu ma zastosowanie m.in. w badaniu kodów DNA i w systemach antyplagiatowych.

Zajmijmy się najpierw szukaniem długości najdłuższego wspólnego podciągu. Oto specyfikacja problemu:

Specyfikacja

Dane: `s`, `t` – dwa napisy złożone z wielkich liter alfabetu łacińskiego – maksymalna długość każdego z nich wynosi 20 liter.

Wynik: `dl` – długość najdłuższego wspólnego podciągu.

Będziemy wyznaczać długości najdłuższych wspólnych podciągów dla kolejnych początkowych fragmentów napisów. Załóżmy, że znamy taką długość dla $i - 1$ pierwszych znaków napisu `s` oraz $j - 1$ pierwszych znaków napisu `t`. Wówczas długość $d_{i,j}$ najdłuższego podciągu dla i pierwszych znaków napisu `s` oraz j pierwszych znaków napisu `t` można określić wzorem:

Warto wiedzieć

Można skorzystać wprost ze wzoru rekurencyjnego na wyznaczanie maksymalnej długości podciągu. Prowadzi to jednak do algorytmu o złożoności wykładniczej, tak jak w przypadku rekurencyjnego wyznaczania liczb Fibonacciego.

$$d_{i,j} = \begin{cases} 0 & \text{dla } i = 0 \text{ lub } j = 0 \\ d_{i-1,j-1} + 1 & \text{dla } s_i = t_j, 0 < i \leqslant n, 0 < j \leqslant m \\ \max(d_{i-1,j}, d_{i,j-1}) & \text{dla } s_i \neq t_j, 0 < i \leqslant n, 0 < j \leqslant m \end{cases}$$

gdzie n oznacza długość napisu `s`, s_i – i-ty znak napisu `s`, m – długość napisu `t`, a t_j – j-ty znak napisu `t`.

Jeśli jeden z napisów jest pusty, to długość podciągu jest równa 0. Jeśli i-ty znak napisu `s` i j-ty znak napisu `t` są takie same, to dotychczas znalezioną długość powiększamy o 1. W przeciwnym przypadku wybieramy większą z wartości dotychczas znalezionych.

Zbudujemy tablicę wartości $d_{i,j}$. Dla napisów KOMPUTER i KALKULATOR będzie ona wyglądać tak jak na rysunku 19.13. Wartość w prawym dolnym rogu ($d_{n,m}$) jest wynikiem.

		K	A	L	K	U	L	A	T	O	R
	0	0	0	0	0	0	0	0	0	0	0
K	0	1	1	1	1	1	1	1	1	1	1
O	0	1	1	1	1	1	1	1	1	2	2
M	0	1	1	1	1	1	1	1	1	2	2
P	0	1	1	1	1	1	1	1	1	2	2
U	0	1	1	1	1	2	2	2	2	2	2
T	0	1	1	1	1	2	2	2	3	3	3
E	0	1	1	1	1	2	2	2	3	3	3
R	0	1	1	1	1	2	2	2	3	3	4

Rys. 19.13. Tablica pozwalająca wyznaczyć długość najdłuższego wspólnego podciągu dla kolejnych fragmentów napisów KOMPUTER i KALKULATOR

Ćwiczenie 10

Utwórz tabelę maksymalnych długości i wyznacz długość najdłuższego wspólnego podciągu dla napisów:

a. AUTOSTRADA i STRATEGIA,

b. LITERATURA i MATEMATYKA.

Zapis omówionego algorytmu w pseudokodzie jest następujący:

```
n ← długość napisu s
m ← długość napisu t
dla i ← 0, 1, …, n wykonuj d[i][0] ← 0
dla j ← 0, 1, …, m wykonuj d[0][j] ← 0
dla i ← 1, …, n wykonuj
    dla j ← 1, …, m wykonuj
        jeśli s[i-1] = t[j-1] to
            d[i][j] ← d[i-1][j-1] + 1
        w przeciwnym przypadku
            d[i][j] ← maksimum(d[i-1][j],d[i][j-1])
dl ← d[n][m]
```

Zwróć uwagę na indeksy przy porównaniu znaków napisów. Ponieważ elementy napisów są indeksowane od 0, wartości indeksów są równe `i-1` oraz `j-1` (indeksy `i` oraz `j` odpowiadają długościom napisów w tablicy, więc wyrażają wartości odpowiednio od 1 do n oraz od 1 do m). Kod źródłowy funkcji `main` realizującej powyższy algorytm można zapisać następująco:

Kod źródłowy funkcji głównej programu wyznaczającego najdłuższy wspólny podciąg dla dwóch napisów

```
1.  int main()
2.  {
3.      int i, j, n, m;
4.      int d[MAX_DL+1][MAX_DL+1];
5.      string s, t;
6.      cout<<"Pierwszy napis: "; cin>>s;
7.      cout<<"Drugi napis: "; cin>>t;
8.      n=s.size(); m=t.size();
9.      for (i=0;i<=n;i++) d[i][0]=0;
10.     for (j=0;j<=m;j++) d[0][j]=0;
11.     for (i=1;i<=n;i++)
12.         for (j=1;j<=m;j++)
13.             if (s[i-1]==t[j-1]) d[i][j]=d[i-1][j-1]+1;
14.             else d[i][j]=max(d[i-1][j],d[i][j-1]);
15.     cout<<"Najwieksza dl. wspolnego podciagu: "<<d[n][m];
16.     return 0;
17. }
```

Dwuwymiarowa tablica `d` zadeklarowana w linii 4 ma wymiary o jeden większe od wartości stałej `MAX_DL` (maksymalna długość napisu), ponieważ uwzględnia też napisy o długości 0. Długości wczytanych napisów (linie 6–7) pamiętane są w zmiennych `n` i `m` (linia 8). Ich wartości muszą być mniejsze lub równe stałej `MAX_DL`.

Wyznaczanie najdłuższego podciągu

Wyznaczymy teraz najdłuższy podciąg z wykorzystaniem informacji o długości najdłuższych podciągów dla początkowych fragmentów napisów. Oto specyfikacja problemu.

Specyfikacja

Dane: `s, t` – dwa napisy złożone z wielkich liter alfabetu łacińskiego – maksymalna długość każdego z nich wynosi 20 liter.

Wynik: `nwp` – najdłuższy wspólny podciąg (dowolny, jeśli jest więcej niż jeden).

Warto wiedzieć

Przechodzenie po przekątnej tablicy dwuwymiarowej oznacza jednoczesne zwiększanie lub zmniejszanie indeksów w kolumnie i wierszu tej tablicy.

Tablicę przeglądamy od ostatniego elementu do lewej kolumny lub górnego wiersza. Jeśli znaki w napisach na pozycjach odpowiadających danemu elementowi tablicy są takie same, to znak dodajemy na początku podciągu wynikowego i przechodzimy do elementu po przekątnej.

Jeśli znaki są różne, wybieramy element na lewo lub u góry o większej wartości (zmniejszamy numer wiersza lub kolumny). Dla takich samych wartości wybieramy ruch w górę. Ilustruje to rysunek 19.14.

Przypadek 1: litery (E i O) są różne, a sąsiednie pola mają tę samą wartość. Wybieramy ruch w górę, czyli zmniejszamy wartość `i` o jeden.

	...	A	T	O	R
...					
U		2	2	2	2
T		2	3	3	3
E		2	3	3	3
R		2	3	3	4

`d[i][j-1]` ≥ `d[i-1][j]`

Przypadek 2: litery (T i O) są różne, a pole po lewej ma wyższą wartość niż pole powyżej. Wybieramy ruch w lewo, czyli zmniejszamy wartość `j` o jeden.

	...	A	T	O	R
...					
U		2	2	2	2
T		2	3	3	3
E		2	3	3	3
R		2	3	3	4

`d[i][j-1]` > `d[i-1][j]`

Rys. 19.14. Wybór pola przy wyznaczaniu wspólnego podciągu – analiza dwóch przypadków

Dobra rada

Gdy znaki są różne, a sąsiednie wartości równe, możesz też wybrać ruch w lewo.

Warto wiedzieć

Jeśli kilka podciągów dwóch napisów ma maksymalną długość, można wybrać dowolny z nich.

Rysunek 19.15 pokazuje sposób wyznaczenia podciągu dla napisów KOMPUTER i KALKULATOR. Elementy zaznaczone na zielono wskazują znaki wchodzące w skład najdłuższego wspólnego podciągu. Na czerwono zaznaczono miejsca, przez które tylko przechodzimy, wyznaczając najdłuższy wspólny podciąg.

		K	A	L	K	U	L	A	T	O	R
	0	0	0	0	0	0	0	0	0	0	0
K	0	1	1	1	1	1	1	1	1	1	1
O	0	1	1	1	1	1	1	1	1	2	2
M	0	1	1	1	1	1	1	1	1	2	2
P	0	1	1	1	1	1	1	1	1	2	2
U	0	1	1	1	1	2	2	2	2	2	2
T	0	1	1	1	1	2	2	2	3	3	3
E	0	1	1	1	1	2	2	2	3	3	3
R	0	1	1	1	1	2	2	2	3	3	4

Rys. 19.15. Wyznaczenie najdłuższego wspólnego podciągu dla napisów KOMPUTER i KALKULATOR

Ćwiczenie 11

Wyznacz najdłuższy wspólny podciąg dla napisów AUTOSTRADA i STRATEGIA oraz LITERATURA i MATEMATYKA.

Oto zapis algorytmu w pseudokodzie i kluczowy fragment kodu programu:

```
i ← n
j ← m
nwp ← ""
dopóki i > 0 oraz j > 0 wykonuj
    jeśli s[i-1] = t[j-1] to
        nwp ← s[i-1] + nwp
        i ← i - 1
        j ← j - 1
    w przeciwnym przypadku
        jeśli d[i][j-1] > d[i-1][j] to
            j ← j - 1
        w przeciwnym przypadku
            i ← i - 1
```

```
1.      i=n; j=m; nwp="";
2.      while (i>0 && j>0)
3.          if (s[i-1]==t[j-1])
4.          {
5.              nwp=s[i-1]+nwp;
6.              i--; j--;
7.          }
8.          else if (d[i][j-1]>d[i-1][j]) j--;
9.               else i--;
10.     cout << "Podciag: " << nwp;
```

Fragment kodu źródłowego odpowiadający za wypisanie najdłuższego wspólnego podciągu dla dwóch napisów

W linii 1 zadeklarowanemu wcześniej napisowi nwp przypisujemy pusty ciąg znaków. Jeśli sprawdzane litery obu wyrazów są równe (linia 3), napis nwp powiększamy z przodu o tę literę (linia 5) i przechodzimy do kolejnego elementu tablicy d po przekątnej (linia 6).

Ćwiczenie 12

Napisz program wypisujący najdłuższy wspólny podciąg dwóch napisów (dowolny, jeśli takich podciągów jest kilka).

Szukanie maksymalnego podciągu – złożoność obliczeniowa

Zarówno złożoność czasowa, jak i złożoność pamięciowa algorytmu wynosi $O(n \cdot m)$, gdzie n i m oznaczają długości przeszukiwanych ciągów. Wykorzystywana jest dwuwymiarowa tablica o rozmiarach $n \times m$. Aby wyznaczyć wartość w tabeli długości, trzeba ją przejrzeć raz. Wyznaczenie najdłuższego podciągu na jej podstawie ma już złożoność liniową.

Podsumowanie

- Metoda programowania dynamicznego polega na tym, że w danym kroku rozwiązania podejmuje się najkorzystniejszą decyzję na podstawie rozwiązań znalezionych wcześniej dla pośrednich danych.
- W metodzie programowania dynamicznego często wykorzystuje się tablice pomocnicze jednowymiarowe i dwuwymiarowe.
- Metoda programowania dynamicznego znajduje optymalne rozwiązania problemów: wydawania reszty, znajdowania na planszy drogi o maksymalnej sumie, pakowania plecaka czy znajdowania najdłuższego wspólnego podciągu.
- Aby zastosować metodę programowania dynamicznego do rozwiązania problemu wydawania reszty, należy zapamiętywać liczbę monet oraz rozkład poszczególnych nominałów dla wszystkich kwot pośrednich od 0 do kwoty, którą chcemy wydać.
- Zastosowanie metody programowania dynamicznego do znalezienia na planszy drogi o maksymalnej sumie polega na wyliczaniu wszystkich pośrednich sum dla pól planszy.
- Kiedy wykorzystujemy metodę programowania dynamicznego do rozwiązania ogólnego problemu plecakowego, zapamiętujemy wartość spakowanych przedmiotów oraz informację o liczbie i rodzaju pakowanych przedmiotów dla wszystkich pośrednich wag plecaka od 0 do maksymalnej wagi.
- Aby wykorzystać metodę programowania dynamicznego do znalezienia najdłuższego wspólnego podciągu, trzeba wyznaczyć maksymalne długości podciągów dla poszczególnych początkowych fragmentów wyjściowych ciągów i na tej podstawie wyznaczyć najdłuższy wspólny podciąg.

Zadania

* **1** Problem wydawania reszty metodą programowania dynamicznego można też rozwiązać, przeglądając dla kolejnych nominałów wszystkie kwoty. Program znajdujący minimalną liczbę monet potrzebnych do wydania reszty zmodyfikuj tak, aby działał zgodnie z powyższą zasadą.

* **2** Program znajdujący maksymalną sumę na planszy metodą programowania dynamicznego uzupełnij tak, aby wypisywał także drogę przejścia od lewego górnego rogu do prawego dolnego.

** **3** Napisz program rozwiązujący decyzyjny problem plecakowy metodą programowania dynamicznego.

** **4** Pewien sklep prowadzi sprzedaż towaru w jednostkach całkowitych. W zależności od tego, ile jednostek nabywca kupuje, płaci inną cenę (niekoniecznie jedna jednostka jest tańsza przy zakupie większej ilości). W sklepie znajduje się n jednostek towaru oraz cennik c, gdzie c_i oznacza cenę zakupu i jednostek, $1 \leqslant i \leqslant n$. Napisz program, który obliczy maksymalny zarobek sklepu.

** **5** Żetony są ułożone w n stosach. W każdym stosie znajduje się m_i żetonów, $1 \leqslant i \leqslant n$. Gracz zbiera cały stos (wszystkie żetony z danego stosu), ale nie może zebrać dwóch sąsiednich stosów. Napisz program, który znajdzie maksymalną liczbę żetonów, które gracz może zebrać.

** **6** Napisz program losujący dwie tablice n-elementowe liczb jednocyfrowych oraz znajdujący ich najdłuższy wspólny podciąg.

*** **7** Napisz program znajdujący dla danego ciągu liczbowego najdłuższy podciąg rosnący. Podciąg nie musi być spójny, tzn. jego elementy nie muszą występować obok siebie w ciągu źródłowym, ale muszą być w tej samej kolejności co w ciągu źródłowym.

*** **8** Dane są dwa zdania złożone z wielkich liter alfabetu łacińskiego oraz spacji. Napisz program, który znajdzie najdłuższy wspólny podciąg tych dwóch zdań składający się z wyrazów (wyraz traktujemy jako pojedynczy element ciągu). Na przykład dla zdań ALA MA KOTA I DWA PSY oraz OLA MA PSA I DWA KOTY najdłuższym wspólnym podciągiem jest MA I DWA.

20. Dziel i zwyciężaj, czyli sortujemy sprawniej

Władcy starożytnego Rzymu uważali, że łatwiej jest rządzić, kiedy poddani są skłóceni, a w społeczeństwie zarysowane są podziały. Jest wtedy zdecydowanie mniejsza szansa na to, że zwaśnione strony razem wystąpią przeciw władzy i ją pokonają. Zasada ta została nazwana „dziel i rządź" (łac. *divide et impera*). Od niej pochodzi nazwa jednej z podstawowych metod projektowania efektywnych algorytmów.

Cele lekcji

- Zastosujesz strategię „dziel i zwyciężaj" oraz rekurencję w algorytmach sortowania.
- Poznasz algorytmy sortowania szybkiego oraz sortowania przez scalanie.
- Oszacujesz złożoność czasową i pamięciową algorytmów sortowania szybkiego i sortowania przez scalanie.
- Scalisz posortowane ciągi w miejscu (bez użycia tablicy pomocniczej).

Dziel i zwyciężaj

Jednym z podstawowych sposobów rozwiązywania problemów algorytmicznych jest metoda **„dziel i zwyciężaj"** (ang. *divide and conquer*). Polega ona na tym, że problem dzieli się na podproblemy tego samego typu dla mniejszej liczby danych. Technikę tę stosowaliśmy już podczas znajdowania elementu w zbiorze uporządkowanym za pomocą **przeszukiwania binarnego**. Wówczas w kolejnych krokach przedział połowiliśmy do momentu, gdy zawierał już tylko jedną liczbę.

Przeszukiwanie binarne, s. 154–158

W tym temacie zastosujemy tę metodę w algorytmach sortowania, które będą działać zdecydowanie szybciej niż poznane wcześniej.

20.1. Sortowanie szybkie

Sortowanie szybkie

Sortowanie szybkie (ang. *quicksort*) polega na podzieleniu danych na dwa takie fragmenty, aby każdy element pierwszego fragmentu był mniejszy lub równy każdemu elementowi drugiego fragmentu. Podział ten trzeba zaplanować tak, żeby do jego przeprowadzenia wystarczyło jednokrotne przejrzenie danych.

Warto wiedzieć

Sortować można dowolne elementy, pod warunkiem, że umiemy je porównać.

Element dzielący

Dane dzielimy na fragmenty względem jednego wyróżnionego elementu, który nazywamy **elementem dzielącym**. Rysunek 20.1 pokazuje przykładowe dane wraz z wyróżnionym elementem dzielącym – w tym wypadku jest nim element środkowy.

