OPERON
Edukacja jest podróżą

ZAKRES ROZSZERZONY

Informatyka

4

Podręcznik dla **szkół ponadpodstawowych**

Joanna Śmigielska

Lech Duraj

Informatyka

Podręcznik dla **szkół ponadpodstawowych**

4

ZAKRES ROZSZERZONY

OPERON
Edukacja jest podróżą

2024

Projekt okładki: Paweł Kwacz
Redaktor prowadzący: Sebastian Przybyszewski
Redakcja językowa: zespół
Redakcja graficzna i skład: zespół
Korekta: Eliza Perkowska
Zdjęcia: Shutterstock

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 w zakresie rozszerzonym na poziomie szkoły ponadpodstawowej, na podstawie opinii rzeczoznawców: dr. Tomasza Huka, dr. Przemysława Ogonowskiego, mgr Teresy Kosyry-Cieślak.

Etap edukacyjny: III
Typ szkoły: szkoła ponadpodstawowa
Rok dopuszczenia: 2022
Numer dopuszczenia: 1048/4/2022

N7613

Wydawca:
OPERON Sp. z o.o.
81-541 Gdynia, al. Zwycięstwa 96/98
tel. centrali 58 679 00 00
e-mail: info@operon.pl
www.operon.pl

ISBN: 978-83-8197-245-1

Zgodnie z decyzją MEN od września 2024 r. obowiązuje zmieniona, uszczuplona o ok. 20% podstawa programowa. W związku z tym **w podręczniku znalazły się stosowne oznaczenia**.

Takim symbolem oznaczono treści usunięte z podstawy programowej.

Takim symbolem oznaczono treści, które zostały zmienione. Nauczyciel przedstawi uczniowi zmiany i wyjaśni, czego powinien się nauczyć.

Spis treści

II. Arkusz kalkulacyjny

Wstęp

Jak jest zbudowany nasz podręcznik

Zanim zaczniecie korzystać z podręcznika, zapoznajcie się z jego budową i przyjrzyjcie się wszystkim jego elementom. Zostały one skonstruowane tak, by ułatwić wam naukę i pomóc w błyskawicznym znalezieniu potrzebnych informacji.

Każdy rozdział podręcznika otwiera spis tematów zamieszczonych w tym rozdziale.

Bezpośrednio pod tytułem rozdziału znajdziecie moduł „Na tej lekcji", w którym są wypunktowane cele danego tematu.

W tekście pojawiają się wytłuszczenia. Dzięki nim łatwiej odnajdziecie istotne zagadnienia, pojęcia i nazwy.

10. Najkrótsze ścieżki w grafie, czyli jak komputery znajdują drogę

NA TEJ LEKCJI:

- poznasz algorytm BPS (przeszukiwania grafu wszerz), służący do znajdowania najkrótszych ścieżek w grafie;
- dowiesz się, jak znaleźć najszybsze połączenie lotnicze albo zweryfikować teorię o sześciu stopniach oddalenia.

Najkrótsze ścieżki

Jak opowiadaliśmy w rozdziale Grafy, wiele praktycznych, życiowych sytuacji modeluje się jako grafy. W szczególności wiele takich, w których grafem jest sieć połączeń drogowych lub lotniczych.

Rozważmy zatem taki graf. Oznaczmy pewne dwa z jego wierzchołków start i meta, a na polu start ustawmy pionek. Pionek może teraz poruszać się po grafie, w jednym ruchu przechodząc wzdłuż krawędzi z jednego wierzchołka do innego.

Przykładowy graf z zaznaczonymi wierzchołkami: startowym (**start**) i docelowym (**meta**)

Naturalne pytania, które można postawić, brzmią:

- Czy da się dojść od startu do mety?
- Ile co najmniej ruchów potrzeba, aby dojść do mety?
- Na które pola da się dojść ze startu, a na które nie?

Jeśli nasz graf to np. sieć połączeń kolejowych czy lotniczych, to tak naprawdę pytamy, czy da się dostać z jednego miasta do drugiego, a jeśli tak, to ile przejazdów/lotów trzeba w tym celu wykonać. Nawet w sieci społecznościowej można znaleźć pewną analogię – popularną tzw. legendą miejską jest teoria o „sześciu stopniach oddalenia", czyli stwierdzenie, że każde dwie osoby na świecie łączy łańcuch co najwyżej 6 znajomości. Wygląda to na niemożliwe do sprawdzenia, ale gdyby ktoś był w posiadaniu pełnych danych sieci społecznościowej i zastosował poniższy algorytm... jak najbardziej może przekonać się, czy to prawda.

przeszukiwanie grafu wszerz (BFS)

Algorytm, który opiszemy, potrafi znaleźć odpowiedź na wszystkie te pytania. Po polsku nazywa się przeszukiwaniem grafu wszerz, ale bardziej jest

I tak otrzymaliśmy wykres:

Powstała figura to trójkąt Sierpińskiego.

tworzenie przekształceń

- Do wygenerowania tej figury wykorzystaliśmy układ iterowanych przekształceń funkcji liniowych. Jest to matematyczna metoda rozwiązania równania, w tym wypadku dwóch zmiennych $x=f(x,y)$ i $y=g(x,y)$:

$$x' = a*x + b*y + c$$
$$y' = d*x + e*y + f$$

polegająca na wyznaczaniu kolejnych przybliżeń rozwiązania

$$x_0, y_0, x_1, y_1, \ldots, x_n, y_n$$

x_0, y_0 wybieramy dowolnie, x_1, y_1 obliczamy dzięki przekształceniu x_0, y_0, x_2, y_2 – dzięki przekształceniu x_1, y_1, a x_{n+1}, y_{n+1} – dzięki przekształceniu x_n, y_n.

Do skonstruowania trójkąta Sierpińskiego użyliśmy trzech układów równań. Każdy nowo obliczony punkt jest obrazem poprzedniego przekształconego przez jedno z trzech wylosowanych odwzorowań. Kolejne punkty pojawiają się w sposób losowy w zbiorze punktów tego układu przekształceń. Zbiór ten wykazuje cechy samopodobieństwa. Tak powstała figura jest jednym z najbardziej znanych fraktali. Fraktal to w znaczeniu potocznym obiekt samopodobny. Układ iterowanych przekształceń jest jedną z najprostszych metod generowania fraktali.

Przystępny język publikacji ułatwi wam zrozumienie treści.

Schematy i rysunki pomogą wam usystematyzować wiedzę.

Hasła na marginesach to treści obowiązkowe do zapamiętania z lekcji. Zostaną one powtórzone w podsumowaniu lekcji.

Zrzuty ekranu pełnią funkcję tutoriali, dzięki którym będziecie mogli szybko je zastosować w praktyce.

Podsumowanie przed sprawdzianem zawiera najważniejsze treści do zapamiętania z danego rozdziału. Jeśli ich nie pamiętacie, wróćcie na stronę wskazaną obok i poszukajcie hasła na marginesie.

PODSUMOWANIE PRZED SPRAWDZIANEM

1. Rekurencja raz jeszcze, czyli dziel i zwyciężaj

- s. 10 • Rekurencja w algorytmach działa, jeśli są spełnione trzy warunki: wywoływanie się zawsze na mniejszych danych, określony początek rekursji i prawidłowe odtworzenie wyniku z wyników wywołań rekurencyjnych.
- s. 11 • Szybkie potęgowanie – działające w czasie $O(\log N)$, gdzie N jest wykładnikiem potęgi do obliczenia – można łatwo zaimplementować za pomocą rekurencji.
- s. 12 • Metoda dziel i zwyciężaj działa przez podzielenie danych na dwie (czasem więcej) części, wywołanie się na każdej osobno, a potem połączenie uzyskanych wyników w jedno.

2. Sortowanie przez scalanie, czyli pierwszy efektywny algorytm sortowania

- s. 14 • Dwie posortowane tablice możemy scalić w czasie liniowym procedurą, która za każdym razem wybiera mniejszy z dwóch początkowych elementów, a następnie wykreśla go z tej tablicy.
- s. 17 • Sortowanie przez scalanie dzieli tablicę na dwie połowy, sortuje każdą z nich rekurencyjnie, po czym łączy w jedną za pomocą algorytmu scalania.
- s. 17 • Algorytm sortowania przez scalanie działa w czasie $O(n \log n)$.

3. Sortowanie szybkie, czyli jak się sortuje w praktyce

- s. 19 • Algorytm sortowania szybkiego (*Quicksort*) wybiera klucz (jeden z elementów tablicy, przeważnie losowy), a następnie dzieli tablicę tak, aby w lewej części były elementy mniejsze od klucza, a w prawej – większe. Potem wywołuje się rekurencyjnie na obu częściach.
- s. 24 • Algorytm *Quicksorta* ma złożoność pesymistyczną $O(n^2)$, ale średnią $O(n \log n)$ w wersji z losowaniem klucza.
- s. 25 • Algorytm *Quicksort* jest w praktyce najszybszym i najczęściej stosowanym z dotychczas poznanych algorytmów.

4. Specjalne elementy w tablicy, czyli największy, najmniejszy i lider

- s. 26 • Standardowo element największy lub najmniejszy w tablicy można znaleźć za pomocą $n-1$ porównań.
- s. 26 • Jeżeli jednak chcemy znaleźć jednocześnie największego i najmniejszego elementu, zamiast $2n-2$ można wykonać $3/2^*n$ porównań.
- s. 27 • Element najczęściej występujący (dominantę) można znaleźć w tablicy za pomocą sortowania w czasie $O(n \log n)$.
- s. 28 • Jeżeli szukamy lidera – elementu, który występuje więcej niż $n/2$ razy – można użyć prostego, liniowego algorytmu opartego na wykreślaniu z tablicy par różnych elementów.

Zadania do wykonania przećwiczycie w toku lekcji lub wykorzystacie do nauki w domu albo przed kartkówką czy sprawdzianem.

UWAGA! W podręczniku nie należy umieszczać pisemnych odpowiedzi do zadań.

Zastosowania kolejki

Kolejki również mają wiele praktycznych zastosowań. Na przykład: wyobraźmy sobie serwer WWW, który musi obsługiwać żądania od łączących się z nim klientów – na ogół żądania postaci „prześlij mi stronę internetową". Żądania nie przychodzą równomiernie: czasem kilka z nich przyjdzie naraz, praktycznie w jednym momencie, a czasem jest chwilowa przerwa. Typowo w takiej sytuacji serwer ma kolejkę żądań, których jeszcze nie zrealizował. Każde przychodzące żądanie trafia na koniec tej kolejki (co działa błyskawicznie), a jeśli serwer ma chwilę, obsługuje żądania z kolejki, poczynając od najwcześniejszych. Dla każdego z nich przygotowuje odpowiednią stronę, wysyła ją do klienta, po czym usuwa to żądanie z kolejki i czyta następne.

W algorytmice najbardziej klasycznym zastosowaniem kolejki jest algorytm szukania najkrótszej drogi (jego ideą poznasz w rozdziale poświęconym grafom).