61	43	56	10	77	51	64	78	55	36	15

Rys. 20.1. Dane z zaznaczonym elementem dzielącym

Podzielmy liczby z przykładowego zbioru na mniejsze od 51 i większe od 51. Element dzielący może trafić do dowolnego z tych fragmentów. Rysunek 20.2 przedstawia kroki prowadzące do wyznaczenia dwóch fragmentów: w pierwszym znajdą się liczby mniejsze lub równe elementowi dzielącemu, a w drugim – liczby od niego większe.

Rys. 20.2. Wybrane etapy podziału danych względem elementu dzielącego

Przeglądamy elementy od lewej strony i poszukujemy elementu większego lub równego elementowi dzielącemu, a pomijamy (pozostawiamy na miejscu) elementy od niego mniejsze. Znaleziony element przeniesiemy na prawą stronę. Następnie z prawej strony poszukujemy elementu mniejszego lub równego elementowi dzielącemu. Jeśli go znajdziemy, zamieniamy go miejscami z elementem znalezionym po lewej stronie. Takich par do zamiany poszukujemy tak długo, aż indeksy wskazujące elementy od lewej i prawej strony się miną.

Element dzielący jest jednocześnie **wartownikiem**. Zabezpiecza przed przekroczeniem zakresu tablicy, jeśli nie ma już elementu większego od wartownika przy przeglądaniu z lewej strony lub mniejszego od wartownika przy przeglądaniu z prawej strony. Rysunek 20.3 przedstawia podział danych dla powyższych liczb.

Wartownik, s. 64

Rys. 20.3. Dane podzielone na dwa fragmenty

Element dzielący może trafić do jednego z fragmentów (w przykładzie trafia do lewego) lub nie trafić do żadnego – wówczas jest już na właściwym miejscu. Oto specyfikacja problemu podziału danych:

Specyfikacja

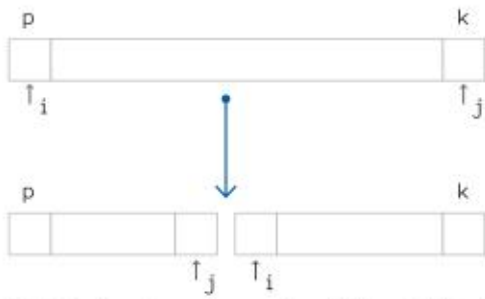
Dane: A[p..k] – fragment tablicy n losowych liczb całkowitych, $0 \leqslant p < k < n$.

Wynik: A[p..k] – fragment tablicy spełniający warunek, że każda liczba we fragmencie A[p..j] jest mniejsza lub równa każdej liczbie we fragmencie A[i..k], $p \leqslant j < i \leqslant k$.

W algorytmie sortowania szybkiego będziemy wielokrotnie dzielić coraz mniejsze fragmenty tablicy. Zapiszmy w pseudokodzie algorytm podziału fragmentu tablicy:

```
i ← p
j ← k
m ← A[(p+k) div 2] // element dzielący
dopóki i ≤ j wykonuj
    dopóki A[i] < m wykonuj i ← i + 1
    dopóki A[j] > m wykonuj j ← j - 1
    jeśli i ≤ j to
        Zamień(A[i],A[j])
        i ← i + 1
        j ← j - 1
```

Rysunek 20.4 przedstawia schemat podziału fragmentu tablicy na dwa mniejsze fragmenty. Zwróć uwagę na położenie indeksów i oraz j.



Rys. 20.4. Położenie indeksów i oraz j przed podziałem tablicy i po nim

Ćwiczenie 1

Zgodnie z algorytmem zapisanym w pseudokodzie podziel dwa fragmenty tablicy z rysunku 20.3 na s. 293.

Nie zawsze w wyniku podziału otrzymamy fragmenty składające się ze zbliżonej liczby elementów. Na przykład prawy fragment tablicy na rysunku 20.3 podzieli się na jedną liczbę i pozostałe.

Dalsze postępowanie po podziale jest już proste: otrzymane fragmenty dzielimy tak długo, aż otrzymamy fragmenty jednoelementowe. Ponieważ każdy element z lewego fragmentu jest mniejszy lub równy każdemu elementowi z prawego fragmentu, całość jest już uporządkowana od najmniejszego elementu do największego.

Warto wiedzieć

Dla różnych metod sortowania specyfikacja może być taka sama lub podobna. Celem jest zawsze uporządkowanie zbioru danych.

Kolejne fragmenty możemy sortować (dzielić) rekurencyjnie, pod warunkiem, że p < j dla lewego fragmentu (składa się on z więcej niż jednego elementu) oraz i < k dla prawego fragmentu.

Zapiszemy teraz specyfikację problemu sortowania, algorytm sortowania szybkiego w pseudokodzie oraz funkcję rekurencyjną `QuickSort`, realizującą ten algorytm w języku C++.

Specyfikacja

Dane: A[0..n–1] – tablica n liczb całkowitych.

Wynik: liczby w tablicy A spełniające warunek A[0] ⩽ A[1] ⩽ ... ⩽ A[n–1].

```
funkcja QuickSort(A[],p,k)
    i ← p
    j ← k
    m ← A[(p+k) div 2]
    dopóki i ≤ j wykonuj
        dopóki A[i] < m wykonuj i ← i + 1
        dopóki A[j] > m wykonuj j ← j - 1
        jeśli i ≤ j to
            Zamień(A[i],A[j])
            i ← i + 1
            j ← j - 1
    jeśli p < j to QuickSort(A,p,j)
    jeśli i < k to QuickSort(A,i,k)
```

Kod źródłowy funkcji sortującej liczby z wykorzystaniem algorytmu sortowania szybkiego

```
1.  void QuickSort(int A[], int p, int k)
2.  {
3.      int i=p, j=k, m=A[(p+k)/2];
4.      while (i<=j)
5.      {
6.          while (A[i]<m) i++;
7.          while (A[j]>m) j--;
8.          if (i<=j)
9.          {
10.             swap(A[i],A[j]);
11.             i++; j--;
12.         }
13.     }
14.     if (p<j) QuickSort(A,p,j);
15.     if (i<k) QuickSort(A,i,k);
16. }
```

W linii 3 określone są wartości początkowe indeksów i, j oraz wartość zmiennej m, która jest elementem dzielącym. Zewnętrzna pętla **while** (linia 4) wykonuje instrukcje do momentu minięcia się indeksów i oraz j (patrz rys. 20.4). W liniach 8–12 elementy tablicy A są zamieniane, aby znalazły się we właściwych fragmentach. Do zamiany wykorzystujemy funkcję **swap**. W liniach 14 i 15 następują rekurencyjne wywołania funkcji QuickSort, osobno dla pierwszego i drugiego fragmentu.

Oto kod źródłowy funkcji `main` programu sortującego liczby z wykorzystaniem funkcji `QuickSort`:

Kod źródłowy funkcji `main`, wykorzystującej funkcje `QuickSort` oraz `Losuj` i `Wypisz`

```
1.  int main()
2.  {
3.      int A[N];
4.      srand(time(NULL));
5.      Losuj(A);
6.      Wypisz(A);
7.      QuickSort(A,0,N-1);
8.      Wypisz(A);
9.      return 0;
10. }
```

Ćwiczenie 2

Napisz program porządkujący tablicę liczb metodą sortowania szybkiego z wykorzystaniem funkcji `QuickSort`. W programie umieść **funkcje losującą** i **wypisującą** tablicę liczb. W funkcji `main` wywołaj funkcję `QuickSort` dla wartości parametru `p` równej `0` i wartości parametru `k` równej `N-1` (od pierwszego do ostatniego elementu tablicy).

Funkcja `Losuj`, funkcja `Wypisz`, s. 152

Złożoność obliczeniowa algorytmu sortowania szybkiego

Złożoność pesymistyczna, s. 165

Oszacujmy **złożoność pesymistyczną** algorytmu. Najmniej korzystny byłby podział na jeden element i pozostałe. Jeśli występowałby za każdym razem (np. gdyby wartość elementu dzielącego była równa wartości minimum lub maksimum w każdym fragmencie), to jednokrotne przejrzenie danych w tablicy powodowałoby ustawienie na właściwej pozycji tylko jednego elementu. Wówczas działanie algorytmu nie różniłoby się od metod sortowania prostego. Pesymistyczna złożoność czasowa algorytmu sortowania szybkiego wynosi więc $O(n^2)$.

Złożoność średnia, s. 165

Prawdopodobieństwo, że tak się będzie działo dla losowych danych, szczególnie składających się z większej liczby elementów, jest bliskie zeru. **Średnia złożoność** czasowa algorytmu sortowania szybkiego wynosi $O(n \cdot \log n)$ i jest to jeden z najszybszych algorytmów sortujących.

Algorytmy sortowania prostego omówione w temacie 12 nie wymagały użycia dodatkowej pamięci zależnej od rozmiaru danych (np. wykorzystania tablicy pomocniczej), aby te dane uporządkować. Mówimy, że takie algorytmy **sortują w miejscu** (***in situ***).

Sortowanie w miejscu (*in situ*)

Czy algorytm sortowania szybkiego także sortuje w miejscu, a więc potrzebuje dodatkowej pamięci rzędu $O(1)$? Nie korzysta on z tablicy pomocniczej, ale wywołania rekurencyjne wymagają dodatkowej pamięci w zależności od rozmiaru sortowanych danych. Średnio wywołania rekurencyjne potrzebują $O(\log n)$ dodatkowej pamięci. Nie jest to złożoność stała $O(1)$, więc nie jest to algorytm sortujący w miejscu.

Warto wiedzieć

Istnieją modyfikacje przedstawionej implementacji algorytmu sortowania szybkiego, które działają w miejscu.

20.2. Modyfikacje algorytmu sortowania szybkiego

Elementem dzielącym może być dowolny element, niekoniecznie element środkowy. Gdyby wartość elementu dzielącego była medianą dzielonego fragmentu, w wyniku podziału otrzymywalibyśmy fragmenty tej samej długości (z dokładnością do 1). Wyznaczenie mediany jest jednak oddzielnym problemem algorytmicznym i jego rozwiązanie zwiększyłoby czasową złożoność obliczeniową algorytmu.

Wprowadzimy teraz modyfikację, która usprawni algorytm. W sortowaniu szybkim możemy ograniczyć wywołania rekurencyjne, jeśli zamienimy rekurencyjne sortowanie dłuższego fragmentu na iterację. Oto algorytm realizujący ten pomysł zapisany w pseudokodzie:

```
funkcja QuickSort(A[],p,k)
    dopóki p < k wykonuj
        i ← p
        j ← k
        m ← A[(p+k) div 2]
        dopóki i ≤ j wykonuj
            dopóki A[i] < m wykonuj i ← i + 1
            dopóki A[j] > m wykonuj j ← j - 1
            jeśli i ≤ j to
                Zamień(A[i],A[j])
                i ← i + 1
                j ← j - 1
        jeśli j - p < k - i to
            QuickSort(A,p,j)
            p ← i
        w przeciwnym przypadku
            QuickSort(A,i,k)
            k ← j
```

Dopóki mamy więcej niż jeden element (warunek p < k), należy kontynuować sortowanie. Po podziale tablicy sprawdzamy, który fragment jest krótszy. Jest on sortowany rekurencyjne, a dłuższy – w kolejnej iteracji zewnętrznej pętli, po wcześniejszej modyfikacji wartości p lub k.

Warto wiedzieć

Mediana to wartość zajmująca środkowe miejsce w niemalejącym ciągu wartości.

Dobra rada

Jednym z rozwiązań, które możesz zastosować przy wyznaczaniu elementu dzielącego, jest jego losowanie.

A to ciekawe

Jak powstał algorytm sortowania szybkiego?

W 1959 r. angielski informatyk Charles Antony Richard Hoare przebywał w Moskwie, gdzie studiował tłumaczenie maszynowe. Kiedy szukał sposobu na uporządkowanie alfabetyczne słów języka rosyjskiego, aby przetłumaczyć je na język angielski, wymyślił algorytm sortowania szybkiego. Dziś jest to jeden z najbardziej znanych algorytmów na świecie. Za swoje wybitne osiągnięcia Hoare został uhonorowany nagrodą Turinga oraz tytułem doktora *honoris causa* m.in. Uniwersytetu Warszawskiego.

Oto kod źródłowy zmodyfikowanej funkcji QuickSort:

Kod źródłowy zmodyfikowanej funkcji realizującej algorytm sortowania szybkiego

```
1.  void QuickSort(int A[], int p, int k)
2.  {
3.      int i, j, m;
4.      while (p<k)
5.      {
6.          i=p; j=k; m=A[(p+k)/2];
7.          while (i<=j)
8.          {
9.              while (A[i]<m) i++;
10.             while (A[j]>m) j--;
11.             if (i<=j)
12.             {
13.                 swap(A[i],A[j]);
14.                 i++; j--;
15.             }
16.         }
17.         if (j-p<k-i)
18.         {
19.             QuickSort(A,p,j); p=i;
20.         }
21.         else
22.         {
23.             QuickSort(A,i,k); k=j;
24.         }
25.     }
26. }
```

Ćwiczenie 3

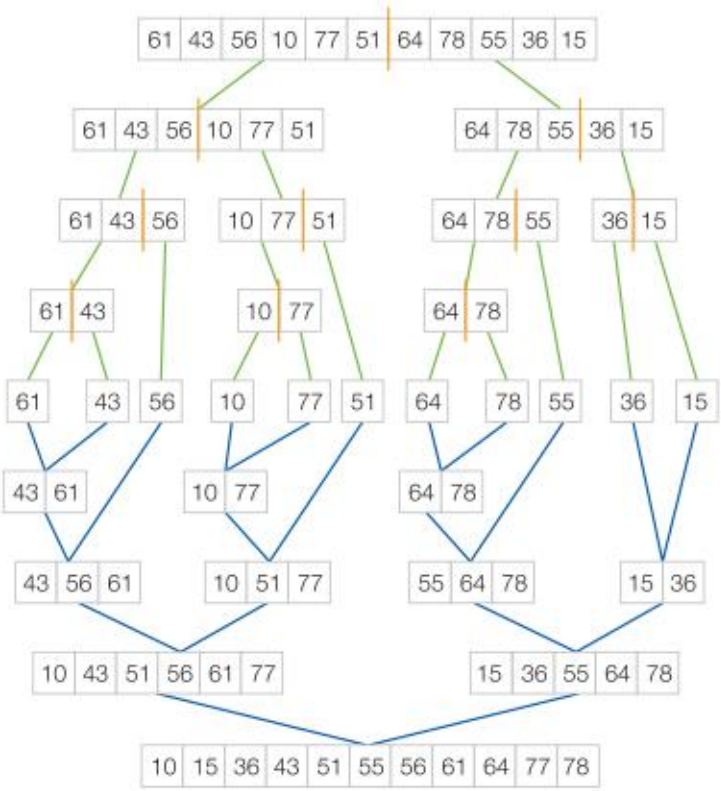
Napisz program, który wykorzysta zmodyfikowaną funkcję QuickSort. Uruchom go i sprawdź jego działanie.

20.3. Sortowanie przez scalanie

Sortowanie przez scalanie

Algorytm **sortowania przez scalanie** (ang. *merge sort*) działa na podobnej zasadzie jak algorytm sortowania szybkiego. Dane dzielimy na dwa fragmenty, ale o takiej samej liczbie elementów (z dokładnością do 1). Nie nakładamy przy tym żadnych warunków na dane podczas podziału. Po posortowaniu obydwu fragmentów danych scalamy je, czyli łączymy w całość, która będzie uporządkowana. Odpowiednie scalanie fragmentów jest kluczowym etapem algorytmu.

Rysunek 20.5 przedstawia schemat ilustrujący działanie algorytmu sortowania przez scalanie:



Rys. 20.5. Działanie algorytmu sortowania przez scalanie

Dobra rada

Pamiętaj, że tablicy o nieparzystej liczbie elementów nie da się podzielić na dwa równe fragmenty. W takiej sytuacji świadomie zdecyduj, który z fragmentów będzie o jeden element większy.

Pomarańczowe linie wyznaczają miejsca podziału. Podziały przeprowadzamy tak długo, aż otrzymamy fragmenty jednoelementowe (fragment złożony z jednego elementu jest uporządkowany). Zielone linie wskazują etap podziałów, a granatowe – etap scalania krótszych fragmentów w dłuższe, już uporządkowane.

Ćwiczenie 4

Zapisz na kartce papieru kilka losowych liczb. Na ich przykładzie pokaż działanie algorytmu sortowania przez scalanie.

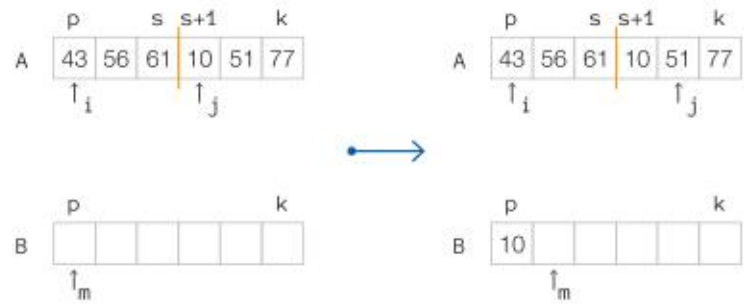
Funkcja realizująca opisany powyżej algorytm, podobnie jak w przypadku algorytmu sortowania szybkiego, będzie rekurencyjna. Jej zapis w pseudokodzie może być następujący:

```
funkcja MergeSort(A[],p,k)
    jeśli p < k to
        s ← (p+k) div 2   // indeks środka fragmentu
        MergeSort(A,p,s)  // sortuj lewy fragment
        MergeSort(A,s+1,k) // sortuj prawy fragment
        Merge(A,p,s,k)    // scalaj posortowane fragmenty
```

Scalanie posortowanych fragmentów

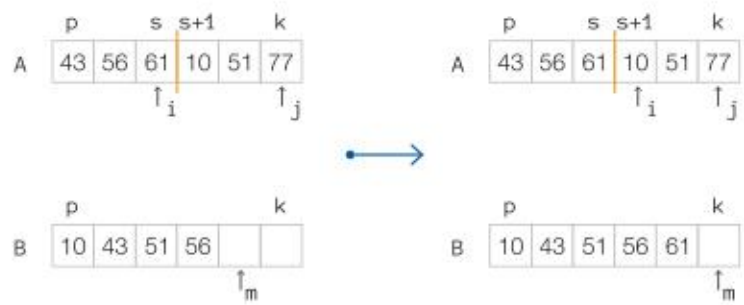
Głównym zadaniem tego algorytmu jest odpowiednie scalenie dwóch fragmentów danych. Jedną z możliwości jest scalenie dwóch fragmentów tablicy (nazwijmy ją A) w tablicy pomocniczej (nazwijmy ją B), a następnie przekopiowanie scalonego fragmentu z powrotem do sortowanej tablicy. Porównywane są kolejne elementy fragmentów i mniejszy z nich jest kopiowany do tablicy pomocniczej. Indeks skopiowanego elementu przesuwany jest do następnego elementu.

Rysunek 20.6 przedstawia pierwszy krok scalania dwóch fragmentów z wykorzystaniem tablicy pomocniczej B.



Rys. 20.6. Pierwszy krok scalania fragmentów danych – porównanie pierwszych elementów fragmentów i skopiowanie mniejszego

W ten sposób można postępować, dopóki w obydwu fragmentach są jeszcze elementy do porównywania. W pewnym momencie jeden z fragmentów zostanie w całości skopiowany do tablicy B. Wówczas do tablicy B należy skopiować brakującą część drugiego fragmentu. Ilustruje to rysunek 20.7.



Rys. 20.7. Porównanie ostatnich elementów pierwszego i drugiego fragmentu

Oto algorytm scalania zapisany w pseudokodzie:

```
i ← p
j ← s + 1
m ← p
dopóki i ≤ s oraz j ≤ k wykonuj
    jeśli A[i] < A[j] to
        B[m] ← A[i]
        i ← i + 1
```

```
    w przeciwnym przypadku
        B[m] ← A[j]
        j ← j + 1
    m ← m + 1
dopóki i ≤ s wykonuj
        B[m] ← A[i]
        i ← i + 1
        m ← m + 1
dopóki j ≤ k wykonuj
        B[m] ← A[j]
        j ← j + 1
        m ← m + 1
dla i ← p, ..., k wykonuj A[i] ← B[i]
```

Zwróć uwagę na drugą i trzecią pętlę **dopóki** w powyższym algorytmie. Wykona się tylko jedna z nich, ponieważ spełniony będzie tylko jeden z warunków: $i \leq s$ albo $j \leq k$. Wówczas zostanie przekopiowana brakująca końcówka jednego z fragmentów.

Funkcja Merge, realizująca powyższy algorytm zapisany w pseudokodzie, może wyglądać następująco:

Kod źródłowy funkcji scalającej posortowane fragmenty danych z wykorzystaniem tablicy pomocniczej

```
1.  void Merge(int A[], int p, int s, int k)
2.  {
3.      int i=p, j=s+1, m=p;
4.      int B[N];
5.      while (i<=s && j<=k)
6.      {
7.          if (A[i]<A[j])
8.          {
9.              B[m]=A[i]; i++;
10.         }
11.         else
12.         {
13.             B[m]=A[j]; j++;
14.         }
15.         m++;
16.     }
17.     while (i<=s)
18.     {
19.         B[m]=A[i]; i++; m++;
20.     }
21.     while (j<=k)
22.     {
23.         B[m]=A[j]; j++; m++;
24.     }
25.     for (i=p;i<=k;i++) A[i]=B[i];
26. }
```

Zwróć uwagę na deklarację tablicy pomocniczej B (linia 4). Przechowuje ona elementy z tablicy A od indeksu p do k, czyli k – p + 1 elementów. N jest stałą zdefiniowaną w programie, określającą rozmiary tablic A i B.

Kod źródłowy funkcji MergeSort, realizującej algorytm sortowania przez scalanie (zgodnie z pseudokodem przedstawionym na s. 299), jest następujący:

Kod źródłowy funkcji MergeSort, realizującej algorytm sortowania przez scalanie z wykorzystaniem funkcji Merge

```
1.  void MergeSort(int A[], int p, int k)
2.  {
3.      if (p<k)
4.      {
5.          int s=(p+k)/2;
6.          MergeSort(A,p,s);
7.          MergeSort(A,s+1,k);
8.          Merge(A,p,s,k);
9.      }
10. }
```

Wywołania rekurencyjne znajdują się w liniach 6 i 7. W linii 8 następuje scalanie posortowanych fragmentów.

Kod źródłowy funkcji main programu sortującego liczby z wykorzystaniem funkcji MergeSort może być następujący:

Kod źródłowy funkcji main wykorzystującej funkcje MergeSort oraz Losuj i Wypisz

```
1.  int main()
2.  {
3.      int A[N];
4.      srand(time(NULL));
5.      Losuj(A);
6.      Wypisz(A);
7.      MergeSort(A,0,N-1);
8.      Wypisz(A);
9.      return 0;
10. }
```

Ćwiczenie 5

Napisz i uruchom program porządkujący tablicę liczb metodą sortowania przez scalanie. Pamiętaj, żeby zastosować **funkcję losującą** i **funkcję wypisującą** tablicę liczb. W funkcji main wywołaj funkcję MergeSort dla wartości parametru p równej 0 i wartości parametru k równej N – 1 (od pierwszego do ostatniego elementu tablicy).

Funkcja Losuj, funkcja Wypisz, s. 152

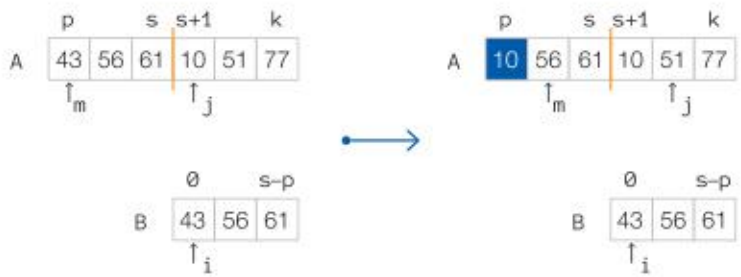
Zmniejszenie rozmiaru tablicy pomocniczej przy scalaniu

Innym pomysłem na scalanie danych jest skopiowanie w pierwszej kolejności jednego z fragmentów do pomocniczej tablicy B i scalanie fragmentów bezpośrednio w źródłowej tablicy. Zauważ, że teraz tablica B będzie dwa razy mniejsza niż w poprzedniej metodzie. Należy jednak pamiętać o przesunięciu indeksu – pierwszy kopiowany element w tablicy A ma indeks p, a w tablicy B będzie zapisany pod indeksem 0. Dlatego tablica B jest indeksowana od 0 do s–p.

Warto wiedzieć

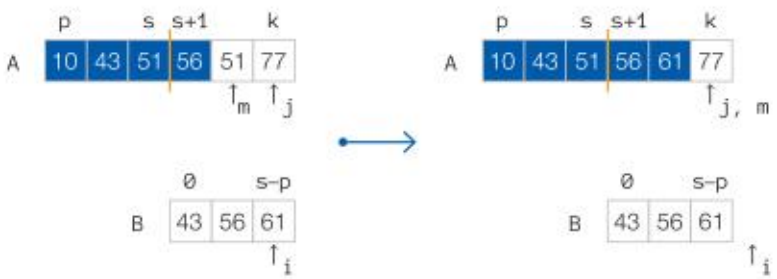
Istnieje wiele różnych algorytmów sortujących. Do popularnych należą m.in. sortowanie pozycyjne, przez kopcowanie lub sortowanie Shella.

Rysunek 20.8 przedstawia początek działania algorytmu scalania fragmentów z wykorzystaniem tablicy pomocniczej o mniejszym rozmiarze.



Rys. 20.8. Pierwszy krok scalania – porównanie pierwszych elementów scalanych fragmentów i skopiowanie mniejszego

Porównywane są wartości A[j] oraz B[i] – mniejsza z nich zostanie ustawiona na miejscu A[m]. Po skopiowaniu właściwego elementu na pozycję m w tablicy A, indeks m oraz jeden z indeksów – i albo j – są powiększane o jeden. Jeżeli jako pierwszy skończy się fragment, który został skopiowany do tablicy pomocniczej, największe elementy drugiego fragmentu są już na właściwych miejscach. Przedstawia to rysunek 20.9. Zauważ, że indeks i został zwiększony o jeden, zatem znajduje się on poza obszarem tablicy pomocniczej B.



Rys. 20.9. Wybór przedostatniego elementu tablicy A

Jeśli do wykorzystania pozostaną elementy z tablicy pomocniczej, należy je dopisać na końcu scalanego fragmentu.

Oto zapis powyższego algorytmu w pseudokodzie oraz kod źródłowy funkcji Merge, realizującej ten algorytm w języku C++.

```
dla i ← p, …, s wykonuj B[i-p] ← A[i]
i ← 0
j ← s + 1
m ← p
dopóki i ≤ s - p oraz j ≤ k wykonuj
    jeśli B[i] < A[j] to
        A[m] ← B[i]
        i ← i + 1
    w przeciwnym przypadku
        A[m] ← A[j]
        j ← j + 1
    m ← m + 1
dopóki i ≤ s - p wykonuj
    A[m] ← B[i]
    i ← i + 1
    m ← m + 1
```

Kod źródłowy funkcji scalającej posortowane fragmenty danych z wykorzystaniem zredukowanej tablicy

```
1.  void Merge(int A[], int p, int s, int k)
2.  {
3.      int i, j, m;
4.      int B[N/2+1];
5.      for (i=p;i<=s;i++) B[i-p]=A[i];
6.      i=0; j=s+1; m=p;
7.      while (i<=s-p && j<=k)
8.      {
9.          if (B[i]<A[j])
10.         {
11.             A[m]=B[i]; i++;
12.         }
13.         else
14.         {
15.             A[m]=A[j]; j++;
16.         }
17.         m++;
18.     }
19.     while (i<=s-p)
20.     {
21.         A[m]=B[i]; i++; m++;
22.     }
23. }
```

Dobra rada

Zauważ, że jeśli pierwszy fragment skończy się jako pierwszy (i będzie większe od s-p), pętla z linii 19–22 nie wykona się ani razu.

W linii 4 zadeklarowana została tablica pomocnicza B o rozmiarze równym połowie długości tablicy A, czyli N/2+1 (taka jest maksymalna długość fragmentu). N jest stałą zdefiniowaną w programie, określającą rozmiar tablicy A. W wierszu 5 tablica B jest wypełniana liczbami z pierwszego fragmentu. Pętla w liniach 7–18 porównuje elementy obydwu fragmentów i wstawia mniejszy we właściwe miejsce. Pętla w liniach 19–22 kopiuje elementy z tablicy pomocniczej, jeśli w pierwszej kolejności zostały wykorzystane wszystkie elementy drugiego fragmentu.

Ćwiczenie 6

Napisz i uruchom program porządkujący tablicę liczb metodą sortowania przez scalanie. Użyj funkcji scalającej, która korzysta z tablicy pomocniczej o zmniejszonym rozmiarze.

Zapamiętaj

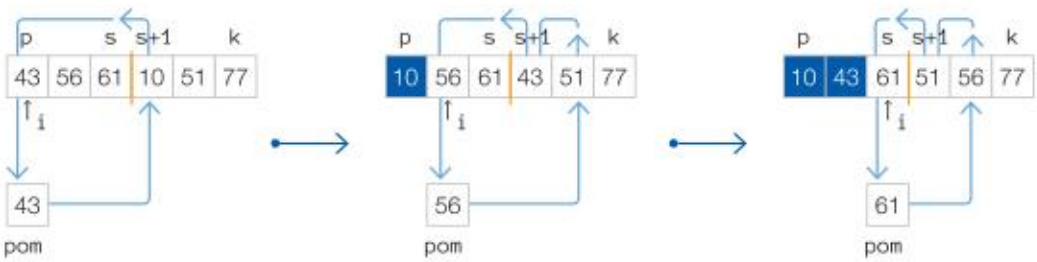
Metody sortowania szybkiego i przez scalanie wykorzystują strategię „dziel i zwyciężaj". W implementacji tych algorytmów można zastosować rekurencję.

Scalanie posortowanych fragmentów w miejscu (*in situ*)

Zastanówmy się, jak wyeliminować tablicę pomocniczą podczas scalania. Jeśli badany element lewego fragmentu jest mniejszy lub równy elementowi w prawym fragmencie, to znajduje się on na właściwym miejscu. Wystarczy przesunąć indeks w lewym fragmencie. W przeciwnym razie, gdy prawy element jest mniejszy, należy go przestawić do lewego fragmentu, a element z tego miejsca wstawić do prawego fragmentu, zachowując jego uporządkowanie. Wymaga to przesunięć analogicznych jak wykonywane w **algorytmie sortowania przez wstawianie**.

Algorytm sortowania przez wstawianie, s. 175–178

Rysunek 20.10 przedstawia kolejne kroki scalania fragmentów w miejscu.



Rys. 20.10. Kolejne kroki scalania fragmentów danych metodą *in situ*

Zapis powyższego algorytmu w pseudokodzie może być następujący:

```
dla i ← p, ..., s wykonuj
    jeśli A[s+1] < A[i] to
        pom ← A[i]
        A[i] ← A[s+1]
        j ← s + 2
        dopóki j ≤ k oraz A[j] < pom wykonuj
            A[j-1] ← A[j]
            j ← j + 1
        A[j-1] ← pom
```

Zapiszmy kod źródłowy funkcji w języku C++.

Dobra rada

Zauważ, że wystarczy przeglądać tylko pierwszy fragment i porównywać kolejne jego elementy z pierwszym elementem drugiego fragmentu. Drugi fragment cały czas pozostaje uporządkowany.

Kod źródłowy funkcji scalającej posortowane fragmenty danych z wykorzystaniem metody *in situ*

```
1.  void Merge(int A[], int p, int s, int k)
2.  {
3.      int i, j, pom;
4.      for (i=p;i<=s;i++)
5.          if (A[s+1]<A[i])
6.          {
7.              pom=A[i];
8.              A[i]=A[s+1];
9.              j=s+2;
10.             while (j<=k && A[j]<pom)
11.             {
12.                 A[j-1]=A[j];
13.                 j++;
14.             }
15.             A[j-1]=pom;
16.         }
17. }
```

Warto wiedzieć

Jeśli nie mamy żadnej wiedzy na temat danych, które chcemy posortować (np. nie wiemy, jaki jest ich zakres), to możemy je uporządkować z wykorzystaniem algorytmu o złożoności czasowej $O(n \cdot \log n)$. Nie znamy algorytmów, które pozwalałyby wykonać sortowanie szybciej.

Jeden obrót zewnętrznej pętli powoduje ustawienie jednego elementu na właściwej (i-tej) pozycji. Jeśli liczba na pozycji `s+1` jest mniejsza od liczby na pozycji `i` (warunek w linii 5), to ta ostatnia jest przechowana w zmiennej pomocniczej `pom` (linia 7), a w jej miejsce wstawiana jest liczba z pozycji `s+1` (linia 8). Wewnętrzna pętla **`while`** (linie 10–14) poszukuje w prawej połowie scalanego fragmentu miejsca, w które należy wstawić wartość ze zmiennej `pom`. Elementy mniejsze od wartości zmiennej `pom` są przesuwane o jedną pozycję w lewo. W linii 15 wartość zmiennej `pom` zostaje wstawiona w znalezione miejsce (pozycja `j–1`) tak, aby prawy fragment był cały czas uporządkowany.

Ćwiczenie 7

Uruchom i przetestuj program sortujący tablicę liczb metodą sortowania przez scalanie z wykorzystaniem scalania w miejscu.

Złożoność obliczeniowa algorytmu sortowania przez scalanie

Warto wiedzieć

Podczas tworzenia programu trzeba czasami podjąć decyzję, czy algorytm ma działać nieco wolniej, ale oszczędniej korzystać z pamięci, czy szybciej, ale z użyciem większej ilości pamięci. Algorytmy sortowania przez scalanie są przykładami takich właśnie rozwiązań.

Algorytm sortowania przez scalanie zawsze dzieli tablicę na dwa fragmenty tej samej długości. Jego złożoność czasowa (pesymistyczna i średnia) wynosi $O(n \cdot \log n)$.

Mimo że złożoność pamięciowa przedstawionych wersji algorytmu sortowania przez scalanie wynosi $O(n)$, różnią się one wymaganą ilością pamięci dodatkowej. W kolejnych wersjach jest ona odpowiednio rzędu: $O(n)$, $O(n)$ i $O(\log n)$. Chociaż w ostatniej wersji scalanie jest przeprowadzane *in situ*, algorytm sortowania przez scalanie w tej wersji nie jest algorytmem *in situ* ze względu na wywołania rekurencyjne, które wymagają dodatkowej pamięci.

Jak szybko sortują algorytmy?