Jak użyć kolejki w programie?

W języku C++ jest typ `queue<dowolny_typ>` (np. `queue<int>`). Zmienne tego typu działają tak jak opisane wyżej: mają operacje `push()`, `front()` i `pop()` (a także `empty()`). Podobna klasa `queue` występuje też w języku Python.

ZADANIA DO WYKONANIA

1. Załóżmy, że mamy do dyspozycji tablicę o stałej długości `T[1]...T[100]`. Zapisz funkcje `push()`, `pop()` i `top()` tak, aby działały jak operacje stosu oraz `push()` dokładała nowy element, `pop()` usuwała ostatni wstawiony element itd.).
2. Zaimplementuj w analogiczny sposób kolejkę – mając do dyspozycji tylko tablicę `T[1]...T[100]`, w której przechowywane będą elementy kolejki. Użyj dwóch pomocniczych zmiennych głowa i ogon – głowa będzie przechowywać miejsce, gdzie jest początek kolejki, ogon – miejsce w tablicy, gdzie jest jej ostatni element.

PODSUMOWANIE LEKCJI

- Struktura stosu pozwala na dokładanie i zdejmowanie elementów – zawsze jest zdejmowany najpóźniej włożony element.
- Struktura kolejki działa analogicznie, ale zawsze jest zdejmowany najwcześniej włożony element.
- Stos używany jest m.in. do przechowywania wywołań funkcji, a także w algorytmie obliczania wyrażeń w odwrotnej notacji polskiej.
- Kolejka jest używana m.in. do obsługi żądań serwera, a także w algorytmie szukania najkrótszej ścieżki.

Podsumowanie lekcji to esencja najważniejszych treści do zapamiętania. Jeśli ich nie pamiętacie, wróćcie na stronę wskazaną obok. Znajdziecie tam na marginesie odpowiednie hasło, a w treści lekcji – jego omówienie.

I. Algorytmika i programowanie

- Więcej o liczbach, czyli logarytm, sito Eratostenesa i schemat Hornera
- Rekursja raz jeszcze, czyli dziel i zwyciężaj
- Sortowanie przez scalanie, czyli pierwszy efektywny algorytm sortowania
- Sortowanie szybkie, czyli jak się sortuje w praktyce
- Specjalne elementy w tablicy, czyli największy, najmniejszy i lider
- Tablice dynamiczne i listy wiązane, czyli kiedy tablica nie wystarcza
- Stos i kolejka, czyli struktury proste i bardzo użyteczne
- Grafy, czyli co mają wspólnego sieć społecznościowa i paryskie metro
- Wyszukiwanie idola, czyli o tym, jak spóźniliśmy się na turniej tenisowy
- Najkrótsze ścieżki w grafie, czyli jak komputery znajdują drogę
- Programowanie strukturalne kontra obiektowe, czyli dane w centrum uwagi
- Jak rozwiązywać zadania programistyczne, czyli podstawowe porady na proste kody

1. Więcej o liczbach, czyli logarytm, sito Eratostenesa i schemat Hornera

NA TEJ LEKCJI:

- poznasz algorytm sita Eratostenesa do generowania listy wszystkich liczb pierwszych w zadanym przedziale;
- nauczysz się schematu Hornera – prostej metody obliczania wartości wielomianu.

Logarytm

Na początku przypomnisz sobie pojęcie logarytmu, które będzie nam potrzebne w dalszych rozdziałach.

Formalna definicja logarytmu jest następująca:

Logarytm o podstawie a z liczby b to taka liczba x, że $a^x = b$. Piszemy wtedy $x = \log_a b$.

Rzadko jednak będziemy podchodzić do logarytmu w ten sposób. Po pierwsze, w informatyce prawie zawsze mamy na myśli logarytm o podstawie 2 (czyli w powyższym wzorze $a = 2$). Kiedy więc mówimy „logarytm" i nie zaznaczymy inaczej, chodzi nam o logarytm dwójkowy, a zapis $\log n$ oznacza taką liczbę x, że $2^x = n$. Na przykład $\log 8 = 3$ (bo $2^3 = 8$), a $\log 1024 = 10$. A ile wynosi $\log 50$? Musi to być trochę więcej niż 5 (bo $2^5 = 32$), ale mniej niż 6 (bo $2^6 = 64$) – dokładnie jest to niecałkowita liczba 5,643... .

Uwaga!

Można spotkać inne sposoby rozumienia logarytmu – np. dla fizyka lub inżyniera zapis „log n" najczęściej oznacza logarytm o podstawie 10.

W algorytmice logarytm najczęściej pojawia się z jednego powodu: jeśli weźmiemy liczbę całkowitą n i będziemy dzielić ją przez 2 (z zaokrągleniem w dół) tak długo, jak to ma sens – czyli aż otrzymamy 0, to liczba dzieleń będzie równa $\log n + 1$, przy czym wartość logarytmu zaokrąglamy w dół. Innymi słowy, pętla:

```
x = n
dopóki x > 0
       x = x div 2
```

wykonuje $\log n + 1$ operacji. Podobna pętla występuje np. w znanym już algorytmie wyszukiwania binarnego, a jej bardziej rozbudowane i zmodyfikowane wersje zobaczymy np. w algorytmach sortowania przez scalanie i szybkiego. Zapamiętajmy zatem intuicję: **logarytm z liczby określa, ile razy możemy ją podzielić przez 2**.

Z powyższej pętli widać też, że tam, gdzie w praktyce będziemy używać logarytmu, najczęściej będziemy go zaokrąglać – w dół lub w górę. Nawet wygodniej

będzie użyć notacji O, która „ukrywa" różnice ±1, w szczególności zaokrąglanie – $O(\log n)$ może oznaczać równie dobrze dokładną wartość $\log n$, zaokrągloną w dół, w górę, a nawet podwojony logarytm.

Pierwszość, sito Eratostenesa

Liczba pierwsza, co prawdopodobnie pamiętasz z lekcji matematyki lub ze wcześniejszych etapów nauki, to taka, która dzieli się tylko przez 1 i przez samą siebie. Głównym zastosowaniem dla liczb pierwszych jest kryptografia – algorytmy szyfrowania często potrzebują dużych (takich o kilkuset cyfrach) liczb pierwszych.

Być może pamiętasz również popularny algorytm do sprawdzania, czy zadana liczba x jest pierwsza: próbujemy dzielić ją przez kolejne liczby: 2, 3, ... Jeśli przez którąś podzieli się bez reszty, oczywiście nie może być pierwsza. Wystarczy sprawdzić tylko dzielniki nie większe niż $\sqrt{x}$ – można pokazać, że jeśli x ma jakieś dzielniki, to musi mieć przynajmniej jeden ze sprawdzonych przez nas.

```
funkcja pierwsza(x)
        s = ⌊√x⌋
        dla j = 2, 3, ..., s
                jeżeli x mod j = 0
                        zwróć wynik fałsz
        zwróć wynik prawda
```

W tym rozdziale chcemy jednak rozwiązać nieco ogólniejszy problem: znaleźć *wszystkie* liczby pierwsze do pewnego N. Oczywiście zadziała poprzedni algorytm, jeśli zastosujemy go dla każdej liczby po kolei:

```
dla i = 2, 3, ..., N
        jeśli pierwsza(i):
                wypisz(i)
```

To jednak nie jest najbardziej efektywne rozwiązanie: będzie działać w czasie $O\left(N\sqrt{N}\right)$. Lepsza technika dla tego problemu nazywa się *sitem Eratostenesa* (i, jak można się domyślać, jest znana od starożytności). Polega ona na ustawieniu liczb 2, 3, ..., N i wykreślaniu z tego ciągu tych liczb, o których dowiemy się, że na pewno *nie* mogą być pierwsze. Dokładniej: dla każdej liczby w ciągu wykreślamy wszystkie liczby większe od niej, które są jej wielokrotnościami:

◀ sito Eratostenesa

- najpierw wykreślamy wszystkie liczby parzyste większe od 2, czyli 4, 6, 8, ..., samą liczbę 2 pozostawiamy;
- potem wszystkie podzielne przez 3, czyli 6, 9, 12, ...; to, że liczba 6 znowu się pojawi, nie przeszkadza – możemy wyobrazić sobie, że wykreślamy ją „po raz drugi", co nic nie zmienia;
- liczba 4 jest wykreślona (przy 2), więc wszystkie jej wielokrotności też (były również wielokrotnościami 2) – ignorujemy ją zatem zupełnie;
- kolej na liczby podzielne przez 5: 10, 15, 20, ...;
- i tak dalej, aż dojdziemy do N.

Każda napotkana *niewykreślona* liczba (najpierw 2, potem 3, 5 itd.) musi być liczbą pierwszą: skoro nie została wykreślona wcześniej, to nie jest wielokrotnością żadnej mniejszej liczby. Tym samym na końcu algorytmu liczby pierwsze to dokładnie te, które nie zostały wykreślone.

Możliwa (i dość wygodna) implementacja tego algorytmu używa tablicy *pierwsza*[1...*N*], w której każda komórka ma wartość *prawda* lub *fałsz*. Wartość *fałsz* odpowiada liczbie wykreślonej, wartość *prawda* – liczbie, która wciąż ma szansę być liczbą pierwszą. Na końcu algorytmu wszystkie wartości *k*, dla których *pierwsza*[*k*] = *prawda*, będą liczbami pierwszymi. Zmienna *k* odpowiada rozważaniu kolejnych liczb w tablicy, a zmienna *w* przechodzi przez wielokrotności *k* (poczynając od 2*k*, a kończąc, gdy zostanie przekroczone *N*).

```
funkcja sito(N)
        dla k = 2, 3, ..., N
                w = 2*k
                dopóki w<=N
                        pierwsza[w] = fałsz
                        w = w+k
        dla k = 2, 3, ..., N
                jeżeli pierwsza[k] = prawda
                        wypisz(k)
```

Często zamiast linijki:

```
                w = 2*k
```

pisze się:

```
                w = k*k
```

tzn. wykreślanie wielokrotności 2 zaczyna się od 2*2=4, wielokrotności 3 – od 3*3=9, wielokrotności 5 – od 25 itd. Można to zrobić dlatego, że wcześniejsze wielokrotności na pewno zostały już wykreślone (2**k* przy 2, 3**k* przy 3 itd.). To w pewnym stopniu przyśpiesza algorytm, chociaż na klasę złożoności nie wpływa: algorytm działa, w obu wersjach, w dość nietypowej złożoności $O(\log \log N)$ – zapis log log *N* oznacza logarytm z logarytmu liczby *N*, przy czym oba logarytmy są dwójkowe. Na przykład $\log\log 16 = \log 4 = 2$, a $\log\log 1\,000\,000\,000 \approx \log 30 < 5$. Wartość $\log\log N$ jest bardzo mała dla wszystkich „rozsądnych" wartości *N*. Dowód, dlaczego złożoność jest akurat taka, wymaga dość zaawansowanej matematyki i przekracza, niestety, ramy tej książki.