Poznane wcześniej algorytmy sortowania: bąbelkowego, przez wybieranie, przez wstawianie, przez scalanie i szybkiego* znacznie różnią się szybkością działania. Aby efektywnie zmierzyć średni czas** wykonania algorytmu, można wielokrotnie posortować zbiór liczb daną metodą, a następnie wyznaczyć średnią z pomiarów czasu.

Zbiór 10 000 losowych liczb

Sortowanie bąbelkowe:	0,175 s
Sortowanie przez wybieranie:	0,091 s
Sortowanie przez wstawianie:	0,043 s
Sortowanie przez scalanie:	0,001 s
Sortowanie szybkie:	0,001 s

Zbiór 100 000 losowych liczb

Sortowanie bąbelkowe:	21,154 s
Sortowanie przez wybieranie:	8,892 s
Sortowanie przez wstawianie:	4,359 s
Sortowanie przez scalanie:	0,023 s
Sortowanie szybkie:	0,008 s

Zbiór 1 000 000 losowych liczb

Sortowanie bąbelkowe:	2200,340 s
Sortowanie przez wybieranie:	885,211 s
Sortowanie przez wstawianie:	438,081 s
Sortowanie przez scalanie:	1,803 s
Sortowanie szybkie:	0,087 s

Zbiór 100 000 liczb posortowanych odwrotnie

Sortowanie bąbelkowe:	14,905 s
Sortowanie przez wybieranie:	15,448 s
Sortowanie przez wstawianie:	8,908 s
Sortowanie przez scalanie:	0,019 s
Sortowanie szybkie:	5,836 s

Zbiór 100 000 liczb w połowie posortowanych właściwie i w połowie posortowanych odwrotnie

Sortowanie bąbelkowe:	11,625 s
Sortowanie przez wybieranie:	11,141 s
Sortowanie przez wstawianie:	4,482 s
Sortowanie przez scalanie:	0,018 s
Sortowanie szybkie:	0,016 s

Zbiór 100 000 liczb w połowie posortowanych odwrotnie i w połowie posortowanych właściwie

Sortowanie bąbelkowe:	11,651 s
Sortowanie przez wybieranie:	11,346 s
Sortowanie przez wstawianie:	4,447 s
Sortowanie przez scalanie:	0,019 s
Sortowanie szybkie:	0,013 s

* W sortowaniu szybkim jako element dzielący ustalono lewy koniec fragmentu.

** Czas ten zależy od wielu czynników, m.in. od procesora oraz wielkości pamięci dostępnej w trakcie wykonywania algorytmu. Testy przeprowadzono na komputerze z procesorem o częstotliwości 3,7 GHz i 16 GB pamięci operacyjnej.

Podsumowanie

- Metoda „dziel i zwyciężaj" polega na podziale problemu na podproblemy tego samego typu dla mniejszej liczby danych. Te podproblemy często są rozwiązywane rekurencyjnie.
- Metoda sortowania szybkiego polega na tym, że dane dzieli się na dwa fragmenty tak, aby wszystkie elementy jednego fragmentu były mniejsze lub równe każdemu elementowi drugiego fragmentu, a następnie sortuje się te fragmenty.
- Metoda sortowania przez scalanie dzieli dane na dwa równoliczne fragmenty, sortuje każdy z nich, a następnie scala posortowane fragmenty.
- Średnia złożoność czasowa algorytmów sortowania szybkiego i przez scalanie wynosi $O(n \cdot \log n)$.
- Pesymistyczna złożoność czasowa algorytmu sortowania szybkiego wynosi $O(n^2)$, a algorytmu sortowania przez scalanie – $O(n \cdot \log n)$.
- Złożoność pamięciowa algorytmów sortowania szybkiego i przez scalanie zapisanych rekurencyjnie wynosi $O(n)$.
- Algorytm sortowania szybkiego w wersji rekurencyjnej wykorzystuje dodatkowo $O(\log n)$ pamięci, zaś algorytm sortowania przez scalanie wykorzystuje dodatkowo, w zależności od wersji, $O(n)$ lub $O(\log n)$ pamięci.
- W algorytmie scalania posortowanych fragmentów wykorzystuje się tablicę pomocniczą, najczęściej o rozmiarze równym połowie sortowanej tablicy.
- Algorytm scalania posortowanych ciągów można zapisać bez wykorzystania tablicy pomocniczej. Wykonuje on wtedy jednak więcej operacji – oprócz porównywania elementów tablicy także przesunięcia elementów.

Zadania

* **1** Program porządkujący tablicę liczb metodą sortowania szybkiego zmodyfikuj tak, aby indeks elementu dzielącego był losowany.

* **2** Napisz program, który losuje dwie tablice n-elementowe, sortuje je dowolną metodą, a następnie tworzy trzecią tablicę n-elementową, składającą się z uporządkowanych n najmniejszych liczb z dwóch pierwszych tablic.

** **3** Napisz program porządkujący nierosnąco liczby w tablicy n-elementowej. Wybierz dowolny algorytm, który wykorzystuje metodę „dziel i zwyciężaj".

** **4** Metodę sortowania szybkiego można połączyć z innymi metodami sortowania i dla krótkich fragmentów danych zastosować inną, efektywniejszą metodę. Na przykład dla danych prawie posortowanych metoda sortowania przez wstawianie będzie miała złożoność liniową. Zmodyfikuj program porządkujący dane metodą sortowania szybkiego, tak aby kończył wywołania rekurencyjne, gdy długość fragmentu jest mniejsza od 5. Następnie posortuj tablicę metodą sortowania przez wstawianie.

** **5** Zapisz w tablicy wyniki symulacji n rzutów sześcienną kostką do gry. Następnie posortuj tablicę z wykorzystaniem algorytmu o liniowej złożoności czasowej.
Uwaga: Możesz zastosować metodę sortowania przez zliczanie znaną ze szkoły podstawowej.

** **6** Napisz program, który losuje dwie tablice n-elementowe, sortuje je dowolną metodą, a następnie tworzy trzecią tablicę o rozmiarze $2 \cdot n$, składającą się z uporządkowanych liczb z dwóch pierwszych tablic.

** **7** Napisz program sortujący n-elementową tablicę liczb według sumy ich cyfr. Wybierz dowolny algorytm, który wykorzystuje metodę „dziel i zwyciężaj".

*** **8** Napisz program porządkujący tablicę liczb metodą sortowania przez scalanie bez wykorzystania rekurencji. Możesz założyć, że rozmiar tablicy jest potęgą liczby 2.

*** **9** Napisz program, który wylosuje i posortuje tablicę n liczb wymiernych metodą sortowania o średniej złożoności czasowej $O(n \cdot \log n)$. Jako operację dominującą przyjmij porównanie dwóch liczb wymiernych. Liczbę wymierną możesz reprezentować jako strukturę (patrz s. 109). Napisz i wykorzystaj funkcję porównującą dwie liczby wymierne bez użycia konwersji na liczby zmiennoprzecinkowe.

*** **10** Poszukaj w internecie informacji dotyczących metody sortowania przez kopcowanie (ang. *heapsort*). Zapisz algorytm sortowania przez kopcowanie w wybranej przez siebie notacji oraz przygotuj prezentację porównującą tę metodę z algorytmami sortowania szybkiego i sortowania przez scalanie.

Sposób na zadania

Zadanie 1

W języku polskim cztery najczęściej występujące litery to samogłoski: *a*, *e*, *i* oraz *o*. Według statystyk stanowią one razem ponad 30% wystąpień wszystkich liter w polskich tekstach. Jedną z możliwości zmniejszenia objętości pliku z tekstem jest kodowanie każdej z tych samogłosek razem z literą znajdującą się przed nią lub za nią tak, jakby stanowiły jeden znak. Zakładamy, że w tekście będą występowały tylko małe litery alfabetu łacińskiego (jest ich 26) oraz spacje.

Opis sposobu kodowania:

- Do grupy znaków wyróżnionych – samogłosek: *a*, *e*, *i*, *o* – dołączono również spację, ponieważ często występuje w tekstach.
- Kody od 0 do 109 przypisano parom znaków, w których pierwszym znakiem jest znak wyróżniony, a drugim – litera spoza tej grupy (poza grupą znaków wyróżnionych są 22 litery). Łącznie otrzymujemy 110 takich par, zgodnie z poniższą tabelą.
- Kodami od 110 do 219 oznaczono pary znaków, w których drugim znakiem jest znak wyróżniony, a pierwszym – litera spoza grupy znaków wyróżnionych.
- Kody od 220 do 245 przewidziano dla 26 pojedynczych liter, których nie można zakodować jako pary (tzn. przed literą ani po niej nie znajduje się znak wyróżniony).
- Kod 246 reprezentuje spację.
- Kodujemy pierwszą możliwą parę od lewej strony, np. `aba` kodujemy jako `ab|a`, a nie `a|ba`.

Wszystkie kody mieszczą się w zakresie od 0 do 255, a więc każdy zajmuje 1 bajt pamięci.
Oto fragment tabeli kodów. Znak podkreślenia (_) oznacza spację.

Znak	Kod	Znak	Kod	Znak	Kod
ab	0	ba	110	a	220
ac	1	be	111	b	221
ad	2	bi	112	c	222
af	3	bo	113	d	223
…	…	b_	114	e	224
az	21	ca	115	f	225
eb	22	…	…	g	226
…	…	za	215	…	…
oz	87	ze	216	x	243
_b	88	zi	217	y	244
…	…	zo	218	z	245
_z	109	z_	219	_	246

Na przykład fraza `zapis elektrokardiografu` będzie zakodowana następującym ciągiem liczb:

za	pi	s_	el	ek	t	ro	ka	r	di	og	ra	f	u
215	167	184	30	29	239	178	145	237	122	70	175	225	240

Napisz program lub programy rozwiązujące poniższe zadania. Odpowiedzi zapisz w sposób wskazany przez nauczyciela.

Zadanie 1.1 (0–3)

W pliku, który otrzymasz od nauczyciela (np. *WUZ3_zad1_tekst.txt*), jest zapisanych 10 linii tekstu złożonego z małych liter alfabetu łacińskiego oraz spacji. Podaj liczbę wszystkich znaków w tekście, liczbę samogłosek: *a*, *e*, *i*, *o* (łącznie) oraz liczbę spacji.

Na przykład we frazie `zapis elektrokardiografu` występują łącznie 24 znaki, w tym 9 wyróżnionych samogłosek oraz 1 spacja.

Rozwiązanie

Tagi: kompresja danych, szyfry podstawieniowe, zliczanie elementów, kody ASCII, operacje na napisach

Krok 1

Aby rozwiązać to zadanie, wystarczy pobierać z pliku kolejne wiersze tekstu i zliczać w nich wystąpienia wyróżnionych samogłosek, spacji oraz wszystkich znaków łącznie.

Krok 2

Deklarujemy zmienne, które będą służyły do zliczania znaków w trzech kategoriach. Początkowe wartości tych zmiennych ustawiamy na zero.

Krok 3

Ponieważ dane wejściowe składają się z 10 wierszy tekstu, wykorzystamy pętlę **`for`**, która wykona się 10 razy. Analizowany tekst zawiera spacje, dlatego kolejne wiersze wczytamy przy użyciu funkcji `getline`.

Krok 4

W wewnętrznej pętli będziemy zwiększać licznik wszystkich znaków oraz sprawdzać, czy aktualny znak jest spacją lub wyróżnioną samogłoską. Zależnie od tego będziemy zwiększać odpowiedni licznik.

Krok 5

Aby stwierdzić, czy badany znak jest jedną z wyróżnionych samogłosek, sprawdzimy, czy występuje w stałej `sam4` typu **`string`**, która będzie przechowywała łańcuch znaków złożony z samogłosek: *a*, *e*, *i*, *o*. Wykorzystamy stałą **`npos`** (patrz s. 329) do sprawdzenia, czy samogłoska została odnaleziona w stałej `sam4`.

Odpowiedź: Kod źródłowy programu rozwiązującego to zadanie może być następujący:

```
1.  #include <iostream>
2.  #include <fstream>
3.  #include <string>
4.
5.  using namespace std;
6.
7.  int main()
8.  {
9.      const string sam4="aeio";
10.     int ile=0,ile1=0,ile2=0;
11.     // ile - liczba wszystkich znaków
12.     // ile1 - liczba spacji
13.     // ile2 - liczba wyróżnionych samogłosek
14.     string s;
15.     ifstream we("WUZ3_zad1_tekst.txt");
```

```
16.     for (int i=0;i<10;i++)
17.     {
18.         getline(we,s);
19.         for (int j=0;j<s.size();j++)
20.         {
21.             ile++;
22.             if (s[j]==' ') ile1++;
23.             else if (sam4.find(s[j])!=sam4.npos) ile2++;
24.         }
25.     }
26.     we.close();
27.     cout<<"Liczba znakow: "<<ile<<endl;
28.     cout<<"Liczba spacji: "<<ile1<<endl;
29.     cout<<"Liczba samoglosek aeio: "<<ile2;
30.     return 0;
31. }
```

Zadanie 1.2 (0–4)

W pliku, który otrzymasz od nauczyciela (np. *WUZ3_zad1_liczby.txt*), jest zapisanych 10 wierszy złożonych z liczb z zakresu od 0 do 255 oddzielonych spacją. Każda liczba, z wyjątkiem ostatniej w każdym wierszu, przedstawia zakodowaną parę znaków lub znak zgodnie z zasadami przedstawionymi wcześniej. Ostatnim elementem każdego wiersza jest liczba 255, oznaczająca koniec wiersza. Nie jest ona kodem żadnego znaku ani pary znaków. Odkoduj tekst z pliku.

Dla wiersza liczb:

`215 167 184 30 29 239 178 145 237 122 70 175 225 240 255`

powinien zostać zapisany następujący wiersz:

`zapis elektrokardiografu`

Rozwiązanie

Tagi: kompresja, szyfry podstawieniowe, kody ASCII, operacje na napisach

Krok 1

Aby rozwiązać zadanie, należy odczytywać kolejne liczby z pliku, przekształcać je na znaki zgodnie z tabelą kodów i budować z tych znaków odkodowany napis w zmiennej `s` typu `string`.

Krok 2

Wygodnie będzie zdefiniować dwie stałe napisowe, w których zapamiętamy znaki wyróżnione i pozostałe litery:
`spec5` – stała składająca się z 5 znaków wyróżnionych (samogłoski: *a*, *e*, *i*, *o* oraz spacja),
`pozlit` – stała składająca się z pozostałych liter w kolejności alfabetycznej.

Krok 3

Na początku odczytu każdego wiersza nadajemy zmiennej `s` wartość początkową będącą napisem pustym. Po rozkodowaniu danej liczby dodajemy do zmiennej `s` jeden lub dwa znaki.

Krok 4

Aby łatwiej było przekształcać kody na znaki, podzielimy te kody na cztery kategorie. Opiszemy również czynności konieczne do wyznaczenia odpowiedniego znaku dla wczytanej liczby `x`.

1. **Kod 246 oznacza spację.**
   Jeśli wczytana liczba `x` ma taką wartość, dodajemy do zmiennej `s` znak spacji.
2. **Kody od 220 do 245 oznaczają pojedynczą literę.**
   Jeśli wczytana liczba `x` należy do tego przedziału, dodajemy do zmiennej `s` znak o kodzie wynoszącym: `x - 220 + 'a'`. Różnica `x - 220` pozwala numerować znaki od 0 do 25. Dodanie kodu ASCII litery *a* (wynoszącego 97) pozwoli wyznaczyć właściwą literę.
3. **Kody od 110 do 219 oznaczają parę znaków: pierwszy to litera spoza grupy znaków wyróżnionych, a drugi – jeden ze znaków wyróżnionych.**
   Pierwsza litera zmienia się więc co 5 znaków (ba ma kod 110, ca 115 itd.), a drugi znak znajduje się na jednej z 5 pozycji w stałej napisowej `spec5`. Jeśli wczytana liczba `x` należy do przedziału 110–219, to w pierwszej kolejności zmniejszamy jej wartość o 110, aby kody znaków zaczynały się od 0 (napisy indeksowane są od 0). Do zmiennej `s` dodajemy znak o indeksie `x/5` ze stałej `pozlit`, a następnie znak o indeksie `x%5` ze stałej `spec5`.
4. **Kody od 0 do 109 oznaczają parę znaków: pierwszy jest znakiem wyróżnionym, a drugi – jedną z pozostałych liter.**
   Pierwsza litera zmienia się więc co 22 znaki (ab ma kod 0, eb 22 itd.), a drugi znak znajduje się na jednej z 22 pozycji w stałej napisowej `pozlit`. Jeśli wczytana liczba `x` należy do przedziału 0–109, to do zmiennej `s` dodajemy znak o indeksie `x/22` ze stałej `spec5`, a następnie znak o indeksie `x%22` ze stałej `pozlit`.

Krok 5

Po przetworzeniu wiersza liczb (napotkaniu liczby 255) wartość łańcucha `s` zapisujemy w pliku wynikowym z wykorzystaniem zmiennej plikowej `wy` typu **`ofstream`**.

Odpowiedź: Kod źródłowy programu rozwiązującego to zadanie może być następujący:

```
1.  #include <iostream>
2.  #include <fstream>
3.  #include <string>
4.
5.  using namespace std;
6.
7.  int main()
8.  {
9.      const string spec5="aeio ";
10.     const string pozlit="bcdfghjklmnpqrstuvwxyz";
11.     string s;
12.     int i,x;
13.     ifstream we("WUZ3_zad1_liczby.txt");
14.     ofstream wy("wyniki1_2.txt");
15.     for (i=0;i<10;i++)
16.     {
17.         s="";
18.         we>>x;
```

```
        while (x!=255)
        {
            if (x==246) s=s+' ';
            else if (x>=220) s=s+char(x-220+'a');
                 else if (x>=110)
                      {
                          x=x-110;
                          s=s+pozlit[x/5]+spec5[x%5];
                      }
                      else s=s+spec5[x/22]+pozlit[x%22];
            we>>x;
        }
        wy<<s<<endl;
    }
    we.close(); wy.close();
    cout<<"Wyniki w pliku wyniki1_2.txt";
    return 0;
}
```

Zadanie 1.3 (0–5)

W pliku, który otrzymasz od nauczyciela (np. *WUZ3_zad1_tekst.txt*), jest zapisanych 10 linii tekstu złożonego z małych liter alfabetu łacińskiego oraz spacji. Zakoduj ten tekst zgodnie z zasadami podanymi wcześniej.