Schemat Hornera

schemat Hornera ▸ Schemat Hornera jest algorytmem używanym najczęściej na liczbach całkowitych lub rzeczywistych. Służy do szybkiego i efektywnego obliczenia wartości wielomianu w podanym punkcie. Innymi słowy, gdy mamy daną tablicę *A*[*0*...*n*] i pewną liczbę *x*, interesuje nas wartość:

$$A[0] + x \cdot A[1] + x^2 \cdot A[2] + x^3 \cdot A[3] + \ldots + x^n \cdot A[n]$$

Można do jej obliczenia podchodzić na kilka sposobów. Schemat Hornera jest najczęściej najprostszy w implementacji, a jednocześnie szybki. Opiera się na obliczaniu kolejno wartości:

$$A[n]$$
$$A[n-1]+x\cdot A[n]$$
$$A[n-2]+x\cdot A[n-1]+x^2\cdot A[n]$$
$$\ldots$$
$$A[0]+x\cdot A[1]+x^2\cdot A[2]+\ldots+x^n\cdot A[n]$$

Każda następna wartość w tym ciągu powstaje z poprzedniej przez pomnożenie jej przez x oraz dodanie kolejnego wyrazu, np.:

$$A[n-2]+x\cdot A[n-1]+x^2\cdot A[n]=A[n-2]+x(A[n-1]+x\cdot A[n]).$$

Stąd już łatwo o stosunkowo prosty algorytm, działający w liniowej złożoności:

```
funkcja Horner(A[0...n], x)
        w = 0
        dla i = n, n-1, ... , 1, 0
                w = w*x + A[i]
        podaj wynik w
```

Zauważmy, że widzieliśmy już jedną sytuację, w której obliczaliśmy podobną wartość (a wręcz stosowaliśmy schemat Hornera, nie wiedząc, że się tak nazywa) – obliczanie wartości liczby zapisywanej w różnych systemach liczbowych. Na przykład liczba 4392_7, czyli liczba zapisywana w systemie siódemkowym jako 4392, to tak naprawdę:

$$2+3\cdot 7+9\cdot 7^2+2\cdot 7^3$$

Moglibyśmy więc ją zapisać jako wartość funkcji *Horner*([2,9,3,4],7). W tablicy są zapisane cyfry naszej liczby w odwrotnym porządku, a x to podstawa systemu.

ZADANIA DO WYKONANIA

1. W turnieju tenisowym bierze udział pewna liczba zawodniczek. Turniej rozgrywany jest systemem pucharowym: w każdej rundzie zawodniczki dobierane są w pary i rozgrywają mecz, wygrywająca przechodzi dalej, przegrywająca odpada. Dzieje się tak, aż pozostanie tylko jedna zawodniczka. Jeśli w danej rundzie zawodniczek jest nieparzyście wiele, jedna ma „wolny los" i przechodzi dalej bez konieczności rozgrywania spotkania.

   Ile rund zostanie rozegranych, jeśli jest 16 zawodniczek? Ile, jeśli jest ich 31? Podaj wzór na liczbę rund, jeśli zawodniczek jest N.

2. Zaimplementuj „pierwiastkowy" algorytm sprawdzania pierwszości (funkcję `pierwsza(x)`) w wybranym języku programowania tak, aby mógł działać na liczbach 64-bitowych (typ `long long` w C++, w języku Python nie trzeba przejmować się wielkością liczb). Jak długo będzie działał dla liczb o 18–19 cyfrach, np. dla `x = 1 000 000 000 000 000 003`? Czy nadawałby się do sprawdzenia pierwszości liczb mających np. 100 cyfr dziesiętnych?

3. Zaimplementuj w wybranym języku programowania algorytm sita Eratostenesa i użyj go, aby wypisać wszystkie liczby pierwsze mniejsze od 50 000.

4. Stwórz tablicę $A[0...20]$, w której $A[k] = k$ dla $k = 1, 2, ..., 20$. Oblicz za pomocą schematu Hornera wartość `Horner(A, x)` dla $x = 0.25$.

PODSUMOWANIE LEKCJI

s. 11

- Algorytm sita Eratostenesa służy do obliczania naraz wszystkich liczb pierwszych mniejszych od N. Opiera się na wykreślaniu spośród wszystkich liczb kolejnych wielokrotności, a działa w czasie bliskim liniowemu.

s. 12

- Schemat Hornera to technika, która pozwala w prosty sposób, w czasie liniowym, obliczyć wartość wielomianu.

2. Rekursja raz jeszcze, czyli dziel i zwyciężaj

NA TEJ LEKCJI:

- dowiesz się więcej o tym, kiedy możesz użyć rekursji do rozwiązywania problemów;
- nauczysz się bardzo prostego w implementacji rekurencyjnego wariantu algorytmu szybkiego potęgowania;
- przypomnisz sobie, czym jest metoda *dziel i zwyciężaj*.

Jak być może pamiętasz z poprzednich lekcji, z rekursją mamy do czynienia, gdy funkcja (procedura) wywołuje samą siebie. W tym rozdziale przedyskutujemy bliżej dwa przykłady zapisane w pseudokodzie:

```
funkcja suma_cyfr(x)
        jeżeli x<10, zwróć x
        w przeciwnym razie:
                zwróć suma_cyfr(x div 10) + x mod 10
```

Ta funkcja przyjmuje jako argument liczbę naturalną x i oblicza jej sumę cyfr. Na przykład: żeby obliczyć sumę cyfr liczby $x = 254$, oblicza najpierw wynik dla liczby 10 razy mniejszej (x div 10 = 25), a do uzyskanego wyniku (7) dodaje ostatnią cyfrę (x mod 10 = 4).

```
funkcja potęga(a, n)
        jeżeli n = 0, zwróć 1
        w przeciwnym razie:
                zwróć a * potega(a, n-1)
```

Ta funkcja, dla danej liczby rzeczywistej a i liczby naturalnej n, oblicza wartość a^n, wykorzystując zależność $a^n = a^{n-1} * a$.

Kiedy rekursja działa?

warunki działania rekursji

Aby program oparty na rekursji mógł działać, konieczne jest spełnienie trzech warunków:

1. **Wywołanie rekurencyjne zawsze następuje na mniejszych danych** – mniejszej liczbie, mniejszej tablicy, krótszym ciągu itp. Wywołanie rekurencyjne w stylu:

```
funkcja f(n)
        (...)
        policz wartość f(2*n)
        (...)
```

   nie ma szansy powodzenia – gdybyśmy chcieli obliczyć *f*(10), spowodowałoby to wywołanie kolejno *f*(20), *f*(40), *f*(80)... bez żadnej perspektywy, że obliczenia kiedyś się skończą.

2. **Na odpowiednio małych danych potrafimy obliczyć odpowiedź bez rekursji.** Przykładowa funkcja `potega(a, n)` zawsze dochodziła w końcu do wywołania `potega(a, 0)`,dla którego automatycznie podawała odpowiedź 1. Funkcja `suma _ cyfr(x)` przerywa działanie i podaje odpowiedź, jeśli *x* jest liczbą jednocyfrową. Instrukcje, które wykonują się dla odpowiednio małych danych zamiast wywołań rekurencyjnych, nazywamy *początkiem rekursji*. Bez początku program będzie działał w nieskończoność, jak ten powyżej.
3. **Wynik działania funkcji jest możliwy do odtworzenia z tego, co dostaniemy z rekursji**. W funkcji `potega(a, n)` szukaliśmy liczby *a*^*n*. Wywołanie rekurencyjne `potega(a, n-1)` dało nam liczbę a^{n-1}, o której wiem, że wystarczy ją pomnożyć przez *a*, żeby dostać a^n.

Podobnie cały pomysł funkcji `suma _ cyfr(x)` opiera się na tym, że sumę cyfr liczby *x* możemy łatwo obliczyć z sumy cyfr liczby *x*/10. Jeśli zastanawiasz się, kiedy użyć rekursji, ten warunek ci to podpowie: jeżeli wynik dla twoich danych (np. liczby naturalnej *N*) da się obliczyć z analogicznego wyniku dla mniejszych danych (z wyniku dla liczby *N*−1 albo z wyniku dla *N*/2) – wtedy możesz użyć rekursji; co więcej, wtedy możesz mieć pewność, że ona zadziała.

Szybkie potęgowanie raz jeszcze

rekurencyjne szybkie potęgowanie

Dzięki użyciu rekursji można stworzyć wariant algorytmu szybkiego potęgowania, który jest prostszy w implementacji i łatwiejszy do zapamiętania: wiemy już, że liczbę a^n da się odtworzyć z a^{n-1}. Bądźmy jednak nawet odważniejsi: jeśli *n* jest parzyste, liczbę a^n da się łatwo odtworzyć, mając $a^{n/2}$ – ta pierwsza liczba jest kwadratem tej drugiej. Na przykład: $7^6 = (7^3)^2$, a $2^{50} = (2^{25})^2$. Napiszmy zatem funkcję rekurencyjną `potega(a, n)`, która będzie korzystać z wywołania rekurencyjnego `potega(a, n/2)` zawsze wtedy, kiedy to będzie możliwe, czyli dla *n* parzystych. Z kolei dla *n* nieparzystych skorzystamy, tak jak poprzednio, z wywołania `potega(a, n-1)`:


```
funkcja potega(a, n)
        jeżeli n = 0, zwróć 1
        w przeciwnym razie:
                jeżeli n jest parzyste
                        oblicz w = potega(a, n/2)
                        zwróć w*w
                jeżeli n jest nieparzyste
                        zwróć a*potega(a, n-1)
```

Ile będzie wywołań rekurencyjnych w takiej funkcji? Wywołania na parzystych n zmniejszają liczbę n dwukrotnie – jak pamiętamy z rozdziału o logarytmie – nie może ich być więcej niż $\log(n)$, bo liczbę n możemy podzielić przez 2 tylko $\log(n)$ razy. Ale nie każde wywołanie tak robi – wywołania dla nieparzystych n tylko zmniejszają wykładnik o 1. Zauważmy jednak, że jeśli n jest nieparzyste, to $n-1$ jest parzyste, a więc nie będzie nigdy dwóch kolejnych wywołań na nieparzystym n. Tym samym wiemy, że co drugie wywołanie (lub nawet częściej) dzieli n przez 2, a to znaczy, że nie będzie ich więcej niż $2*\log(n)$. Całkowita złożoność algorytmu jest więc logarytmiczna, a to znaczy, że może działać efektywnie nawet dla bardzo dużych n.