Dla wiersza tekstu:

`zapis elektrokardiografu`

powinny zostać zapisane następujące liczby:

`215 167 184 30 29 239 178 145 237 122 70 175 225 240`

Rozwiązanie

Tagi: kompresja, szyfry podstawieniowe, kody ASCII, operacje na napisach

Krok 1

Aby rozwiązać zadanie, należy odczytywać z pliku kolejne wiersze i sprawdzać w pętli:

- czy dwa kolejne znaki należą do grupy znaków o kodach od 0 do 109,
- jeśli nie, to czy dwa kolejne znaki należą do grupy znaków o kodach od 110 do 219,
- jeśli nie, badamy pojedynczy znak: czy jest on spacją (ma kod 246). Jeśli pojedynczy znak nie jest spacją, to jest pojedynczą literą (ma kod od 220 do 245).

Na początku pętli analizujemy więc dwa kolejne znaki. Rozpoznany dwuznak lub pojedynczy znak trzeba przekształcić na odpowiedni kod liczbowy (zgodnie z tabelą kodów) i zapisać go w pliku.

Krok 2

Wiersze tekstu z pliku będziemy wczytywać za pomocą polecenia `getline`, ponieważ w tekście występują spacje. Wiersz tekstu zapiszemy w zmiennej `s` typu `string`.

Krok 3

Wygodnie będzie zdefiniować dwie stałe napisowe, w których zapamiętamy znaki wyróżnione i pozostałe litery:

`spec5` – stała składająca się z 5 znaków wyróżnionych (samogłoski: *a*, *e*, *i*, *o* oraz spacja),

`pozlit` – stała składająca się z pozostałych liter w kolejności alfabetycznej.

Krok 4

Każdy wiersz należy przeglądać w pętli od pierwszego do przedostatniego znaku, ponieważ w jednym kroku analizujemy w pierwszej kolejności dwa kolejne znaki. Wygodnie będzie zastosować pętlę **`while`**, ponieważ w jednej iteracji sprawdzamy jeden lub dwa znaki i zmienna określająca indeks badanego znaku (pierwszego z ewentualnej pary) będzie się powiększać o jeden lub o dwa. W pierwszej kolejności należy rozpatrzyć pary znaków.

Krok 5

Kody znaków wyznaczamy w następujący sposób.

W pierwszej kolejności sprawdzamy dwa kolejne znaki. Możliwe są dwie sytuacje:

1. **Pierwszy znak jest znakiem wyróżnionym, a drugi – jedną z pozostałych liter.**
   W tej sytuacji wyznaczamy indeksy tych znaków: `m` – pozycja pierwszego znaku w stałej `spec5`, `n` – pozycja drugiego znaku w stałej `pozlit`. Jeśli obydwa indeksy mieszczą się w zakresie stałych (przy sprawdzaniu korzystamy ze stałej **`npos`**, patrz s. 329), to kod pary znaków wynosi: `m*22+n` (znak wyróżniony występuje w parze z 22 pozostałymi literami).
2. **Pierwszy znak to litera spoza grupy znaków wyróżnionych, a drugi to jeden ze znaków wyróżnionych.**
   W tym przypadku wyznaczamy indeksy tych znaków: `m` – pozycja pierwszego znaku w stałej `pozlit`, `n` – pozycja drugiego znaku w stałej `spec5`. Jeśli obydwa indeksy mieszczą się w zakresie stałych, kod pary znaków wynosi: `110+m*5+n` (kody w tej grupie zaczynają się od 110 i każda z pozostałych liter występuje w parze z 5 znakami wyróżnionymi).

Jeśli nie mamy do czynienia z żadnym z dwóch powyższych przypadków, sprawdzamy pojedynczy znak. Możliwe są dwie sytuacje:

3. **Znakiem tym jest spacja.**
   Kod ma wtedy wartość 246.
4. **Znakiem tym jest pojedyncza litera.**
   Kod ma wtedy wartość różnicy kodu ASCII tej litery i kodu ASCII litery *a* (wartość ta to 97) powiększonej o 220 (kody pojedynczych znaków zaczynają się od 220).

Krok 6

Ponieważ pętla sprawdza znaki od pierwszego do przedostatniego, należy sprawdzić, czy na końcu nie ma pojedynczego znaku. Jeśli jest, należy go zakodować zgodnie z punktami 3 i 4 z kroku 5.

Odpowiedź: Kod źródłowy programu rozwiązującego to zadanie może być następujący:

```
1. #include <iostream>
2. #include <fstream>
3. #include <string>
4.
5. using namespace std;
6.
```

```
7.  int main()
8.  {
9.      const string spec5="aeio ";
10.     const string pozlit="bcdfghjklmnpqrstuvwxyz";
11.     string s;
12.     int i,j,m,n;
13.     ifstream we("WUZ3_zad1_tekst.txt");
14.     ofstream wy("wyniki1_3.txt");
15.     for (i=0;i<10;i++)
16.     {
17.         getline(we,s);
18.         j=0;
19.         while (j<s.size()-1)
20.         {
21.             m=spec5.find(s[j]);
22.             n=pozlit.find(s[j+1]);
23.             if (m!=spec5.npos && n!=pozlit.npos)
24.             {
25.                 wy<<m*22+n<<" ";
26.                 j+=2;
27.             }
28.             else
29.             {
30.                 m=pozlit.find(s[j]);
31.                 n=spec5.find(s[j+1]);
32.                 if (m!=pozlit.npos && n!=spec5.npos)
33.                 {
34.                     wy<<110+m*5+n<<" ";
35.                     j+=2;
36.                 }
37.                 else
38.                 {
39.                     if (s[j]==' ') wy<<246<<" ";
40.                     else wy<<220+s[j]-'a'<<" ";
41.                     j++;
42.                 }
43.             }
44.         }
45.         if (j<s.size())
46.         {
47.             if (s[j]==' ') wy<<246<<" ";
48.             else wy<<220+s[j]-'a'<<" ";
49.             j++;
50.         }
51.         wy<<endl;
52.     }
53.     we.close(); wy.close();
54.     cout<<"Wyniki w pliku wyniki1_3.txt";
55.     return 0;
56. }
```

Trening

ROZWIĄŻ NA KOMPUTERZE

Zadanie 1

Oto pewien system szyfrowania tekstu składającego się wyłącznie z małych liter alfabetu łacińskiego (jest ich 26). Kodowane są zgodnie z nim pary liter.
Każdej literze alfabetu łacińskiego przypisany jest kod liczbowy od 0 do 25. Każdej parze liter przypisany jest kod od 0 do 675 (liczba możliwych par wynosi $26^2 = 676$). Kod pary uzyskujemy w ten sposób, że mnożymy kod pierwszej litery przez 26 i dodajemy kod drugiej litery.
Oto fragment tabeli kodów.

Znak	Kod	Znak	Kod
a	0	aa	0
b	1	ab	1
c	2	...	...
...	...	az	25
...	...	ba	26
...	...	...	...
z	25	zz	675

Zasady szyfrowania tekstu:

- W szyfrowaniu wykorzystywany jest klucz.
- Szyfrogram tworzymy w następujący sposób: do kodu każdej kolejnej pary liter (zgodnego z tabelą kodów) dodajemy klucz szyfrowania. Otrzymana liczba wyznacza parę liter szyfrogramu.
- Operacja wyznaczania szyfrogramu dla kolejnych par jest cykliczna, tzn. jeśli otrzymana liczba przekroczy wartość 675, wyznaczana jest wartość odpowiedniej pary z początku, np. szyfrogram dla pary `zz` i klucza `1` wynosi `aa`.
- Szyfrowane są tylko litery. Jeśli liczba liter w wyrazie jest nieparzysta, ostatnia litera pozostaje bez zmiany.

Na przykład dla frazy `bezprzewodowa karta sieciowa` oraz klucza `1` szyfrogramem jest:
`bfzqsaexoeoxa kbrua sjedipwb`

Dla tej samej frazy i klucza `300` szyfrogramem jest:
`mslddnqkzraka vodha dwpqucho`

Napisz program lub programy rozwiązujące poniższe zadania. Odpowiedzi zapisz w sposób wskazany przez nauczyciela.

Zadanie 1.1 (0–3)

W pliku, który otrzymasz od nauczyciela (np. *WUZ3_zad1_wiadomosc_1.txt*), jest zapisanych 10 linii tekstu złożonego ze słów oddzielonych pojedynczą spacją. Wyznacz liczbę wszystkich słów w tekście.

Zadanie 1.2 (0–4)

W pliku, który otrzymasz od nauczyciela (np. *WUZ3_zad1_wiadomosc_2.txt*), jest zapisanych 10 linii tekstu złożonego ze słów oddzielonych pojedynczą spacją. Zaszyfruj tekst zgodnie z podanymi zasadami szyfrowania i kluczem równym 473.

Zadanie 1.3 (0–4)

W pliku, który otrzymasz od nauczyciela (np. *WUZ3_zad1_wiadomosc_3.txt*), jest zapisanych 10 linii tekstu stanowiącego szyfrogram utworzony według podanych zasad szyfrowania z kluczem równym 814. Odszyfruj tekst.

Algorytmika i programowanie

Wyobraź sobie, że...

...zaczynasz studia na kierunku automatyka i robotyka. Okazuje się, że na pierwszym roku masz zajęcia z programowania. Na szczęście w szkole poznałaś **podstawowe konstrukcje programistyczne**, **typy danych** i **proste struktury danych** oraz zdobyłaś pewne doświadczenie w programowaniu w **środowisku tekstowym**. Radzisz sobie, rozumiesz omawiane algorytmy. Potrafisz je też zaprogramować, mimo że używasz innego języka programowania niż w szkole.

...podczas studiów na kierunku informatycznym odbywasz praktyki w dużej firmie software'owej. Masz za zadanie zaprogramować moduł systemu, w którym musisz wykorzystać metodę **programowania dynamicznego** i **tablice dwuwymiarowe**. Dzięki wiedzy wyniesionej ze szkoły doskonale sobie radzisz. Dostajesz propozycję kontynuacji współpracy przy systemie po zakończeniu praktyki.

...skończyłaś studia medyczne i odbyłaś staż. Trafiasz na ofertę pracy w projekcie badawczym, w ramach którego ma powstać oprogramowanie wykorzystywane do kształcenia przyszłych lekarzy w Afryce. W czasie rozmowy kwalifikacyjnej sprawdzana jest m.in. znajomość podstawowych pojęć związanych z projektowaniem oprogramowania, takich jak **sytuacja problemowa** i **specyfikacja problemu**. Dostajesz pracę. Współpracując z programistami, możesz zmieniać świat na lepsze.

...skończyłeś studia wyższe oraz prestiżowe studia podyplomowe MBA. Odbyłeś kilka staży za granicą, masz kilkuletnie doświadczenie w kierowaniu zespołem pracowników. Ubiegasz się o stanowisko kierownika zespołu zarządzania jakością oprogramowania. Rozumiesz pojęcie **złożoności obliczeniowej** i procesy związane z tworzeniem efektywnego oprogramowania. Wiedza i doświadczenie zdobyte w szkole podczas kierowania **zespołowym projektem programistycznym** przydają się w aplikowaniu na to stanowisko. Dostajesz nową pracę.

Programowanie zespołowe

Codziennie korzystamy z wielu różnych aplikacji, na komputerach i na urządzeniach mobilnych. Często są to programy bardzo złożone – do ich stworzenia potrzebny był cały zespół programistów. Realizując ten projekt, będziecie pracować w grupie, aby na podstawie zdobytych doświadczeń stworzyć program bardziej złożony niż omawiane do tej pory.

Cel projektu

Napisanie programu rozwiązującego określony problem, poznanie zasad tworzenia aplikacji oraz pracy zespołowej podczas tworzenia oprogramowania.

To trzeba wiedzieć

Dokumentacja projektu

- Dokumentacja projektowa – opis funkcjonalności programu dla programistów, podzielenie go na mniejsze fragmenty oraz ustalenie, jak te fragmenty mają ze sobą współdziałać.
- Dokumentacja użytkownika – opis poszczególnych funkcji programu przeznaczony dla odbiorcy, użytkownika stworzonego oprogramowania. Informacja, jak korzystać z programu, oraz jego wymagania techniczne (system operacyjny, potrzebna pamięć itp.).
- Dokumentacja techniczna – szczegółowy opis, jak oprogramowanie zostało stworzone, ułatwiający w przyszłości jego modyfikację (tym razem możecie ją pominąć).

Globalne struktury danych i komunikacja

Poszczególne fragmenty programu, przygotowywane przez różnych członków zespołu, warto zapisać w postaci oddzielnych funkcji. Najlepiej, jeśli funkcja odwołuje się wyłącznie do swoich parametrów oraz zmiennych lokalnych. Funkcja jest wtedy uniwersalna – nie zależy od zewnętrznych (globalnych) struktur danych. Jeśli program będzie wykorzystywał definicje globalnych struktur danych, należy dokładnie opisać je w dokumentacji projektowej.

Aplikacja tekstowa a graficzna

Programy, które do tej pory pisaliście, rozwiązując problemy algorytmiczne, były aplikacjami tekstowymi (konsolowymi). Większość proponowanych poniżej tematów projektów można zrealizować jako aplikacje tekstowe. Jeśli chcecie, aby wasz program działał w środowisku graficznym, skorzystajcie z odpowiednich bibliotek. Informacji na ten temat możecie poszukać w sieci.

Wykorzystanie zewnętrznych bibliotek lub gotowych fragmentów kodu

W programie można też wykorzystywać zewnętrzne biblioteki opracowane przez niezależnych twórców (niebędące elementami zintegrowanego środowiska programistycznego, którego używacie) lub gotowe funkcje znalezione w sieci. Należy bezwzględnie przestrzegać zasad prawa autorskiego i sprawdzić, czy z takich materiałów możecie korzystać bezpłatnie. W każdym przypadku musicie podać autora i źródło wykorzystanego kodu.

Propozycje tematów

1. Program grający z użytkownikiem w starochińską grę Nim
2. Program modyfikujący pliki typu BMP, np. zwiększający kontrast obrazu
3. Program generujący plik HTML z kodem strony, np. prezentującej opracowane dane wejściowe dotyczące określonego tematu
4. Program ukrywający wiadomość w pliku BMP (steganografia)
5. Program zamieniający tekst na kod Morse'a i odwrotnie
6. Gra komputerowa według własnego pomysłu

Etapy projektu

1. Wybierzcie i uzgodnijcie z nauczycielem temat projektu. Możecie wykorzystać jedną z powyższych propozycji.
2. Przygotujcie dokumentację projektową programu. Podzielcie problem na mniejsze części i ustalcie sposób komunikacji pomiędzy modułami realizującymi poszczególne podproblemy.
3. Wybierzcie narzędzia, które wykorzystacie do realizacji projektu: język programowania, narzędzia do komunikacji pomiędzy członkami zespołu oraz sposób oznaczania kolejnych wersji oprogramowania.
4. Ustalcie zasady testowania poszczególnych fragmentów programu.
5. Podzielcie się rolami. Ustalcie, kto jest odpowiedzialny za pisanie poszczególnych fragmentów (modułów) programu, testowanie, przygotowanie dokumentacji użytkownika.
6. Napiszcie i przetestujcie programy rozwiązujące określone podproblemy.
7. Połączcie utworzone fragmenty programu (podprogramy) w całość.
8. Przetestujcie działanie całego programu.
9. Przygotujcie dokumentację użytkownika programu.
10. Zaprezentujcie efekt swojej pracy przed całą klasą.

Przykładowe narzędzia

Code::Blocks, **Dev-C++**, **IDLE** (Python), **PyCharm** (Python) – środowiska programistyczne. Wybierzcie środowisko, w którym powstaną poszczególne części kodu źródłowego.

Allegro (C++), **Pygame** (Python) – dodatkowe biblioteki, które można wykorzystać do tworzenia gier.

Edytor tekstu – będzie potrzebny do sporządzenia dokumentacji projektu.

Microsoft Office 365 (Teams), **Slack**, **Dokumenty Google** – oprogramowanie do komunikacji i pracy grupowej.

Dodatki

1. Sposoby zapisu algorytmów

Algorytm to opis postępowania prowadzącego do rozwiązania problemu. Istotą algorytmu jest jego uniwersalność, czyli to, że służy do rozwiązywania wielu zadań tego samego rodzaju. Na przykład szkolny algorytm mnożenia pisemnego możesz zastosować dla dowolnych dwóch liczb, np. 21 · 57, 123 · 87. Algorytm musi być precyzyjny, choć może mieć różną formę.

Opis słowny

Algorytmy mogą być opisane słownie. Przykładem może być instrukcja obsługi kuchennego minutnika: „Wykonać pełen obrót wskaźnika aż do pozycji 0. Następnie ustawić wskaźnik na wymaganą liczbę minut".