Dziel i zwyciężaj

metoda dziel i zwyciężaj

Zauważmy, że nic nie stoi na przeszkodzie, aby wewnątrz jednego wywołania funkcji użyć rekursji kilkukrotnie. Rozważmy na przykład poniższą funkcję, przyjmującą jako argument liczbę naturalną x:

```
funkcja f(x)
        jeżeli x = 0 lub x = 1, zwróć x
        w przeciwnym razie
                zwróć f(x-1) + f(x-2)
```

Aby obliczyć np. $f(3)$, nastąpią wywołania $f(2)$ i $f(1)$. Wywołanie $f(1)$ zakończy się od razu (z wartością 1), a $f(2)$ wywoła z kolei $f(1)$ i $f(0)$, które zwrócą odpowiednio 1 i 0, skąd wyliczy się $f(2) = 1$ i ostatecznie $f(3) = 2$.

Szczególnym rodzajem rekursji jest metoda zwana *dziel i zwyciężaj* (a po angielsku *divide and conquer*): otrzymane dane – najczęściej ciąg lub tablicę – dzielimy na dwie (czasem więcej) części, obliczamy wynik dla każdej z części, po czym uzyskane wyniki częściowe łączymy tak, aby uzyskać wynik dla całych danych.

Dla przykładu spróbujmy taką metodą obliczyć *największy* element danej tablicy liczb $A[1 \ldots n]$. Aby użyć rekursji, zaprojektujemy funkcję `najwiekszy(A, poczatek, koniec)`, która przyjmuje dwa indeksy `poczatek` i `koniec` i ma za zadanie znaleźć największy element między `A[poczatek]` a `A[koniec]`. Zrobi to, dzieląc tę część tablicy na dwie połówki i wywołując się na każdym osobno:

```
funkcja najwiekszy(A, poczatek, koniec)
        jeżeli poczatek=koniec,
                zwróć A[poczatek]
        srodek = (poczatek+koniec)/2
```

```
L = najwiekszy(poczatek, srodek)
P = najwiekszy(srodek+1, koniec)
jeżeli L>P
        zwróć L
w przeciwnym razie
        zwróć P
```

Zastanówmy się, jak nasza metoda zadziała np. na tablicy A = [4, 1, 7, 3]. Pierwsze wywołanie to `najwiekszy(A, 1, 4)`, które z kolei wywoła `najwiekszy(A, 1, 2)` i `najwiekszy(A, 3, 4)`. Wywołanie `najwiekszy(A, 1, 2)` będzie potrzebowało wyników `najwiekszy(A, 1, 1)` oraz `najwiekszy(A, 2, 2)` – te już obliczymy bez rekursji; będą wynosiły odpowiednio $A[1] = 4$ i $A[2] = 1$. Zatem `najwiekszy(A, 1, 2)` = 4. Podobnie `najwiekszy(A, 3, 4)` wywoła `najwiekszy(A, 3, 3)`, które zwróci wartość $A[3] = 7$, a także `najwiekszy(A, 4, 4)` = `A[4]` = 3. Zatem `najwiekszy(A, 3, 4)` = 7. Ostatnia instrukcja to porównanie obliczonych wartości `L=najwiekszy(A, 1, 2)` = 4 oraz `P=najwiekszy(A, 3, 4)` = 7, z których większą jest 7.

Nie powinno być zaskoczeniem, że ten algorytm jest prawidłowy, skoro spełnia podane wcześniej warunki: wywołuje się dla mniejszych danych (choć kilkukrotnie), w granicznych wypadkach (tutaj: dla jednoelementowych tablic) podaje wartość bez rekursji oraz prawidłowo odtwarza ostateczny wynik z częściowych wyników otrzymanych z rekursji.

Oczywiście w praktyce nie używa się tego sposobu do obliczania największej wartości w tablicy – są łatwiejsze sposoby. Jednak istnieją problemy, dla których metoda *dziel i zwyciężaj* okazuje się najlepsza. W kolejny rozdziałach poznamy dwa takie przykłady, oba związane z sortowaniem tablic: sortowanie przez scalanie oraz sortowanie szybkie.

ZADANIA DO WYKONANIA

1. Napisz rekurencyjny algorytm podobny do `suma_cyfr(x)`, który obliczy największą cyfrę zadanej liczby x. Twoja funkcja powinna zatem przyjmować jedną liczbę naturalną x i zwracać liczbę naturalną będącą największą cyfrą x.

2. Rozważ poniższą funkcję rekurencyjną, przyjmującą jako argument liczbę naturalną x:

```
funkcja f(x)
      jeżeli x = 0, zwróć 1
      w przeciwnym razie
            zwróć f(x-1) + f(x-1)
```

   Ile wyniesie `funkcja(3)`? Ile wyniesie `funkcja(4)`? Co tak naprawdę oblicza `funkcja(x)` i ile wykonuje kroków?

3. Czemu poniższy algorytm nie jest poprawny? Jak by zadziałał, gdyby go zaimplementować np. w C++ lub w Pythonie? Jak poprzednio zakładamy, że argument x jest liczbą naturalną.

```
funkcja silnia(x)
      w = silnia(x-1)
      zwróć w*x
```

4. Dana jest poniższa funkcja rekurencyjna, przyjmująca jako argument łańcuch znaków `napis[1...n]`:

```
funkcja f(napis, k)
      jeżeli k = 0, powróć
      w przeciwnym razie
            wypisz napis[k]
            funkcja(napis, k-1)
```

Co zwróci wywołanie `f("abc",3)`? Jeśli *n* jest długością napisu, co zwróci wywołanie `f(napis, n)`?

PODSUMOWANIE LEKCJI

s. 15

- Rekursja w algorytmach działa, jeśli są spełnione trzy warunki: wywołanie rekurencyjne następuje zawsze na mniejszych danych, jest określony początek rekursji oraz wynik jest prawidlowo odtwarzany z wyników wywołań rekurencyjnych.

s. 15

- Szybkie potęgowanie – działające w czasie $O(\log N)$, gdzie N jest wykładnikiem potęgi do obliczenia – można łatwo zaimplementować za pomocą rekursji.

s. 16

- Metoda *dziel i zwyciężaj* działa przez podzielenie danych na dwie (czasem więcej) części, wywołanie się na każdej osobno, a potem połączenie uzyskanych wyników w jeden.

3. Sortowanie przez scalanie, czyli pierwszy efektywny algorytm sortowania

NA TEJ LEKCJI:

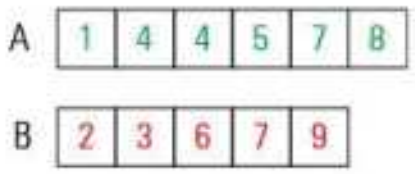
- nauczysz się łączyć dwie posortowane tablice w jedną;
- wykorzystasz tę procedurę, aby rekurencyjnie sortować dowolną tablicę;
- poznasz pierwszy algorytm sortowania działający szybko – w czasie $O(n \log n)$.

Rozgrzewka: scalanie tablic

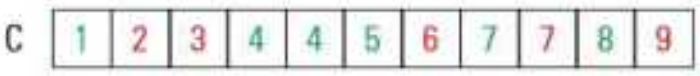
scalanie posortowanych tablic

Zanim przejdziemy do właściwego algorytmu, rozwiążemy najpierw inny, nieco prostszy problem – wkrótce przekonasz się, jak go zastosujemy w ostatecznym algorytmie.

W problemie scalania tablic mamy dane dwie tablice: $A[1 \ldots n]$ oraz $B[1 \ldots m]$, obie ustawione w porządku niemalejącym. Wszystkie podane w tym rozdziale algorytmy działają na dowolnym rodzaju elementów, które da się ze sobą porównywać (liczby całkowite, liczby rzeczywiste, słowa według porządku w słowniku itd.). W przykładowych tablicach, dla prostoty i oszczędności miejsca, będziemy przechowywać liczby całkowite.



Tablice *A* i *B* chcemy *scalić*, to znaczy elementy z nich połączyć w jednej wspólnej tablicy $C[1 \ldots n+m]$, która docelowo powinna wyglądać tak:



Czyli: dane są posortowane na tablice $A[1 \ldots n]$ oraz $B[1 \ldots m]$, a teraz musimy zapisać algorytm, którego wynikiem będzie tablica *C*, zawierająca wszystkie elementy tablic *A* i *B*, również posortowana niemalejąco. Oczywiście tablica *C* będzie miała długość $m+n$.

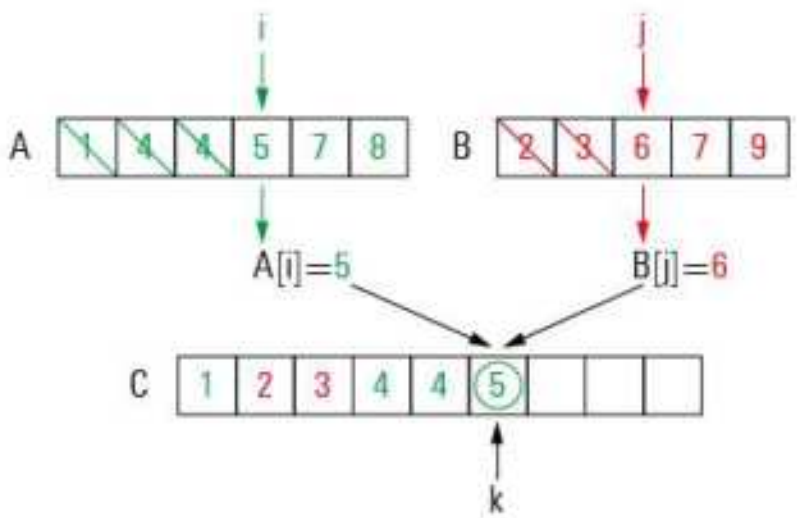
Algorytm najłatwiej opracować, rozważając elementy od najmniejszych: który element może trafić na miejsce $C[1]$? Musi być najmniejszy ze wszystkich, a więc nie może to być inny element niż $A[1]$ lub $B[1]$ – pozostałe elementy tablic *A* i *B* są większe, a więc nie powinny trafiać na początek. Z kolei z elementów $A[1]$ i $B[1]$ wybieramy oczywiście mniejszy: jeśli $A[1] < B[1]$, to na pozycję $C[1]$ ma trafić $A[1]$, w przeciwnym wypadku powinien to być $B[1]$. Zrobiliśmy więc pierwszy krok algorytmu:

```
jeżeli A[1]<B[1], to C[1] = A[1]
w przeciwnym razie C[1] = B[1]
```

Jeden krok to oczywiście za mało, docelowo musimy wpisać wszystkie elementy za pomocą pętli:

```
dla k = 1, 2, ..., m+n
//     wybierz, jaki element powinien stać na pozycji C[k]
i wpisz go w to miejsce.
```

Zastanówmy się zatem, jaki powinien być krok drugi. Jeśli na $C[1]$ wybraliśmy $A[1]$, to kandydatami na kolejny element są znowu dwa najmniejsze – $A[2]$ oraz $B[1]$. Możemy sobie wyobrazić, że po przepisaniu $A[1]$ do tablicy *C* wykreśliliśmy go z *A* i zapomnieliśmy o nim. Z dwóch aktualnych kandydatów na kolejny element *C* wybieramy teraz mniejszy. Następnie kontynuujemy ten sam schemat: w każdym kroku porównujemy ze sobą pewien element $A[i]$ tablicy *A* z pewnym elementem $B[j]$ tablicy *B* – i mniejszy z nich przepisujemy do *C*. Następnie zwiększamy o 1 zmienną *i* albo zmienną *j*, co odpowiada wykreśleniu tego elementu z *A* lub *B*:



Prawie kompletny algorytm wygląda zatem tak:

```
procedura Scal(A,B,C)
      dla k = 1, 2, ..., m+n
                  jeżeli A[i]<B[j]
                        C[k] = A[i]
                        i = i+1
                  w przeciwnym razie
                        C[k] = B[j]
                        j = j+1
```

Jest prawie kompletny, bo zapomnieliśmy o ważnej rzeczy: elementy w którejś tablicy mogą się skończyć przed drugą: jeśli $i > n$, to element $A[i]$ nie istnieje i nie możemy go porównywać – wtedy po prostu bierzemy element z B. Podobnie jeśli $j > m$, wybieramy element z A bez porównywania. Sprawdzamy najpierw zatem, czy nie zaszedł ten przypadek:

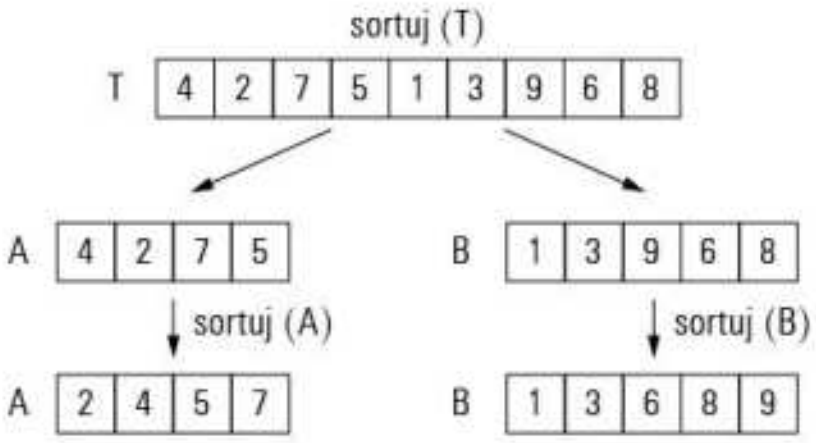
```
procedura Scal(A,B,C)
      dla k = 1, 2, ..., m+n
      jeżeli (i<=n oraz j<=m oraz A[i]<B[j])
      lub jeżeli j>m:
            C[k] = A[i]
            i = i+1
      w przeciwnym razie
            C[k] = B[j]
            j = j+1
```

Rekurencyjne sortowanie

Jak przekonaliśmy się rozdziale *Rekursja*, „magia" rekurencji pozwala na wywołanie przez funkcję samej siebie – co więcej, zawsze zadziała, o ile prawdą są trzy rzeczy:

1. Wywołujemy się rekurencyjnie zawsze na mniejszych danych.
2. Wiemy, kiedy skończyć (dla odpowiednio małych danych liczymy odpowiedź bez rekursji).
3. Wiemy, jak z uzyskanych odpowiedzi na mniejszych danych zrobić odpowiedź na większych.

Spróbujmy zastosować tę strategię do znanego już problemu sortowania. Mamy zatem tablicę $T[1 \ldots n]$ pewnych elementów i chcemy ustawić ją w kolejności rosnącej. Algorytm sortowania przez scalanie dzieli tablicę T na dwie (możliwie równe) części A i B, po czym na każdej próbuje wywołać się rekurencyjnie. (Gdyby przypadkiem było $n = 1$, czyli tablica T była jednoelementowa, nie ma potrzeby jej dzielić ani w ogóle sortować – jest już posortowana i nic nie musimy robić). Załóżmy przez chwilę – jak to przy rekursji bywa – że wywołanie się udało i mamy dwie mniejsze tablice posortowane rekurencyjnie:



Warunek rekursji numer 1 jest w porządku: tablice *A* i *B* są mniejsze niż *T* (obie są wielkości, w przybliżeniu, *n*/2). Zapewniliśmy sobie również warunek 2: dla tablicy długości 1 ustaliliśmy już, że nic nie będziemy robić. Ostatnie, czego potrzebujemy, to zrealizować punkt 3: z uzyskanych dwóch posortowanych tablic zrobić jedną większą. Zauważmy, że właśnie to robi poznany przez nas poprzednio algorytm scalania. Musimy scalić tablice *A* i *B* tak, aby połączone elementy trafiły z powrotem do tablicy *T*, czyli uruchomić funkcję `Scal(A,B,T)`. Pełny algorytm wygląda zatem tak:

```
procedura Sortuj(T[1...n])
      jeżeli n = 1
            powrót
      A = T[1...n/2]
      B = T[n/2+1...n]
      Sortuj(A)
      Sortuj(B)
      Scal(A,B,T)
```

Przyjrzyjmy się, jak nasza nowa funkcja `Sortuj()` zadziała na tablicy *T* = [4,2,3,1]:

1. Podzieli *T* na dwie części: [4,2] i [3,1].
2. Wywoła się rekurencyjnie: `Sortuj([4,2])`
   - podzieli [4,2] na kawałki [4] i [2], po czym wywoła `Sortuj([4])` oraz `Sortuj([2])`;
   - `Sortuj([4])` oraz `Sortuj([2])` pozostawiają jednoelementowe tablice bez zmian;
   - wywoła `Scal([4],[2])` i otrzyma tablicę [2,4] (posortowaną!).
3. Wywoła się rekurencyjnie na [3,1]:
   - podzieli [3,1] na kawałki [3] i [1], po czym wywoła się na nich;
   - `Sortuj([3])` i `Sortuj([1])` nie muszą nic robić;
   - scali [3] oraz [1] w tablicę [1,3] i zakończy to wywołanie.
4. Wywoła `Scal([2,4],[1,3])`, czego efektem będzie tablica [1,2,3,4] – posortowana oryginalna tablica *T*.

sortowanie przez scalanie, *MergeSort*

Opisany algorytm bierze swoją nazwę od operacji scalania i nazywa się sortowaniem przez scalanie (ang. *MergeSort*). W praktyce prawdopodobnie spotkasz się częściej z nazwą angielską niż z polską.

Złożoność algorytmu i kwestie praktyczne

Spróbujmy oszacować, ile operacji łącznie wykona algorytm sortowania przez scalanie. Nie jest to łatwe zadanie (jak często przy algorytmach rekurencyjnych), gdyż nasz nowy algorytm jest szybszy od wszystkich wcześniej poznanych. Algorytm wykonuje kilka rodzajów operacji, np. dzieli tablice na pół, porównuje elementy ze sobą oraz wpisuje elementy do tablic. Aby liczyło się łatwiej, możemy skoncentrować się wyłącznie na operacjach porównywania elementów (można zauważyć, że na jedną operację porównania przypada 1 lub przypadają 2 inne, więc wystarczy policzyć porównania, żeby mieć też dobre oszacowanie pozostałych).

Wszystkie porównania następują wewnątrz funkcji `Scalaj()`. Aby połączyć ze sobą dwie tablice $A[1 \ldots n]$ i $B[1 \ldots m]$, cała procedura wykonuje $n+m$ porównań (czasem mniej, ale tym również nie musimy się przejmować w tej chwili).

Dla przykładu rozważmy, jak algorytm zadziała na tablicy T długości $n = 16$:

- Podzieli tablicę na dwa kawałki po 8 elementów, których późniejsze scalenie w jedną tablicę będzie wymagało 8+8 = 16 porównań.
- Każdy z kawałków podzieli na dwa czteroelementowe mniejsze fragmenty: aby je po wszystkim scalić, będzie potrzebował (4+4) + (4+4) = 16 operacji.
- Każdy z 4 kawałków długości 4 podzieli na fragmenty długości 2: scalenie każdego z nich to 2+2 = 4 operacje, więc łącznie porównań będzie znowu 16.
- Każdy z 8 kawałków długości 2 podzieli na fragmenty długości 1, a późniejsze scalenie będzie wymagało 8*2 = 16 (znowu!) operacji.
- Kawałki długości 1 nie będą wymagały scalania.

Liczbę 16 wybraliśmy celowo: gdybyśmy zamiast niej wzięli inne n, kawałki nie byłyby równe, a obliczenia okazałyby się trudniejsze. Można jednak sprawdzić, że efekt będzie bardzo podobny – na każdym poziomie rekurencji łączna liczba porównań będzie równa (około) n.

Poziomów było w powyższym przykładzie 4. Ustalmy ogólną zasadę dla tablicy długości n. Widać, że na każdym kolejnym poziomie kawałki mają dwukrotnie mniejszą długość – ogólnie poziomów jest tyle, ile razy możemy dzielić n przez 2, aż długość kawałków spadnie do 1. Jak już wiemy, ta liczba to logarytm dwójkowy z n, a zatem mamy odpowiedź: algorytm sortowania przez scalanie wykonuje (około) $n \log n$ porównań.

Taki czas działania to ogromny zysk w porównaniu z wcześniej poznanymi algorytmami o złożoności $O(n^2)$, takimi jak sortowanie przez wstawianie czy bąbelkowe. Dość powiedzieć, że sortowanie tablicy rozmiaru 1 000 000 (milion) przez scalanie wymagałoby ~ 20 000 000 operacji, a przez wstawianie ~ $5*10^{11}$ = 500 000 000 000 (pięćset miliardów). Pierwsze wykonuje się w ułamku sekundy, a drugie wymaga przynajmniej kilku minut.

Czy zatem sortowania przez scalanie używa się w praktyce? Okazuje się, że nie! Zauważ, że nasza funkcja `Sortuj()` potrzebuje przepisać elementy do

pomocniczych tablic. To zarówno zużywa dodatkową pamięć, jak i trochę zwalnia algorytm. Opisany w kolejnym rozdziale algorytm sortowania szybkiego ma (na ogół) tę samą złożoność $O(n \log n)$, a dodatkowo jest pozbawiony tej wady – dlatego to on, a nie *MergeSort* zajmuje miejsce najpowszechniej stosowanego algorytmu.

ZADANIA DO WYKONANIA

1. Zaimplementuj algorytm sortowania przez scalanie w wybranym języku programowania (w C++ najwygodniej użyć, zamiast tablic, wektorów, czyli typu `vector<int>`).
2. Ile (orientacyjnie) będzie równa wartość $n \log n$ dla $n = 1\,000\,000$. Ile (orientacyjnie) zajmie czasu na komputerze? Co można z tego wywnioskować o praktycznej przydatności algorytmu sortowania przez scalanie i algorytmów kwadratowych (sortowania bąbelkowego/przez wstawianie) na danych tej wielkości?
3. Napisz w pseudojęzyku algorytm, który przyjmuje trzy posortowane tablice i scala ich elementy w jedną tablicę, analogicznie do procedury `Scal()`.