Lista kroków

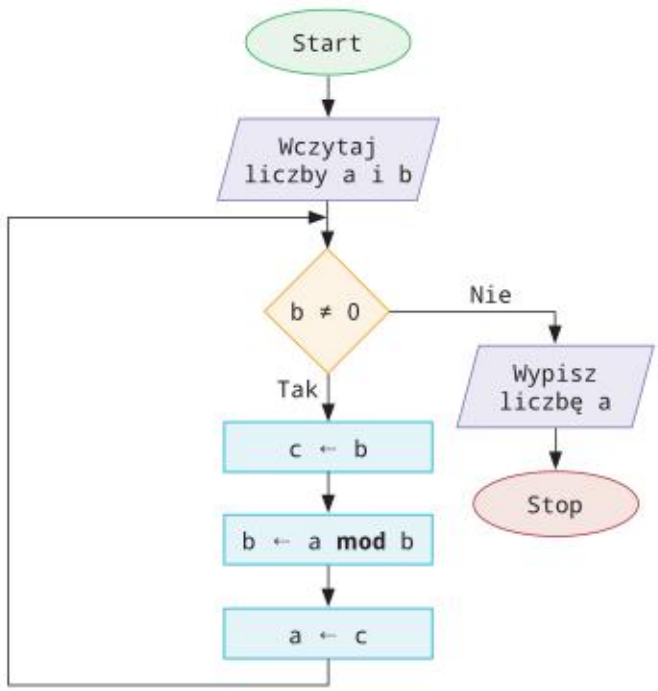
Lista kroków algorytmu to opis postępowania podzielony na poszczególne kroki, które mogą się odwoływać do siebie nawzajem.

Oto lista kroków algorytmu zaplatania warkocza klasycznego:

1. Rozdziel włosy na trzy pasma.
2. Dopóki nie spleciesz warkocza po końcówki włosów, wykonuj krok 3 i krok 4.
   3. Przełóż lewe pasmo nad środkowym. Teraz to pasmo staje się środkowym.
   4. Przełóż prawe pasmo nad środkowym. Teraz to pasmo staje się środkowym.

Schemat blokowy (sieć działań)

Schemat blokowy algorytmu jest zbudowany z umownych kształtów oraz strzałek pokazujących kierunek wykonywania operacji. Oto przykład schematu blokowego algorytmu Euklidesa, obliczającego największy wspólny dzielnik.



Pseudokod (pseudojęzyk)

Pseudokod to zapis przypominający kod języka programowania, ale pozbawiony szczegółów związanych z implementacją w języku programowania. Przedstawienie algorytmu w postaci pseudokodu ułatwia później zapisanie tego algorytmu w języku programowania. Nie istnieje jedna, powszechnie przyjęta konwencja zapisu pseudokodu. Podstawową zasadą jest jednak wybór intuicyjnych oznaczeń i konsekwentne ich wykorzystanie.
Oto pseudokod algorytmu wyznaczania części całkowitej $\sqrt{n}$ dla dodatniej liczby całkowitej n:

```
wczytaj n
p ← 1
dopóki (p + 1) · (p + 1) ≤ n wykonuj p ← p + 1
wypisz p
```

W powyższym zapisie strzałka (←) oznacza instrukcję przypisania wartości do zmiennej. Symbole +, ·, ≤ mają takie samo znaczenie jak w matematyce.

Kod źródłowy

Kod źródłowy to sformalizowany zapis algorytmu zgodny z regułami wybranego języka programowania. Kody źródłowe zazwyczaj zapisuje się w plikach tekstowych. Oto przykładowy kod źródłowy algorytmu obliczania całkowitego pierwiastka kwadratowego z liczby n, zapisany w języku C++.

```
1.  int main()
2.  {
3.      int n; cin>>n;
4.      int p = 1;
5.      while ((p + 1)*(p + 1) <= n)
6.      {
7.          p = p + 1;
8.      }
9.      cout<<p<<endl;
10.     return 0;
11. }
```

Języki programowania, podobnie jak języki naturalne, mają słownictwo podstawowe, czyli **słowa kluczowe**, oraz **składnię**, czyli reguły tworzenia wypowiedzi.

Język programowania służy do porozumiewania się człowieka z komputerem. W tej komunikacji pośredniczą **translatory**, czyli programy tłumaczące, gdyż docelowo kod źródłowy musi być przetłumaczony na **kod maszynowy**, zapisany binarnie, właściwy dla architektury procesora (CPU) i systemu operacyjnego.

2. Logika matematyczna, operatory logiczne i matematyczne

Zdania logiczne

Zdaniem logicznym jest każde zdanie, któremu możemy przyporządkować jednoznacznie wartość logiczną, czyli jedną z dwóch możliwych ocen: prawdę (ang. *true*) albo fałsz (ang. *false*).

Przykłady zdań logicznych:

- Liczba 8 jest liczbą pierwszą. (zdanie fałszywe)
- $2 + 2 < 5$ (zdanie prawdziwe)
- $2 + 8 = 20$ (zdanie fałszywe)
- Liczba parzysta jest podzielna przez 2. (zdanie prawdziwe)

Tradycyjnie przyjęło się, że prawdzie odpowiada wartość 1, a fałszowi – wartość 0.

Formy zdaniowe i zdania złożone

Formą zdaniową jest każde zdanie, które zawiera zmienne i które staje się zdaniem logicznym z chwilą zastąpienia zmiennych konkretnymi wartościami. Formie zdaniowej nie można więc jednoznacznie przypisać wartości logicznej, gdyż ta wartość zależy od wyboru wartości dla zmiennych.

Przykłady form zdaniowych:

- Liczba x jest podzielna przez 3. (np. dla $x = 6$ zdanie jest prawdziwe, a dla $x = 7$ – fałszywe)
- $y < 4$ (np. dla $y = 0$ zdanie jest prawdziwe, a dla $y = 4$ – fałszywe)
- Liczba z jest liczbą złożoną. (np. dla $z = 5$ zdanie jest fałszywe, a dla $z = 12$ – prawdziwe)

Zdania logiczne łączymy w zdania złożone, a formy zdaniowe – w formy zdaniowe złożone. Najczęściej używane konstrukcje to: zaprzeczenie (negacja), alternatywa oraz koniunkcja.

Zaprzeczenie (negacja)

Zaprzeczenie to zdanie logiczne, które ma postać: *nieprawda, że p*, gdzie p jest zdaniem logicznym. Negacją może być też forma zdaniowa o postaci *nieprawda, że* $p(x)$, gdzie $p(x)$ jest formą zdaniową. Zaprzeczenie zdania p zapisuje się często jako $\sim p$.

p	$\sim p$
$x < 4$	$x \geqslant 4$
$2 + 2 = 5$	$2 + 2 \neq 5$

Alternatywa

Alternatywą jest połączenie zdań p i q spójnikiem „lub", które ma postać: p *lub* q. Może to być także połączenie form zdaniowych $p(x)$ i $q(x)$ w postaci: $p(x)$ *lub* $q(x)$.

p	q	p *lub* q
$x < 4$	$x > 9$	$x < 4$ lub $x > 9$
y jest liczbą parzystą	z jest liczbą nieparzystą	y jest liczbą parzystą lub z jest liczbą nieparzystą

Alternatywa jest zdaniem fałszywym jedynie wtedy, gdy tworzące ją zdania logiczne są jednocześnie fałszywe.

Koniunkcja

Koniunkcja to połączenie zdań p i q spójnikiem „i", które ma postać: $p \ i \ q$. Może to być także połączenie form zdaniowych $p(x)$, $q(x)$ w postaci: $p(x) \ i \ q(x)$.

p	q	$p \ i \ q$
$x > 0$	$x < 4$	$x > 0 \ i \ x < 4$
y jest liczbą pierwszą	z jest liczbą złożoną	y jest liczbą pierwszą i z jest liczbą złożoną

Koniunkcja jest zdaniem prawdziwym jedynie wtedy, gdy tworzące ją zdania logiczne są jednocześnie prawdziwe. W pozostałych przypadkach koniunkcja jest fałszywa.

Tabela wartości logicznych

Wszystkie możliwe przypadki wartości logicznych zdań składowych i odpowiadające im wartości logiczne zdań złożonych można zestawić w tabeli:

p	q	$\sim p$	$p \ lub \ q$	$p \ i \ q$
1	0	0	1	0
0	1	1	1	0
1	1		1	1
0	0		0	0

I prawo De Morgana

Negacja koniunkcji dwóch zdań $\sim(p \ i \ q)$ jest równoważna alternatywie negacji każdego z nich $(\sim p) \ lub \ (\sim q)$. Prawo zachodzi także dla form zdaniowych $p(x)$ i $q(x)$.

p	q	$p \ i \ q$	$\sim(p \ i \ q)$	$\sim p$	$\sim q$	$(\sim p) \ lub \ (\sim q)$
$a>b$	$c>d$	$a>b \ i \ c>d$	$\sim(a>b \ i \ c>d)$	$a \leqslant b$	$c \leqslant d$	$a \leqslant b \ lub \ c \leqslant d$

Tabela wartości logicznych tego prawa wygląda następująco:

$\sim(p \ i \ q) \Leftrightarrow (\sim p) \ lub \ (\sim q)$						
p	q	$p \ i \ q$	$\sim(p \ i \ q)$	$\sim p$	$\sim q$	$(\sim p) \ lub \ (\sim q)$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

II prawo De Morgana

Negacja alternatywy dwóch zdań *~(p lub q)* jest równoważna koniunkcji negacji każdego z nich *(~p) i (~q)*. Prawo zachodzi także dla form zdaniowych *p(x)* i *q(x)*.

p	*q*	*p lub q*	*~(p lub q)*	*~p*	*~q*	*(~p) i (~q)*
$a>b$	$c>d$	$a>b \text{ lub } c>d$	$\sim(a>b \text{ lub } c>d)$	$a\leqslant b$	$c\leqslant d$	$a\leqslant b \text{ i } c\leqslant d$

Tabela wartości logicznych tego prawa wygląda następująco:

~(p lub q) ⇔ (~p) i (~q)						
p	*q*	*p lub q*	*~(p lub q)*	*~p*	*~q*	*(~p) i (~q)*
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

Operatory porównań (relacji)

Porównywanie dwóch wartości w językach programowania wykonuje się za pomocą operatorów porównań (relacji). Wynikiem ich działania są wartości `true` albo `false` w języku C++.

Zapis w matematyce	Zapis w języku C++
=	==
≠	!=
>	>
<	<
≥	>=
≤	<=

Operatory logiczne

Spójniki logiczne („i", „lub", „nieprawda, że") nazywa się operatorami logicznymi. Umożliwiają one stosowanie wyrażeń logicznych w tworzonym kodzie. Można je łączyć z operatorami porównań.

Nazwa operatora logicznego	Zapis w języku C++	Przykład w języku C++
AND (koniunkcja)	&& lub and	Dla zmiennej `x` o wartości 3 i zmiennej `y` o wartości 2 wyrażenie `x < 5 && y > 1` ma wartość `true`.
OR (alternatywa)	|| lub or	Dla zmiennej `x` o wartości 5 i zmiennej `y` o wartości 5 wyrażenie `x == 5 || y == 5` ma wartość `true`.
NOT (negacja)	! lub not	Dla zmiennej `x` o wartości 3 wyrażenie `!(x == 4)` ma wartość `true`.

Operatory matematyczne

Operator	Nazwa operatora	Wyrażenie	Wynik
+	Dodawanie	`2 + 3` `2 + 3.5`	`5` `5.5`
-	Odejmowanie	`3 - 2` `3.5 - 2`	`1` `1.5`
*	Mnożenie	`2 * 3` `2.5 * 3.5`	`6` `8.75`

/	Dzielenie	`3 / 2`
		`-3 / 2`
		`3.0 / 2`
			`1`
			`-1`
			`1.5`

%	Reszta z dzielenia (modulo)	`3 % 2`	`1`

W języku C++ nie ma operatora potęgowania. Kwadrat lub sześcian liczby najłatwiej zapisać jako iloczyn, odpowiednio: `2 * 2` lub `2 * 2 * 2`.
Do zapisywania potęg można też użyć funkcji `pow` z biblioteki `cmath`. W bibliotece tej znajduje się również funkcja `abs` do wyznaczania wartości bezwzględnej liczby.

3. Wybrane typy danych

W języku C++ każda zmienna, w zależności od tego, czy jest liczbą całkowitą, liczbą rzeczywistą, znakiem, napisem czy wartością logiczną, musi mieć jawnie przypisany odpowiedni typ danych. Określa on sposób zapisu zmiennej w pamięci komputera oraz zbiór operacji, które można wykonać na tej zmiennej.

Typy liczbowe

Wśród typów liczbowych wyróżnia się typy całkowite oraz zmiennoprzecinkowe (zmiennopozycyjne), czyli takie, które pozwalają zapisać również liczby rzeczywiste. W programowaniu wartości zapisane jako 2 i 2.0 to liczby różnych typów. Pierwsza z nich jest typu całkowitego, a druga jest typu zmiennoprzecinkowego.

Nazwa typu	Liczba bajtów	Przeznaczenie i zakres
`short`	2	Liczby całkowite ze znakiem od -2^{15} do $2^{15}-1$, czyli od –32 768 do 32 767
`long` lub `int`	4	Liczby całkowite ze znakiem od -2^{31} do $2^{31}-1$, czyli od –2 147 483 648 do 2 147 483 647
`long long`	8	Liczby całkowite ze znakiem od -2^{63} do $2^{63}-1$, czyli od –9 223 372 036 854 775 808 do 9 223 372 036 854 775 807
`unsigned short`	2	Liczby całkowite bez znaku od 0 do $2^{16}-1$, czyli od 0 do 65 535
`unsigned long` lub `unsigned int`	4	Liczby całkowite bez znaku od 0 do $2^{32}-1$, czyli od 0 do 4 294 967 295
`unsigned long long`	8	Liczby całkowite bez znaku od 0 do $2^{64}-1$, czyli od 0 do 18 446 744 073 709 551 615
`float`	4	Liczby rzeczywiste z zakresu od 1.17549e–38 do 3.40282e+38, w których liczba cyfr znaczących (dokładność) jest nie większa niż 7
`double`	8	Liczby rzeczywiste z zakresu od 2.22507e–308 do 1.79769e+308, w których liczba cyfr znaczących (dokładność) jest nie większa niż 15

Uwaga: Zapisy typu e–38 oraz e+38 oznaczają notację wykładniczą (tj. odpowiednio czynniki 10^{-38} oraz 10^{38}).

Tabela zawiera informacje o wybranych typach danych wykorzystywanych w języku C++ do zapisu liczb całkowitych i rzeczywistych dostępnych w kompilatorach 32-bitowych. Należy zwrócić uwagę, że w zależności od kompilatora C++ oraz wersji systemu (32-bitowy lub 64-bitowy) niektóre zakresy (liczba bajtów przeznaczonych do reprezentacji poszczególnych typów) mogą się różnić.
Liczbę bajtów przeznaczonych dla zmiennych danego typu można sprawdzić, korzystając z funkcji **sizeof**. Na przykład wynikiem **sizeof(int)** będzie najczęściej 4.

Typ znakowy i typ napisowy

Pojedyncze znaki można zapisywać w zmiennych typu **char**. Na przykład deklaracja `char litera='a';` to deklaracja zmiennej `litera`, której następnie przypisano literę `a`. Napisy, czyli tzw. łańcuchy znaków, zapisuje się z użyciem zmiennych typu **string**. Typ ten w sensie ścisłym nie jest typem prostym C++, ale ma charakter obiektowy, tzn. po deklaracji zmiennej uzyskujemy dostęp do wbudowanych funkcji (tzw. metod), które pozwalają łatwo wykonywać operacje na napisach.

Metody najczęściej używane dla klasy string

Nazwa metody	Parametry	Wynik	Przykład zastosowania metody dla zmiennej `s` typu `string` `s="Ala ma kota"`

find(szukane)	szukane – fraza lub znak	Pozycja pierwszego wystąpienia szukanej frazy w napisie. Jeśli dany fragment nie występuje, wynikiem funkcji jest liczba spoza zakresu `0..size()-1`	Instrukcja:
			`cout<<s.find("ma");`
			Wynik:
			`4`

substr(pozycja,ile)	`pozycja` – indeks znaku w napisie, `ile` – liczba znaków	Fragment ciągu znaków zapisanych od indeksu `pozycja`. Liczbę znaków określa parametr `ile`	Instrukcja:
			`cout<<s.substr(4,2);`
			Wynik:
			`ma`

erase(pozycja,ile)	`pozycja` – indeks znaku w napisie, `ile` – liczba znaków	Napis bez ciągu znaków o długości `ile` liczonego od indeksu `pozycja`	Instrukcje:
			`s.erase(4,3);`
			`cout<<s;`
			Wynik:
			`Ala kota`

size()	–	Długość ciągu znaków	Instrukcja:
			`cout<<s.size();`
			Wynik:
			`11`

Oprócz metod wymienionych w tabeli często korzysta się także ze stałej **npos**. Metoda **find** przyjmuje wartość równą tej stałej, gdy szukana fraza nie zostanie odnaleziona w tekście. Na przykład dla napisu `s="Ala ma kota"`, `s.find("mak")==s.npos`. Stałą **npos** można także wykorzystać do sprawdzenia, czy znak jest elementem określonego zestawu znaków, np. czy litera jest samogłoską. W poniższym przykładzie znak wczytujemy tak długo, aż podamy samogłoskę.

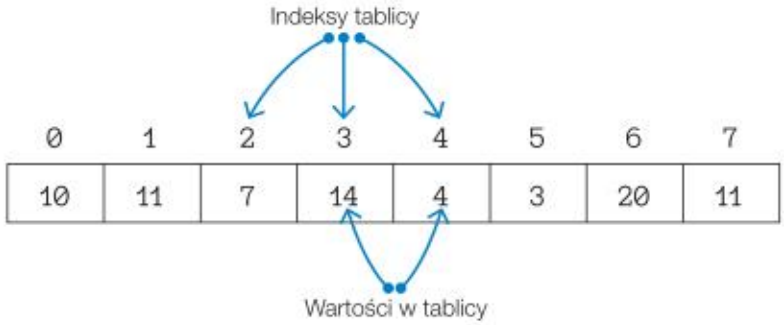
```
1. const string sam = "aeiouy";
2. char znak;
3. do
4.     cin>>znak;
5. while (sam.find(znak)!=sam.npos);
```

Typ logiczny

Do deklarowania zmiennych logicznych używa się typu **bool**, który przyjmuje wartości **true** i **false**. Są to słowa kluczowe języka C++. Wartość **false** reprezentowana jest w pamięci komputera przez liczbę 0, a wartość **true** – przez liczbę 1.

Tablice

Tablica to struktura danych grupująca wiele elementów tego samego typu. Na przykład deklaracja **int** `tablica[4];` zastępuje deklarację czterech zmiennych typu **int**.



Tablice upraszczają proces zapisywania algorytmów w języku programowania, gdyż można odwołać się do wielu wartości za pomocą jednej nazwy. Dostęp do poszczególnych elementów zapisanych w tablicy uzyskuje się za pomocą mechanizmu indeksowania, tj. podania pozycji elementu w tablicy, licząc od 0, np. `tablica[0]`, `tablica[1]`, `tablica[2]` i `tablica[3]`.

Zapis	Wyjaśnienie
`int tablica[4];`	Deklaracja tablicy czterech liczb typu **int**
`const int n = 4;` `int tablica[n];`	Deklaracja tablicy, której liczbę elementów określa stała n
`int tablica[3] = {1,2,3};`	Deklaracja tablicy trzech liczb typu **int** połączona z przypisaniem im wartości
`int tablica[] = {1,2,3};`	Deklaracja tablicy liczb typu **int** połączona z przypisaniem im wartości – rozmiar tablicy określony jest na podstawie liczby przypisanych wartości
`int tablica[4] = {0};`	Deklaracja tablicy czterech liczb typu **int** połączona z przypisaniem im wszystkim wartości 0
`int tablica[4] = {1,2};`	Deklaracja tablicy czterech liczb typu **int** połączona z przypisaniem im wartości: 1, 2, 0 i 0

Struktury

Struktura jest złożonym typem danych, który pozwala zgrupować wiele zmiennych różnych typów pod jedną nazwą, przy czym każda ze zmiennych ma własną nazwę. Oto przykład definicji struktury o nazwie uczen, grupującej wybrane dane o uczniu.

```
struct uczen
{
    string imie;
    string nazwisko;
    int numer;     // numer w dzienniku
    float srednia; // średnia ocen
    bool dysleksja;
};
```

Po zdefiniowaniu typu danych można deklarować zmienne tego typu. Poniżej przykład deklaracji zmiennej uczen1 typu uczen.

```
uczen uczen1;
```

Aby odwołać się do wartości przechowywanych w strukturze, korzystamy z notacji z kropką. Oto przykład przypisania wartości poszczególnym składowym zmiennej uczen1.

```
uczen1.nazwisko="Kowalski";
uczen1.imie="Jan";
uczen1.numer=10;
uczen1.srednia=4.5;
uczen1.dysleksja=false;
```

Można także tworzyć tablice struktur. Wówczas wykorzystujemy pętle do przeglądania elementów tablicy. Korzystając z funkcji **sort** z biblioteki algorithm, można posortować dane umieszczone w tablicy według określonego kryterium. W tym celu należy zdefiniować własną funkcję typu logicznego porównującą dwa elementy.

```
uczen uczniowie[10];

// definicja funkcji porównującej dwóch uczniów
bool porownaj (uczen uczen1, uczen uczen2)
{
    return (uczen1.srednia > uczen2.srednia);
}

sort(uczniowie,uczniowie+10,porownaj);

for (int i=0; i<10; i++)
    cout<<uczniowie[i].nazwisko<<" "<<uczniowie[i].srednia;
```

Typ pair (para)

Typ **pair** jest typem danych służącym do reprezentowania dwóch wartości (pary), które mogą być różnych typów. Deklaracja zmiennej typu **pair** jest następująca:

```
pair<typ pierwszego elementu, typ drugiego elementu> nazwa_zmiennej;
```

Przykładowa deklaracja zmiennej typu **pair** reprezentującej dwie liczby całkowite, np. współrzędne punktu na płaszczyźnie.

```
1. pair<int,int> p;
```

Typ **pair** jest typem obiektowym. Dostęp do poszczególnych wartości pary jest możliwy dzięki zmiennym obiektu first i second. Wartości elementom tej pary najłatwiej przypisać, korzystając z funkcji make_pair.

```
1. p=make_pair(3,5);
2. cout<<p.first<<" "<<p.second; // wypisanie elementów pary
```

Wartości elementom pary można nadać od razu podczas deklaracji, np.:

```
1. pair<int,int> p=make_pair(3,5);
```

lub

```
1. pair<int,int> p(3,5);
```

Element pary może być dowolnego typu, w tym również kolejną parą. Deklarując taką parę, warto pamiętać, że dwa znaki > nie mogą wtedy wystąpić obok siebie. Trzeba je oddzielić spacją.

```
1. pair<pair<int,int>,pair<int,int> > // para dwóch par
```

4. Podstawowe instrukcje sterujące

Instrukcja złożona

Blok kilku instrukcji możemy ująć w nawiasy klamrowe. Są one wówczas traktowane jak jedna instrukcja. Taką instrukcję nazywamy instrukcją złożoną.

```
{
    instrukcja_1;
    instrukcja_2;
    instrukcja_3;
    ...
    instrukcja_n;
}
```

Instrukcja warunkowa

Składnia instrukcji warunkowej jest następująca:

```
if (warunek)
    instrukcja na tak;
else
    instrukcja na nie;
```

Jeśli `warunek` przyjmuje wartość **`prawda`**, wykonywana jest `instrukcja na tak`, w przeciwnym przypadku `instrukcja na nie`.

```
1. cin>>n;
2. if (n%2==0)
3.     cout<<"Parzysta";
4. else
5.     cout<<"Nieparzysta";
```

Druga część instrukcji warunkowej (**`else`**) jest opcjonalna. Wówczas instrukcja warunkowa przyjmuje postać:

```
if (warunek) instrukcja;
```

Po słowach kluczowych **`if`** lub **`else`** może wystąpić instrukcja złożona, czyli blok instrukcji ujętych w nawiasy klamrowe.

Instrukcja powtarzania – pętla `while`

Składnia pętli **`while`** jest następująca:

```
while (warunek) instrukcja;
```

Instrukcja w pętli jest powtarzana, dopóki `warunek` przyjmuje wartość logiczną **`prawda`**. `Warunek` jest sprawdzany przed wykonaniem instrukcji. Oznacza to, że pętla nie wykona się ani razu, gdy na początku `warunek` ma wartość logiczną **`fałsz`**.

Instrukcja wykonywana w pętli może być instrukcją złożoną, czyli blokiem instrukcji ujętych w nawiasy klamrowe.

```
1. cin>>n;
2. while (n>0)
3. {
4.     cout<<n%10<<endl;
5.     n=n/10;
6. }
```

Instrukcja powtarzania – pętla for

Składnia pętli **for** jest następująca:

```
for (instrukcja inicjująca; warunek powtarzania; instrukcja modyfikująca)
    instrukcja;
```

Instrukcja inicjująca to najczęściej instrukcja przypisująca wartość początkową zmiennej, która pełni funkcję licznika. Warunek powtarzania to warunek logiczny, który powinien spełniać licznik, aby instrukcja zawarta w pętli została wykonana. Instrukcja modyfikująca to najczęściej instrukcja zmieniająca wartość licznika. Instrukcja wykonywana w pętli może być instrukcją złożoną, a więc blokiem instrukcji ujętych w nawiasy klamrowe.

```
1. cin>>n;
2. suma=0;
3. for (d=1;d<=n/2;d++)
4.     if (n%d==0) suma+=d;
5. if (suma==n) cout<<"Liczba doskonala";
```

Instrukcję **for** można zapisać za pomocą pętli **while**:

```
instrukcja inicjująca;
while (warunek powtarzania)
{
    instrukcja;
    instrukcja modyfikująca;
}
```

Instrukcja powtarzania – pętla do while

Składnia pętli **do while** jest następująca:

```
do
    instrukcja
while (warunek);
```

Najpierw wykonywana jest instrukcja, a potem sprawdzany jest warunek. Ponieważ sprawdzenie warunku następuje po wykonaniu instrukcji, pętla wykona się co najmniej raz. Kolejne powtórzenia pętli są wykonywane, dopóki warunek przyjmuje wartość logiczną **prawda**. Instrukcja wykonywana w pętli może być instrukcją złożoną, czyli blokiem instrukcji ujętych w nawiasy klamrowe.

```
1. do
2.     cin>>n;
3. while (n<=0);
```

Funkcja

Funkcja to podprogram, który odpowiada za realizację określonej części programu. Funkcje definiuje się przed definicją funkcji `main`. Definicja funkcji składa się z nagłówka funkcji i bloku instrukcji. Nagłówek zawiera typ wyniku funkcji, jej nazwę oraz listę parametrów określającą typ i nazwę każdego z parametrów.

```
typ nazwa_funkcji(typ parametr_1, typ parametr_2, ...)
{
    instrukcja_1;
    instrukcja_2;
    ...
    instrukcja_n;
}
```

Jeśli funkcja ma zwrócić wynik, w jej treści musi zostać użyta instrukcja **`return`**. Jej parametrem jest wartość zwracana przez funkcję i musi być typu określonego w nagłówku funkcji. Na przykład instrukcja **`return`** `a;` oznacza, że wynikiem funkcji będzie wartość zmiennej `a`. Instrukcja **`return`** kończy działanie funkcji, może być wielokrotnie użyta w bloku instrukcji funkcji.

```
1. int SumaCyfr(int n)
2. {
3.     int suma=0;
4.     while (n>0)
5.     {
6.         suma=suma+n%10; n=n/10;
7.     }
8.     return suma;
9. }
```

Jeśli typem funkcji w nagłówku jest **`void`**, to funkcja wykonuje tylko określone zadania i nie zwraca żadnego wyniku.

```
1. void Cyfry(int n)
2. {
3.     while (n>0)
4.     {
5.         cout<<n%10;
6.         n=n/10;
7.     }
8. }
```

Dzięki funkcjom program jest bardziej przejrzysty i łatwiej go zmodyfikować. Pozwalają one unikać powtórzeń, ponieważ tę samą funkcję można wywołać wielokrotnie dla różnych wartości parametrów.

5. Obsługa plików tekstowych

Korzystanie z plików tekstowych wymaga dopisania w kodzie źródłowym programu dyrektywy `#include <fstream>`. Komunikacja z plikami z poziomu programu w języku C++ odbywa się za pośrednictwem zmiennych plikowych. Najczęściej korzystamy ze zmiennych następujących typów:

`ifstream` – zmienne tego typu służą do obsługi plików wejściowych (do odczytu danych),
`ofstream` – zmienne tego typu służą do obsługi plików wyjściowych (do zapisu danych),
`fstream` – zmienne tego typu służą do obsługi plików przeznaczonych zarówno do odczytu, jak i zapisu danych.

Praca z plikami tekstowymi ma kilka etapów.

1. Otwarcie pliku.
2. Wczytywanie danych z pliku oraz zapisywanie danych w pliku.
3. Zamknięcie pliku.

Otwarcie pliku

Plik można otworzyć od razu podczas deklaracji zmiennej plikowej lub później, korzystając z metody open. Aby otworzyć pliki skojarzone ze zmiennymi typu **`fstream`**, podajemy parametr określający sposób otwarcia pliku.

Działanie	Zapis	Wyjaśnienie
Otwarcie pliku do odczytu	`ifstream wejscie("dane.txt");`	Otwarcie do odczytu pliku *dane.txt* skojarzonego ze zmienną plikową `wejscie` typu **`ifstream`** podczas deklaracji zmiennej.
	`fstream plik;` `plik.open("dane.txt", ios::in);`	Deklaracja zmiennej plikowej `plik` typu **`fstream`**, a następnie skojarzenie z tą zmienną pliku *dane.txt* i otwarcie go. Parametr `ios::in` (ang. *input* – wejście) sygnalizuje, że plik ma być otwarty do odczytu.
Otwarcie pliku do dopisywania	`fstream plik("dane.txt", ios::app);`	Otwarcie pliku do dopisywania informacji na końcu. Parametr `ios::app` (ang. *append* – dodać) sygnalizuje, że plik ma być otwarty do dopisywania informacji.
Otwarcie pliku do odczytu lub zapisu	`fstream plik("dane.txt", ios::in | ios::out);`	Otwarcie pliku *dane.txt* skojarzonego ze zmienną plikową `plik` typu **`fstream`**. Atrybuty otwarcia pliku można łączyć. Z pliku otwartego w ten sposób można zarówno odczytywać dane, jak i je do niego zapisywać.

Otwarcie pliku do zapisu	`ofstream wyjscie("wyniki.txt");`	Otwarcie do zapisu pliku *wyniki.txt* skojarzonego ze zmienną plikową `wyjscie` typu **`ofstream`** podczas deklaracji zmiennej.
	`fstream plik;` `plik.open("dane.txt", ios::out);`	Deklaracja zmiennej plikowej `plik` typu **`fstream`**, a następnie skojarzenie z tą zmienną i otwarcie pliku do zapisu *dane.txt*. Parametr `ios::out` (ang. *output* – wyjście) sygnalizuje, że plik ma być otwarty do zapisu.

Plik, z którego możemy odczytywać informacje lub je do niego dopisywać, musi istnieć. Jeśli podana jest tylko nazwa pliku (bez ścieżki dostępu), musi on być zapisany w tym samym folderze, w którym znajduje się plik wykonywalny programu. To, czy plik został prawidłowo otwarty, można sprawdzić, korzystając z metody `is_open`. Jej wartością jest **`prawda`**, jeśli otwarcie przebiegło pomyślnie, a jeśli nie – **`fałsz`**.

```
1. ifstream wejscie;
2. wejscie.open("dane.txt");
3. if (wejscie.is_open())
4. {
5. // wczytywanie i przetwarzanie danych z pliku
6. }
7. else cout << "Blad otwarcia pliku";
```

Do sprawdzenia poprawności otwarcia pliku możemy też wykorzystać metodę `good`.

Pliki reprezentowane przez zmienną typu **`ofstream`** lub otwarte tylko z parametrem `ios::out` są tworzone jako nowe puste pliki w tym samym folderze, w którym zapisany jest plik wykonywalny programu (o ile nie została podana ścieżka dostępu). Jeśli plik o danej nazwie już istnieje, zostanie on skasowany i zastąpiony nowym.

Wczytywanie danych z pliku oraz zapisywanie danych w pliku

Wczytywanie i zapisywanie danych odbywa się podobnie jak w przypadku korzystania ze standardowego wejścia (klawiatury) i wyjścia (ekran). Jednak zamiast polecenia **`cin`** lub **`cout`** podajemy nazwę zmiennej plikowej reprezentującej dany plik.

Działanie	Zapis	Wyjaśnienie
Wczytanie danych z pliku	`wejscie>>s;`	Wczytanie do zmiennej `s` danych z pliku reprezentowanego przez zmienną `wejscie`.
	`wejscie>>s>>t;`	Wczytanie danych z pliku do dwóch zmiennych, z których pierwsza może być np. typu **`string`**, a druga np. typu **`int`**.
Wczytanie wiersza danych z pliku	`getline(wejscie,s);`	Wczytanie wiersza danych z pliku reprezentowanego przez zmienną `wejscie` i zapisanie go w zmiennej `s`.
Zapisanie danych w pliku	`wyjscie<<s;`	Zapisanie wartości zmiennej `s` w pliku reprezentowanym przez zmienną `wyjscie`.

Wczytanie danej typu **string** z pliku z użyciem operatora >> powoduje (podobnie jak przy wczytywaniu z klawiatury) pobranie napisu do napotkania spacji lub końca wiersza. Jeśli chcemy pobrać cały wiersz danych wraz ze spacjami, należy użyć funkcji `getline`. Na przykład dla wiersza „Ala ma kota" po wykonaniu instrukcji `wejscie>>s;` zmienna `s` będzie miała wartość `"Ala"`. Natomiast po wykonaniu instrukcji `getline(wejscie,s);` zmienna `s` będzie miała wartość `"Ala ma kota"`.

Metodą przydatną podczas pracy z plikami, z których wczytujemy dane, jest metoda `eof` (ang. *end of file* – koniec pliku). Przyjmuje ona wartość **prawda**, kiedy zostanie napotkany koniec pliku (nie można już wczytywać dalej danych), a w przeciwnym wypadku – **fałsz**.

```
1. ifstream wejscie("dane.txt");
2. while (!wejscie.eof())
3. {
4. // instrukcje wczytywania i przetwarzania danych z pliku
5. }
```

Zamknięcie pliku

Zamknięcie pliku realizuje metoda `close`. Pliki zawsze powinny zostać zamknięte po zakończeniu wszystkich operacji zapisu i odczytu. Na przykład:

```
1. wyjscie.close();
```

Jeśli nie zamkniemy pliku do zapisu, część danych może zostać utracona.

6. Słowa kluczowe

Słowa kluczowe nie mogą być nazwami zmiennych, funkcji ani innych obiektów w kodzie źródłowym.