PODSUMOWANIE LEKCJI

- Dwie posortowane tablice możemy scalić w czasie liniowym procedurą, która za każdym razem wybiera mniejszy z dwóch początkowych elementów, a następnie wykreśla go z tej tablicy. s. 18
- Sortowanie przez scalanie dzieli tablicę na dwie mniejsze, sortuje każdą z nich rekurencyjnie, po czym łączy w jedną za pomocą algorytmu scalania. s. 21
- Algorytm sortowania przez scalanie działa w czasie $O(n \log n)$. s. 21

4. Sortowanie szybkie, czyli jak się sortuje w praktyce

NA TEJ LEKCJI:

- nauczysz się algorytmu sortowania szybkiego (*QuickSort*), w praktyce jednego z najszybszych algorytmów sortowania.

Sortowanie szybkie – pomysł

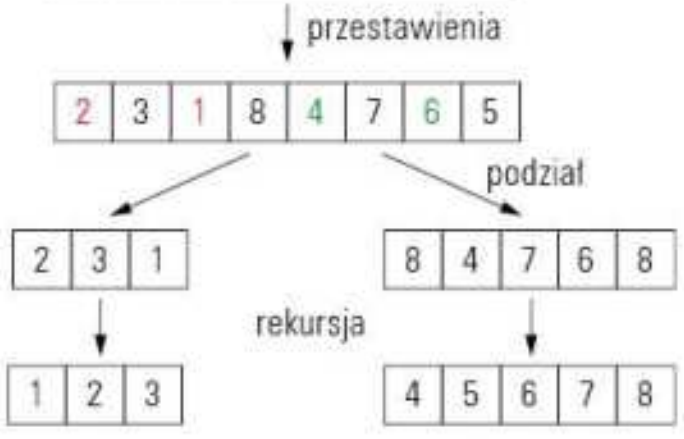
sortowanie szybkie, *QuickSort*

Algorytm *sortowania szybkiego*, wymyślony przez Tony'ego Hoare'a (czytaj: tonego hora) w 1959 r., częściej jest znany pod angielską nazwą *QuickSort*. Jego nazwa nie jest nadużyciem – faktycznie to jeden z najszybszych znanych algorytmów i jeden z najczęściej używanych w praktyce. Z poprzednio poznanym sortowaniem przez scalanie ma pewną cechę wspólną – również dzieli tablicę na dwie części i rekurencyjnie sortuje każdą z nich osobno (a więc również jest

algorytmem typu *dziel i zwyciężaj*). Różnica leży jednak w kolejności kroków: sortowanie przez scalanie najpierw wywoływało się rekurencyjnie, a potem łączyło (scalało) posortowane tablice. Sortowanie szybkie **najpierw wstępnie porządkuje tablicę i dzieli ją na dwie części tak, żeby po powrocie z rekursji cała tablica już była posortowana**. Możemy to opisać bardzo ogólnym schematem:

```
funkcja sortuj(A, poczatek, koniec)
      jeżeli poczatek = koniec
            nic nie rób
      podziel tablicę na dwie części A[poczatek...r]
                  oraz A[r+1...koniec]
      sortuj(A, poczatek, r)
      sortuj(A, r+1, koniec)
```

Jednak nie każdy podział tablicy sprawdzi się w takiej sytuacji. Wiemy, że po dwóch wywołaniach rekurencyjnych będą posortowane oba kawałki `A[poczatek...r]` oraz `A[r+1...koniec]`. Potem już algorytm nic nie będzie robił, a więc – żeby kolejność się zgadzała – już przed rekurencyjnym sortowaniem wszystkie elementy w lewym kawałku muszą być mniejsze od wszystkich elementów w prawym kawałku:



Idea działania algorytmu *QuickSort*: przestawienia elementów, podział i rekursja

Osiągamy to po wybraniu jednego z elementów tablicy (będziemy go nazywać *kluczem*, często spotyka się też angielską nazwę *pivot*) i przestawieniu elementów tak, aby mniejsze od klucza trafiły na lewą stronę tablicy, a większe od klucza – na prawą. Załóżmy wstępnie, że jako klucz bierzemy pierwszy napotkany element (czyli `A[poczatek]`) – w sekcji *Wybór klucza i złożoność* wyjaśnimy inne możliwe strategie wyboru klucza. Zastanówmy się najpierw, jak – mając już wybrany klucz – podzielić tablicę *A* na część „mniejszą od klucza" i część „większą od klucza".

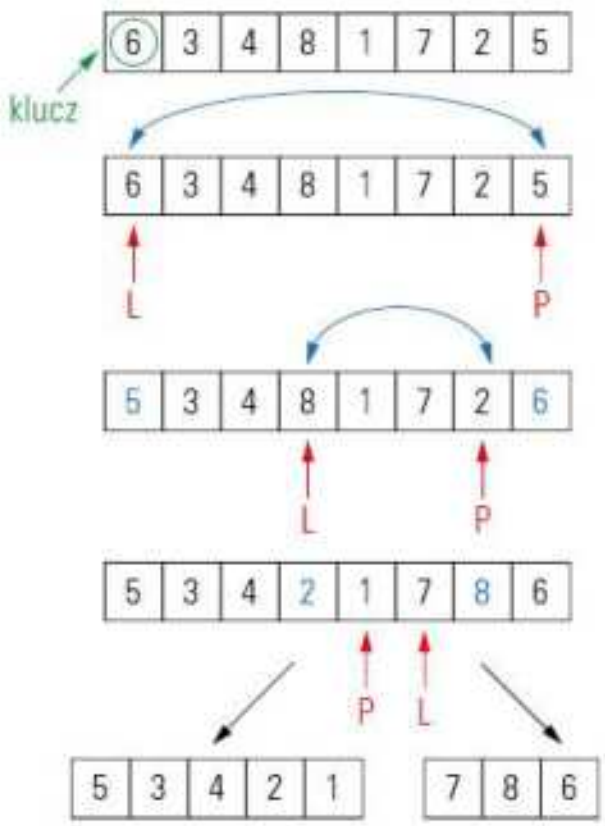
Algorytm podziału tablicy

Załóżmy, że wybraliśmy już *klucz* i chcemy przestawić elementy tablicy *A* tak, aby elementy mniejsze niż klucz były na lewo od elementów większych (elementy równe kluczowi mogą trafić gdziekolwiek).

Jest kilka algorytmów podziału – oryginalny pomysł Hoare'a, autora algorytmu *QuickSort*, opiera się na następującym schemacie:

- zmienna *L* zaczyna od początku przedziału tablicy (czyli *L* = poczatek);
- zmienna *P* zaczyna od końca przedziału (czyli *P* = koniec);
- zmienna *L* wyszukuje element większy od klucza (czyli taki, który powinien trafić na prawą stronę);
- zmienna *P* wyszukuje element mniejszy od klucza (czyli taki, który trzeba przestawić na lewą stronę tablicy);
- elementy *A*[*L*] i *A*[*P*] zamieniają się w tablicy, po czym *L* i *P* przesuwają się dalej – *L* w prawo, *P* w lewo;
- wszystko to dzieje się tak długo, aż *L* i *P* się spotkają (dokładniej, aż nastąpi *L* > *P*).

Szukanym podziałem tablicy *A* jest teraz `A[poczatek...P]` i `A[L...koniec]`.



Algorytm *QuickSort* na przykładowych danych: przestawianie elementów i podział na dwie tablice

Procedura podziału działa w czasie liniowym – zauważ, że zmienna L zawsze się zwiększa, więc nie może wykonać więcej niż (koniec-początek) kroków do przodu. Podobnie zmienna *P* nie może wykonać więcej niż (koniec–początek) kroków do tyłu.

Pseudokod całego algorytmu jest następujący:

```
funkcja sortuj(A, poczatek, koniec)
        jeżeli poczatek = koniec
                powróć
        klucz = A[poczatek]
        L = poczatek
```


```
P = koniec
dopóki L<=P wykonuj:
       dopóki A[L]<klucz
              L = L+1
       dopóki A[P]>klucz
              P = P-1
       jeżeli L<=P:
              zamień(A[L],A[P])
              L = L+1
              P = P-1
jeżeli P>=poczatek, sortuj(A,poczatek,P)
jeżeli L<=koniec, sortuj(A, L, koniec)
```

Wybór klucza i złożoność

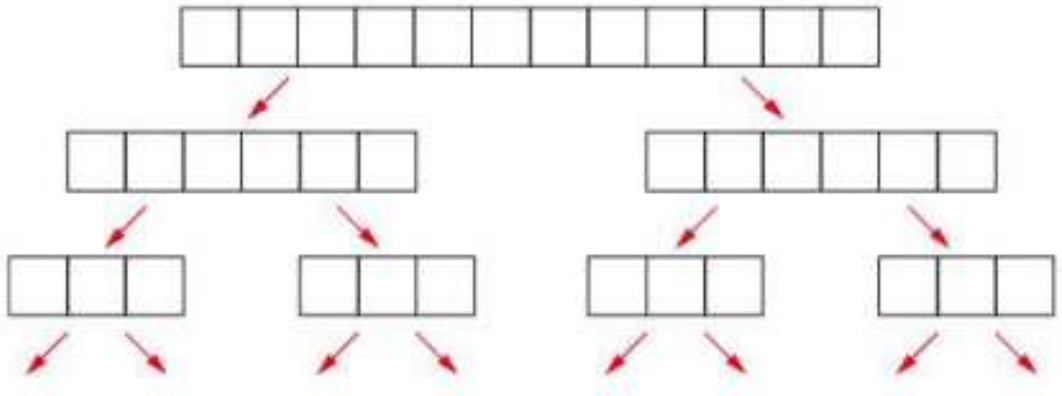
Jest kilka strategii wybierania klucza – : może nim być:

- pierwszy element przedziału (`A[poczatek]`),
- ostatni element przedziału (`A[koniec]`),
- środkowy element przedziału (`A[(poczatek+koniec)/2]`),
- wybrany losowo element przedziału.

Żadna z tych metod nie jest w stanie jednak zagwarantować równego podziału i nawet nie próbuje tego robić – części tablicy „lewa" i „prawa" mogą mieć różne wielkości (a jakie konkretnie, okazuje się dopiero w czasie działania algorytmu). To trochę utrudnia ocenę, jaka będzie złożoność algorytmu, ale zastanówmy się najpierw nad dwoma skrajnymi przypadkami:

1. Równy podział

Czasami zdarza się, że algorytm podziału rozbije tablicę na dwie części tego samego rozmiaru. Dzieje się tak wtedy, kiedy elementów mniejszych od klucza okazuje się być tyle samo, co większych. Rozważmy pomyślną sytuację, kiedy dzieje się tak przy każdym podziale tablicy:

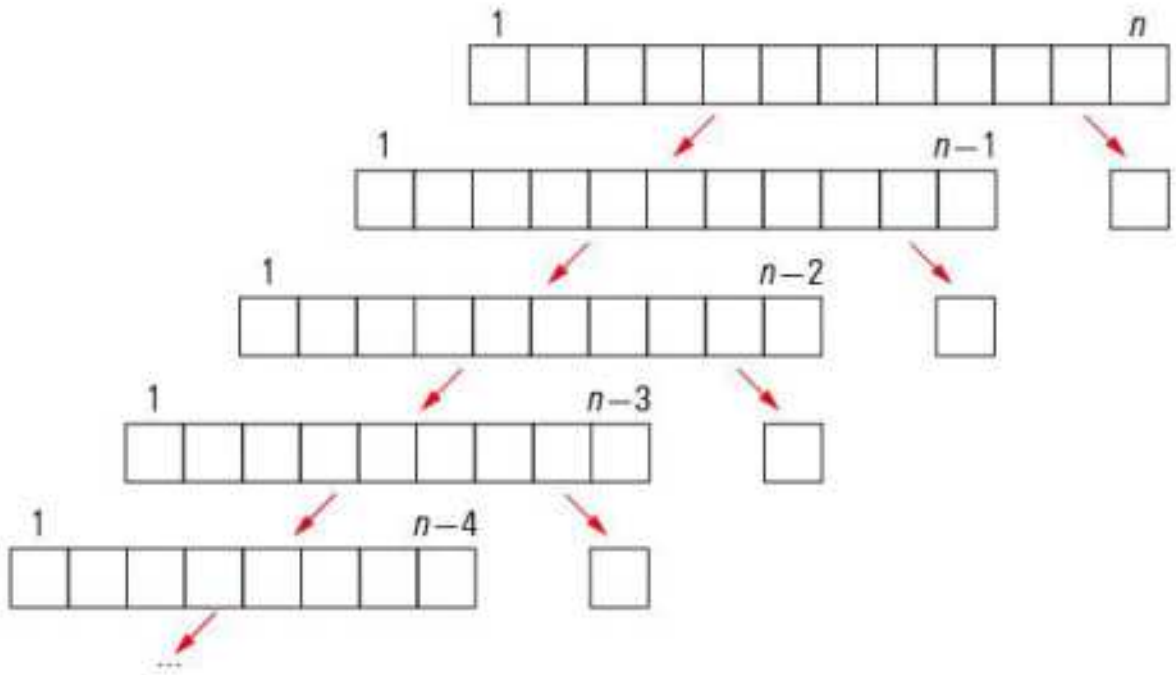


Równe podziały tablicy w algorytmie *QuickSort*

Rysunek wygląda podobnie jak w rozdziale o sortowaniu przez scalanie, a algorytm też wykonuje podobną liczbę operacji – dwa wywołania rekurencyjne na dwukrotnie mniejszych tablicach plus liniowa liczba innych operacji. Złożoność jest w takim szczególnym wypadku podobna do poprzedniego algorytmu, czyli wynosi $O(n \log n)$.

2. Pechowy podział

Co się jednak stanie, jeśli za *klucz* przyjmiemy największy element tablicy? Podział będzie wtedy bardzo nierówny: w jednej części będzie jeden element, a w drugiej – pozostałe:



Nierównomierne podziały tablicy

Jeśli początkowa tablica miała n elementów, do pierwszego wywołania trafi tablica mająca $n-1$ elementów. Gdyby algorytm pechowo za każdym razem dzielił się w ten sposób, całkowita liczba kroków wyniosłaby $n+(n-1)+(n-2)+...+1$. Ta suma wynosi $n^*(n+1)/2$ (jest to suma ciągu arytmetycznego), czyli $O(n^2)$. Algorytm ma w takim wypadku złożoność kwadratową.

pesymistyczna złożoność algorytmu *QuickSort*

Widzimy zatem, że przy nieszczęśliwych wyborach klucza algorytm sortowania szybkiego może jak najbardziej być kwadratowy. Mówi się, że ma *pesymistyczną złożoność* kwadratową. Czemu więc nazywa się szybkim i czemu używa się go w praktyce? Otóż na ogół większość podziałów będzie bliższa przypadkowi 1 (w miarę równy podział) niż przypadkowi 2 (pechowy, nierówny podział). Żeby nierówny podział nastąpił wielokrotnie, albo musielibyśmy mieć ogromnego pecha, albo ktoś musiałby spreparować bardzo odpowiednią tablicę wejściową, specjalnie pod nasz algorytm. Zauważmy też, że wyboru klucza możemy dokonać w sposób przypadkowy, zamiast linijki:

```
klucz = A[poczatek]
```

można wstawić:

```
s = losowa liczba z przedziału [poczatek, koniec]
klucz = A[s]
```

Wtedy w ogóle nie musimy się martwić, że trafimy na złośliwy przypadek spreparowanej tablicy. Można udowodnić (choć przekracza to znacznie ramy tego podręcznika), że wtedy prawie na pewno złożoność algorytmu wyniesie $O(n \log n)$. Mówi się, że dla algorytmu *QuickSort* w wersji z losowym wyborem klucza *złożoność średnia* (albo *złożoność oczekiwana*) wynosi $O(n \log n)$.

złożoność średnia algorytmu *QuickSort*

Praktyczne zastosowania

Oczekiwana złożoność algorytmu sortowania szybkiego jest taka sama jak opisanego w poprzednim rozdziale sortowania przez scalanie (*MergeSort*), czyli $O(n \log n)$. W praktyce *QuickSort* okazuje się być kilkukrotnie szybszy. Ponadto nie potrzebuje dodatkowej pamięci (sortuje *w miejscu*), podczas gdy *MergeSort* musi korzystać z dodatkowych tablic. Dlatego też do praktycznych zastosowań używa się sortowania szybkiego. W szczególności biblioteka standardowa w języku C++ (czyli funkcja *std::sort*) używa tego właśnie algorytmu, choć dodatkowo „zabezpiecza się" na wypadek pechowego wyboru klucza: gdyby procedura miała działać za długo, przestawia się na inny algorytm sortowania.

ZADANIA DO WYKONANIA

1. Jaką złożoność będzie miał algorytm sortowania szybkiego w wersji oryginalnej (kluczem jest pierwszy element tablicy) wywołany na tablicy, która już jest posortowana?
2. Zaimplementuj algorytm *QuickSort* w wybranym języku programowania.

PODSUMOWANIE LEKCJI

s. 23 • Algorytm sortowania szybkiego (*QuickSort*) wybiera klucz (jeden z elementów tablicy), przestawia elementy tablicy i dzieli ją tak, aby w lewej części były elementy mniejsze od klucza, a w prawej – większe. Potem wywołuje się rekurencyjnie na obu częściach.

s. 27 • Algorytm *QuickSort* ma złożoność pesymistyczną $O(n^2)$, ale średnią $O(n \log n)$ w wersji z losowaniem klucza.

s. 27 • Algorytm *QuickSort* jest w praktyce najszybszym i najczęściej stosowanym z dotychczas poznanych algorytmów.

5. Specjalne elementy w tablicy, czyli największy, najmniejszy i lider

NA TEJ LEKCJI:

- dowiesz się, jak zaoszczędzić na liczbie operacji przy szukaniu jednocześnie elementu największego i najmniejszego w tablicy;
- poznasz algorytm na szukanie dominanty (najczęstszego elementu) oraz algorytm na szukanie lidera (elementu występującego częściej niż wszystkie pozostałe razem).

Szukanie elementu najmniejszego i największego

Z poprzednich lekcji na pewno wiesz, jak wśród wielu elementów znajdować największy (albo najmniejszy). Dokładniej: mamy daną tablicę $A[1 \ldots n]$, w której jest n elementów – w najprostszym wypadku to zwykłe liczby całkowite, ale mogą zdarzyć się także liczby rzeczywiste, napisy (porównywane alfabetycznie) czy też jeszcze bardziej skomplikowane elementy.

Najprostszy algorytm wygląda tak:

```
funkcja najwiekszy(A)
        w = A[1]
        dla i = 2, 3, ..., n
                jeżeli A[i]>w
                        w = A[i]
        zwróć wynik w
```

Spróbujmy policzyć nieco dokładniej, ile operacji wykona ten algorytm. Za każdą iteracją pętli następuje jedno porównanie (jeżeli $A[i] > M$) i być może jedno przypisanie ($M = A[i]$), zależne od wyniku porównania. Dla ułatwienia będziemy zatem liczyć tylko instrukcje porównania, pamiętając, że przy każdej może nastąpić też przypisanie. Operacji porównania jest **dokładnie** $n-1$.

Gdybyśmy zamiast największego chcieli znajdować najmniejszy element w tablicy, kod programu wyglądałby tak samo, przy czym zamiast znaku „>" byłby znak „<".

jednoczesne znajdowanie elementów najmniejszego i największego

Co jednak, jeśli chcemy znaleźć w jednym algorytmie **jednocześnie element największy i najmniejszy**? Oczywiście możemy uruchomić kolejno powyższy algorytm w obu wersjach – pierwsza znajdzie największy, druga – najmniejszy element, wykonując łącznie $2n-2$ porównań. Istnieje jednak sztuczka, dzięki której można wykonać istotnie mniej porównań – około $3/2*n$.

Załóżmy najpierw na chwilę, że rozmiar tablicy n jest liczbą parzystą (za chwilę zajmiemy się również nieparzystymi). Połączmy – na razie tylko wirtualnie, w wyobraźni – elementy w pary: $A[1]$ z $A[2]$, $A[3]$ z $A[4]$, ..., $A[n-1]$ z $A[n]$. W każdej parze porównajmy elementy ze sobą. Zauważmy, że jeśli okaże się, że $A[1] > A[2]$, to $A[1]$ wciąż może być największym elementem, ale nie może już być najmniejszym. Z kolei $A[2]$ wciąż jest kandydatem na najmniejszy element, ale w kontekście największego możemy już nie brać go pod uwagę.

Zróbmy zatem algorytm podobny do poprzedniego, ale dodatkowo porównujący elementy w parach – z każdej pary mniejszy element będzie kandydatem tylko na najmniejszy, a większy element tylko na największy:

```
funkcja jednoczesne(A)
        w = A[1]
        m = A[1]
        i = 1
        dopóki i<n wykonuj:
                jeżeli A[i]>A[i+1]:
```


```
                jeżeli A[i]>w
                       w = A[i]
                jeżeli A[i+1]<m
                       m = A[i+1]
          jeżeli A[i]<A[i+1]
                jeżeli A[i+1]>w
                       w = A[i+1]
                jeżeli A[i]<m
                       m = A[i]
          i = i+2
zwróć w jako największy,
      m jako najmniejszy element
```

4	3	10	3	7	6	2	5	4	10	8	8

max: 4 10 7 5 10 8 → 10
min: 3 3 6 2 4 8 → 2

Z każdej pary elementów większy (zielony) jest brany pod uwagę przy szukaniu maksimum, a mniejszy (czerwony) – przy szukaniu minimum.