Słowo kluczowe	Opis
`bool`	Nazwa typu logicznego
`char`	Nazwa typu danych dla znaków oraz liczb całkowitych jednobajtowych
`const`	Używane przy definiowaniu stałej
`double`	Nazwa typu danych, które stanowią liczby zmiennoprzecinkowe, o tzw. podwójnej precyzji
`do`	Używane w instrukcji iteracyjnej `do while`
`else`	Używane w instrukcji warunkowej do wyodrębnienia kodu wykonywanego, gdy warunek po `if` nie jest spełniony
`false`	Stała oznaczająca wartość logiczną **fałsz**
`float`	Nazwa typu danych, które stanowią liczby zmiennoprzecinkowe
`for`	Używane w instrukcji iteracyjnej
`if`	Używane w instrukcji warunkowej
`int`	Nazwa typu danych dla liczb całkowitych
`long`	Nazwa typu danych dla liczb całkowitych
`namespace`	Używane przy określaniu tzw. przestrzeni nazw
`return`	Instrukcja, która kończy działanie funkcji i zwraca wartość do miejsca wywołania tej funkcji
`short`	Nazwa typu danych dla liczb całkowitych
`struct`	Używane przy definicji złożonego typu danych – struktury
`true`	Stała oznaczająca wartość logiczną **prawda**
`unsigned`	Używane przy deklaracji danych będących liczbami całkowitymi bez znaku
`using`	Używane przy określaniu tzw. przestrzeni nazw
`void`	Używane w nagłówku funkcji, która nie zwraca wartości
`while`	Używane w instrukcji iteracyjnej

Inne słowa kluczowe

```
asm
auto
break
case
catch
class
const_cast
continue
default
delete
dynamic_cast
enum
explicit
export
extern
friend
goto
inline
mutable
new
operator
private
protected
public
register
reinterpret_cast
signed
sizeof
static
static_cast
switch
template
this
throw
try
typedef
typeid
typename
union
virtual
volatile
wchar_t
```

Indeks

A
adres powrotu 240
algorytm 10
— Euklidesa 102
—— iteracyjny z wykorzystaniem dzielenia 106
—————— odejmowania 102
—— rekurencyjny z wykorzystaniem dzielenia 250
—————— odejmowania 250
—— rozszerzony 250
— jednoczesnego znajdowania minimum i maksimum 159
— łamania szyfru Cezara 142
— naiwny 57
— obliczania NWW 111
—— silni iteracyjny 239
——— rekurencyjny 237
— optymalnego wykorzystania sali (zachłanny) 259
— optymalny 168
— pakowania plecaka metodą programowania dynamicznego 280
————— zachłanną 265
— przeszukiwania binarnego 154
—— liniowego 153
— rozkładu liczby na czynniki pierwsze 79
— sita Eratostenesa 190
— sortowania przez prostą zamianę (bąbelkowego) 171
——— scalanie 298
——— wstawianie 175
——— wybieranie 174
—— szybkiego 292
— sprawdzający, czy liczba jest pierwsza 75, 76, 77
——— wyraz jest palindromem 61
—————— anagramem 185
— szybkiego podnoszenia do potęgi iteracyjny 57
—————— rekurencyjny 241
— szyfrowania szyfrem Cezara 117
——— kolumnowym 182
——— permutacyjnym 126
——— Vigenère'a 127
— wydawania reszty metodą programowania dynamicznego 271
————— zachłanną 255
— wyznaczania liczb Fibonacciego iteracyjny 246
————— rekurencyjny 245
— zamiany liczby dziesiętnej na dwójkową 48
—————— liczbę w innym systemie pozycyjnym 53
——— dwójkowej na dziesiętną 50
——— w wybranym systemie pozycyjnym na liczbę dziesiętną 55
— znajdowania elementu w zbiorze nieuporządkowanym 153
—————— uporządkowanym 154
—— idola w zbiorze 220, 223
—— lidera w zbiorze 212, 215
—— maksymalnej sumy w kwadracie metodą programowania dynamicznego 277
———————— zachłanną 262
—— najdłuższego spójnego podciągu niemalejącego 199
——— wspólnego podciągu 285
—— spójnego podciągu o maksymalnej sumie 207
algorytmy wykonywania działań na liczbach całkowitych w różnych systemach pozycyjnych 86, 90, 94, 98
—————— wymiernych 110
alternatywa 40
anagram 185
analiza częstości 140

B
bajt 32
biblioteka STL 214
bit 32
— najbardziej znaczący (najstarszy) 33
— najmniej znaczący 33
błąd logiczny 21
— składniowy 21

C
ciąg arytmetyczny 236
— Fibonacciego 245
czasowa złożoność obliczeniowa 164
częstotliwość próbkowania 44

D
decyzyjny problem plecakowy 265
definicja funkcji 65
— stałej 125
deklaracja tablicy 142
— zmiennej 20
digitalizacja (dyskretyzacja) 43
dziel i zwyciężaj 292
dzielniki właściwe 13

E
efektywność algorytmu 14
element dzielący 292

F
faktoryzacja liczby (rozkład liczby na czynniki pierwsze) 79
fps (liczba klatek na sekundę) 44
funkcja 64
— `abs` 113
— `getline` 67
— `main` 18
— `rand` 152
— rekurencyjna 237
— **`sort`** 214
— `srand` 153
— `time` 153
— `tolower` 63
— `toupper` 63
— typu **`void`** 113

I
IDE (zintegrowane środowisko programistyczne) 18
idol 218
indeks 38, 142
instrukcja iteracji (pętla) **`do while`** 194
——— **`for`** 24
——— **`while`** 22
— przypisania wartości zmiennej w języku C++ 22
————— pseudokodzie 17
— warunkowa **`if else`** 20
— wejścia **`cin`** 20
— wyjścia **`cout`** 20
— złożona 17
interpreter 18

K
kanał alfa 43
klatka 44
klucz 117
kod ASCII 36
— sterujący (znak sterujący) 37
— uzupełnień do 2 (U2) 33
kodek 44
kompilator 18
koniunkcja 40
konwersja (rzutowanie typu) 54

L
liczba klatek na sekundę (fps) 44
— pierwsza 73
— trójkątna 207
— wymierna 109
— złożona 73
liczby bliźniacze 81
— względnie pierwsze 109
lider 212
lista kroków 15

M
metoda 39
— `close` 123
— `eof` 124
— **`erase`** 67
— **`find`** 67
— **`size`** 39
— **`substr`** 67
— zachłanna 255
model RGB 43
— RGBA 43

N
najmniejsza wspólna wielokrotność (NWW) 111
największy wspólny dzielnik (NWD) 102
naturalny zapis binarny 33
negacja (zaprzeczenie) 40
notacja dużego O 165

O
ogólny problem plecakowy 265
operacje dominujące (elementarne) 164
operator logiczny (spójnik logiczny) 40

P
palindrom 61
pamięciowa złożoność obliczeniowa 167
parametr aktualny 65
— formalny 65
— funkcji 64
permutacja zbioru 126
podciąg 197
— niemalejący 197
— spójny 197
podstawowa struktura kodu źródłowego programu w języku C++ 19
poprawność (całkowita poprawność) algorytmu 14
porównywanie wartości w języku C++ 20
próbka 43
próbkowanie 43
problem plecakowy 265
—— decyzyjny 265
—— ogólny 265

programowanie dynamiczne 271
przekazanie parametru przez referencję 161
——— wartość 108
——— wskaźnik 152
przeszukiwanie binarne 154
— liniowe (sekwencyjne) 153
pseudokod 16

R
rekurencja (rekursja) 236
— nieskończona 238
— ogonowa 239
rozdzielczość próbkowania 44
rozkład liczby na czynniki pierwsze (faktoryzacja liczby) 79
rozpiętość zbioru 159
rozszerzony algorytm Euklidesa 250
rzutowanie typu (konwersja) 54

S
schemat blokowy 15
silnia 237
sito Eratostenesa 190
skończoność algorytmu 14
słowo kluczowe 18
sortowanie 170
— bąbelkowe (przez prostą zamianę) 171
— przez scalanie 298
—— wstawianie 175
—— wybieranie 174
— szybkie 292
— w miejscu (*in situ*) 296
specyfikacja problemu 12
spójnik logiczny (operator logiczny) 40
stała 125
— tablicowa 258
struktura 109
— danych 142
system dwójkowy (binarny) 28
— dziesiętny (decymalny) 27
— ósemkowy (oktalny) 30
— pozycyjny 27
— szesnastkowy (heksadecymalny) 30
szyfr Cezara 117
— kolumnowy 182
— monoalfabetyczny 126
— podstawieniowy 117
— polialfabetyczny 126
— przestawieniowy 182

T
tablica 142
— dwuwymiarowa 219
— jednowymiarowa 142
— kodów ASCII 37
translator 18
typ danych 20
— prosty 142
— złożony 142

U
U2 (kod uzupełnień do 2) 33
Unicode 39
UTF-8 39

W
wartownik 64
warunek początkowy 237

Z
zależność między NWD i NWW 111
zamiana wartości zmiennych 42
zaprzeczenie (negacja) 40
zdanie logiczne 40
zintegrowane środowisko programistyczne (IDE) 18
złota proporcja 247
złożoność czasowa 164
— kwadratowa 166
— liniowa 166
— logarytmiczna 166
— oczekiwana (średnia) 165
— pamięciowa 167
— pesymistyczna 165
— stała 167
— wykładnicza 167
zmienna 13
— globalna 108
— lokalna 108
— plikowa 122
znak sterujący (kod sterujący) 37

Zdjęcie na okładce: Getty Images/iStock/Getty Images Plus/MF3d.

Zdjęcia: Agencja Gazeta: Łukasz Giza s. 83; Archiwum Wydawnictwa s. 256 (banknoty); BE&W: Alamy Stock Photo – ART Collection s. 8 (Eratostenes), Dumitru Gornet s. 235 (Test Turinga), FLHC 80 s. 9 (Liber Abaci), GL Archive s. 138 (Gottfried Wilhelm Leibniz), Heritage Image Partnership Ltd s. 234 (maszyna Szczepanika), Historic Images s. 138 (Wilhelm Schickard), J Marshall – Tribaleye Images s. 11, Kumar Sriskandan s. 8 (mnożenie egipskie), Panther Media GmbH s. 70, PNC Collection s. 238, Science History Images s. 244, UtCon Collection s. 235 (Jan Łukasiewicz), IAM Ltd/WORLD HISTORY ARCHIVE/WORLD HISTORY ARCHIVE s. 74 (Euklides); Getty Images: Corbis Historical/Kim Kulish s. 57, DigitalVision Vectors s. 8 (Euklides), E+ – asbe s. 156, matejmo s. 141 (kłódka), MF3d s. 233, nicolas_ s. 199, pixelfit s. 318 (menadżer), Getty Images Plus/Photos.com s. 234 (George Boole), Hulton Archive/Heritage Images s. 105, imagenavi s. 29, iStock/Getty Images Plus – AdrianHancu s. 97, matejmo s. 7, matejmo/kaz_c s. 319, mkos83 s. 256 (banknot 500 zł), monsitj s. 127, SSPL/Science & Society Picture Library – s. 138 (kostki Napiera, kalkulator Pascala), s. 234 (maszyna Holleritha), SSPL/Science Museum s. 139 (maszyna tkacka Jacquarda), The LIFE Picture Collection/Alfred Eisenstaedt s. 281, Universal Images Group Editorial/Universal History Archive s. 234 (maszyna licząca Staffela), Universal Images Group/SVF2 s. 9 (Al-Chuwarizmi); Indigo: Alamy Stock Photo/Pictorial Press Ltd s. 139 (Ada Lovelace), s. 235 (John von Neumann), De Agostini Editore/De Agostini/Chomon & Perino s. 139 (maszyna Charlesa Babbage'a); Panthermedia: Peter Schmalfeldt s. 235 (komputer Z1); Shutterstock: Africa Studio s. 266 (notes), Alexander Raths s. 318 (grupa naukowców), alphaspirit s. 168, amiak s. 141 (tło), Anatolii Mazhora s. 144, Balkhi s. 266 (smartfon), CoolPhotoGirl s. 266 (plecak), Derek Brumby s. 266 (aparat fotograficzny), Dmitriy Rybin s. 174 (tło – orbity), DmitriyRazinkov s. 45 (fale dźwiękowe), elRoce s. 248 (huragan), Evgeniya Lystsova s. 248 (muszla), Gekkon s. 249 (Tadż Mahal – zbliżenie), Here s. 172 (woda), I'm Friday s. 266 (kartka papieru), jamesteohart s. 191, Juan Ci s. 220, Klavdiya Krinichnaya s. 172 (tło – kwadraty), s. 174 (tło – kwadraty), s. 176, KSai23 s. 258, Nutnarin Khetwong s. 266 (obiektyw), Odua Images s. 266 (laptop), Olga Popova s. 266 (rękawiczki), Onizu3d s. 266 (drewniana podłoga), photobyphotoboy s. 266 (okulary), PhotoRR s. 266 (powerbank), puhhha s. 318 (programista), SNP_SS s. 266 (zasilacz), spacezerocom s. 188, spainter_vfx s. 137, Stanislaw Tokarski s. 177, Steve Cukrov s. 266 (koszulka, kompas, butelka), Sucha Kittiwararat s. 266 (lina, mapa, garnek, portfel), Sumit.Kumar.99 s. 249 (Tadż Mahal), Sunward Art. S. 307, Titima Ongkantong s. 45 (fale), Voyagerix s. 10, Yupgi s. 256 (bankomat), Yuriy Lukin s. 45 (koncert), zffoto s. 74 (tło), Zoriana Zaitseva s. 318 (studentka robotyki); Maciej Wróbel s. 256 (monety); Wikipedia/Wikimedia Commons: s. 8 (skytale), s. 9 (Heron z Aleksandrii), Hubert Berberich (HubiB) s. 9 (szyfr podstawieniowy), link: http://www.rechenmaschinen-illustrated.com s. 139 (Abraham Jakub Stern), Photograph by Rama, Wikimedia Commons, Cc-by-sa-2.0-fr s. 297 (Charles Antony Richard Hoare).

Wydawnictwo dołożyło wszelkich starań, aby odnaleźć posiadaczy praw autorskich do wszystkich utworów zamieszczonych w podręczniku. Pozostałe osoby prosimy o kontakt z Wydawnictwem.

Wydawnictwo Nowa Era nie ponosi odpowiedzialności za treści i zasoby zamieszczone na cytowanych w tej publikacji stronach internetowych. Adresy stron internetowych zweryfikowano 14.05.2020 r.

nowa era *Twoje mocne strony*

Informatyka na czasie

Podręcznik *Informatyka na czasie 2* do zakresu rozszerzonego zawiera wiedzę dotyczącą różnych problemów algorytmicznych oraz programowania rozwiązań tych problemów w języku C++.

Różnorodne formy przekazu

Rysunki, infografiki, a także kody źródłowe oraz zrzuty ekranu pomagają zrozumieć i zapamiętać kluczowe informacje.

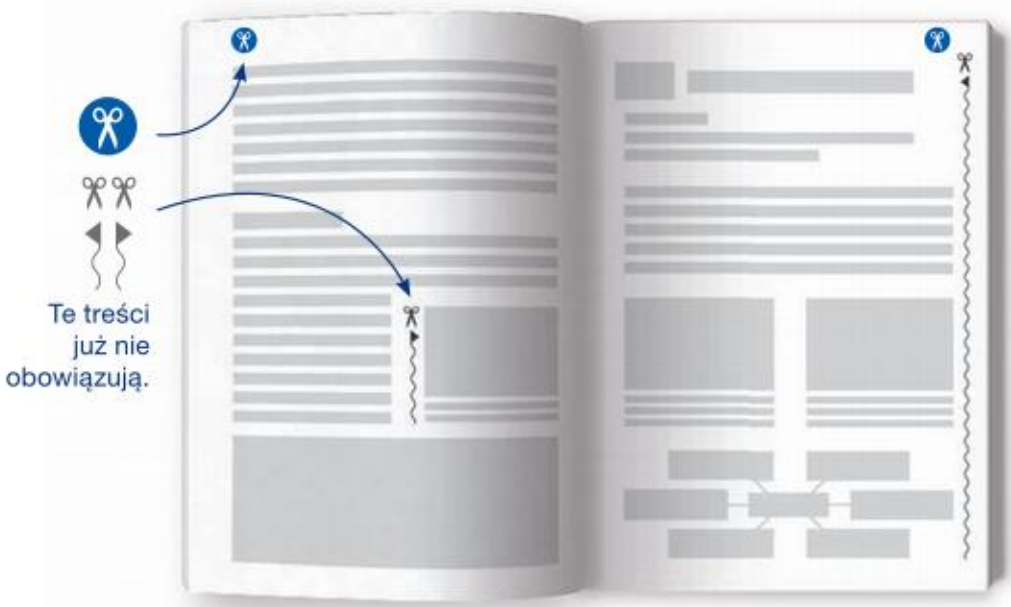
Aktywny margines

Zawiera siatkę pojęć, odsyłacze oraz praktyczne wskazówki i informacje uzupełniające wiadomości z lekcji.

Blok *Wiesz, umiesz, zdasz*

Pozwala wyćwiczyć umiejętność rozwiązywania zadań różnego typu.

W podręczniku znajdziesz oznaczenia treści, które od września 2024 r. nie są już wymagane.



Niebieska ikona na górze strony wskazuje, że na tej stronie znajdują się treści, które nie są wymagane – oznaczono je szarą ikoną i falką.

Nowa Era Sp. z o.o.

www.nowaera.pl nowaera@nowaera.pl

Centrum Kontaktu: 58 721 48 00

ISBN 978-83-267-5070-0