Policzmy porównania w nowym algorytmie:

- w każdej parze wykonujemy jedno porównanie ($A[i]>A[i+1]$) – łącznie $n/2$ porównań;
- w każdej parze jeden element (albo $A[i]$, albo $A[i+1]$) porównujemy z wartością zmiennej w, co daje $n/2$ porównań;
- w każdej parze jeden element porównujemy z aktualną wartością zmiennej m, czyli znowu $n/2$ porównań.

Łącznie wychodzi – jak zapowiedzieliśmy – $3/2*n$ porównań.

Pozostaje jeszcze „obsłużyć" przypadek, kiedy n nie jest liczbą parzystą. W takim wypadku dokładamy na koniec tablicy jeszcze jedną kopię ostatniego elementu. Dołożenie drugiej kopii istniejącego elementu nie może wpłynąć ani na największy, ani na najmniejszy element A. Za to teraz długość tablicy będzie parzysta i poprzedni algorytm możemy już zastosować bez przeszkód:

```
funkcja jednoczesne(A)
      jeśli n jest nieparzyste:
            n = n+1
            A[n] = A[n-1]
      w = A[1]
      m = A[1]
      i = 1
      dopóki i<n wykonuj:
            jeżeli A[i] >= A[i+1]:
                  jeżeli A[i]>w
                        w = A[i]
                  jeżeli A[i+1]<m
                        m = A[i+1]
            jeżeli A[i]<A[i+1]
```

```
                jeżeli A[i+1]>w
                      w = A[i+1]
                jeżeli A[i]<m
                      m = A[i]
          i = i+2
    zwróć w jako największy,
          m jako najmniejszy element
```

Lider i dominanta

dominanta, lider

W tym rozdziale znowu rozważamy tablicę $A[1 \ldots n]$. Element, który w tej tablicy występuje najczęściej, nazywa się (głównie w statystyce) *dominantą*. Dodatkowo element jest *liderem*, jeśli występuje więcej niż $n/2$ razy – innymi słowy, jeśli więcej niż połowa elementów w tablicy ma tę samą wartość, tę wartość nazywamy liderem. W tablicy może być więcej niż jedna dominanta (np. w tablicy [1, 2, 3, 1, 2, 2, 1, 4] dominantą jest zarówno 1, jak i 2), ale jest co najwyżej jeden lider – dwa elementy jednocześnie nie mogą mieć ponad połowy wystąpień. W tablicy zawsze znajduje się przynajmniej jedna dominanta*, ale lidera może nie być w ogóle.

Najbardziej ogólny algorytm znajdujący dominantę razem z liczbą jej wystąpień opiera się na sortowaniu. Zauważmy, że przy sortowaniu tablicy takie same elementy grupują się razem (np. z tablicy [4, 1, 4, 2, 3, 1, 5, 4] robi się [1, 1, 2, 3, 4, 4, 4, 5]). Będziemy zatem przechowywać w pomocniczej zmiennej *aktualny* ten element, który napotkaliśmy poprzednio, a w zmiennej *ile_razy* – ile kopii tego elementu widzieliśmy. Jeśli nowy element jest równy aktualnemu, zwiększamy *ile_razy* o 1. W przeciwnym wypadku nowy element staje się *aktualnym*. Najczęściej występującym elementem będzie ten, dla którego *ile_razy* przyjęło w czasie algorytmu największą wartość:

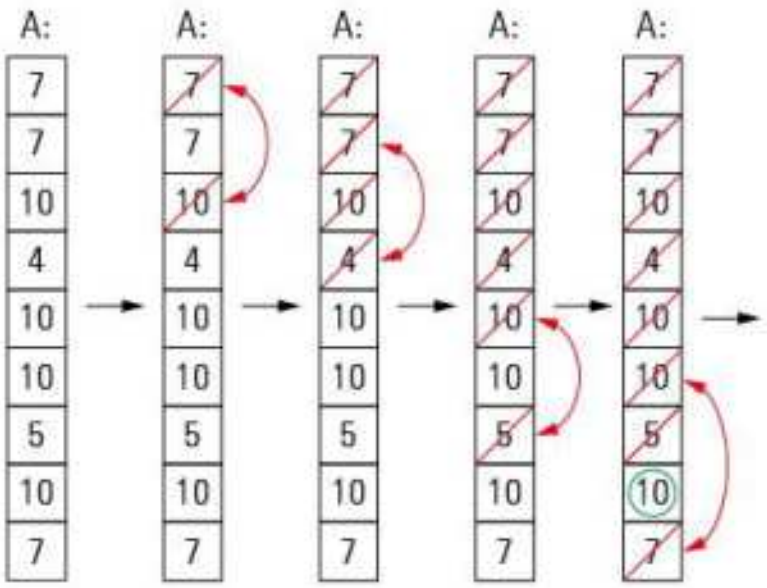
```
funkcja najczestszy(A)
      posortuj tablicę A niemalejąco
      aktualny = A[1]
      ile_razy = 1
      najczestszy = A[1]
      ile_najczestszy = 1
      dla i = 2, 3, ..., n
            jeżeli A[i] = A[i-1]
                  ile_razy = ile_razy + 1
                  jeżeli ile_razy > ile_najczestszy:
                        najczestszy = aktualny
                        ile_najczestszy = ile_razy
            w przeciwnym razie
                  aktualny = A[i]
                  ile_razy = 1
```

* Można również spotkać definicję dominanty jako elementu występującego najczęściej, ale tylko jeśli jest jedynym takim elementem (np. w ciągu [1,1,2,2,3,3] nie ma wtedy dominanty). Nasza definicja prowadzi do prostszego algorytmu, więc przyjmiemy ją na potrzeby tej lekcji.

Główna pętla algorytmu niewątpliwie wykonuje się w czasie liniowym – jest to pojedyncze przejście przez tablicę. Więcej czasu zajmuje zatem sortowanie, które trzeba wykonać w czasie $O(n \log n)$. Jeśli w tablicy jest wiele różnych elementów będących dominantą, algorytm znajdzie pierwszy z nich.

szukanie lidera w czasie liniowym

Jeżeli jednak szukamy lidera (a nie tylko dowolnej dominanty), istnieje szybszy (liniowy) algorytm rozwiązujący ten konkretny problem, opierający się na ciekawym tricku: wiemy, że element *x* jest liderem, jeśli jego wystąpień jest więcej niż wszystkich pozostałych razem wziętych. Oczywiście, rozpoczynając algorytm, nie znamy jeszcze elementu *x*. Możemy jednak przejrzeć A, a kiedy tylko natrafimy na dwa różne elementy, wykreślić je z tablicy. Być może jednym z nich jest *x*, ale po tej operacji nadal będzie więcej (lub tyle samo) *x*-ów niż pozostałych elementów – czyli ten sam *x* musi pozostać liderem. Powtarzamy tę operację wielokrotnie, aż pozostaną tylko identyczne elementy. Element, który pozostał, trzeba jeszcze sprawdzić – albo to on jest liderem, albo lidera nie ma w tablicy w ogóle. Znając kandydata, łatwo już sprawdzić liczbę jego wystąpień.

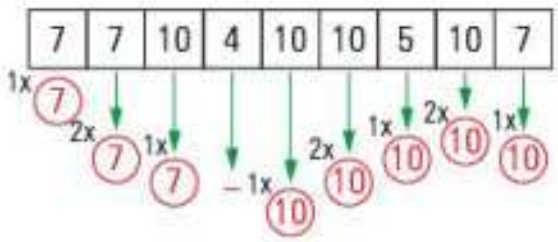


Wykreślamy kolejno pary różnych elementów z tablicy, najpierw (7,10), potem (7,4), (10,5) itd. Element 10 w żadnym momencie nie przestaje być liderem – aż w końcu pozostaje jako ostatni.

Możemy jeszcze bardziej uprościć sprawę: załóżmy, że chcemy przejść tablicę A od lewej do prawej i pamiętać wszystkie dotychczas napotkane niewykreślone elementy. Ponieważ każde dwa różne zostałyby natychmiast wykreślone, do zapamiętania zostaje w każdym momencie tylko jeden element – być może w kilku kopiach. Będziemy wartość tego elementu przechowywać w zmiennej *aktualny*, a liczbę kopii – w zmiennej *ile razy*. Napotkanie nowego takiego samego elementu (`JEŻELI A[i] = aktualny`) sprawia, że musimy tylko zwiększyć liczbę kopii (`ile_razy = ile_razy+1`). Za to napotkanie innego elementu powoduje, że wykreślamy jedną kopię aktualnego razem z nowo napotkanym elementem (`ile_razy = ile_razy-1`). Jeśli `ile_razy` spadnie do 0 i napotkamy nowy element, zastępuje on `aktualny`.

```
funkcja lider(A)
        aktualny = A[1]
        ile_razy = 1
        dla i = 2, 3, ..., n wykonuj:
```

```
        jeżeli A[i]=aktualny
              ile_razy = ile_razy+1
        w przeciwnym razie
              jeżeli ile_razy>0
                    ile razy = ile_razy-1
              w przeciwnym razie
                    aktualny = A[i]
                    ile_razy = 1
zlicz = 0
dla i = 1, 2, ..., n
      jeżeli A[i] = aktualny
            zlicz = zlicz+1
jeżeli zlicz > n div 2
      podaj wynik aktualny
w przeciwnym razie
      podaj wynik "nie ma lidera"
```



Stan zmiennych *aktualny* (czerwone kółko) i *ile_razy* po każdym kroku algorytmu opisuje niewykreślone elementy: po drugim kroku są dwa elementy 7 (*aktualny* = 7, *ile_razy* = 2), a na końcu pozostaje jedna 10.

Jaką przewagę ma nowy algorytm nad poprzednim? Po pierwsze, działa nieco szybciej – w czasie $O(n)$ robi dwa przebiegi tablicy. Poza tym potrzebuje bardzo niewiele dodatkowej pamięci – czyta dodatkową tablicę A, ale poza nią potrzebuje tylko trzech zmiennych liczbowych (*aktualny, ile_razy, zlicz*). Za wadę algorytmu można za to uznać fakt, że niezbyt często będziemy mieli okazję go zastosować – o ile dominanty szukamy stosunkowo często, o tyle lidera – rzadziej.

ZADANIA DO WYKONANIA

1. Zmodyfikuj funkcję `najczestszy(A)` tak, aby wypisywała wszystkie elementy będące dominantą.

2. Zastanów się, jak działałby następujący algorytm:

```
K = 20
funkcja losowy_lider(A)
powtarzaj K razy:
     x = losowy element spośród A[1..n]
     zlicz = 0
           dla i = 1, 2, .., n
                 jeżeli A[i] = x
                 zlicz = zlicz+1
     jeśli zlicz > n div 2
           zakończ i podaj wynik x
podaj wynik „nie ma lidera"
```

Jeśli w tablicy A jest lider, jaka jest szansa znalezienia go za jednym powtórzeniem (jeśli $K=1$)? Ile razy najlepiej powtórzyć algorytm (czyli ile przyjąć za K)?

PODSUMOWANIE LEKCJI

s. 29
- Standardowo element największy lub najmniejszy w tablicy można znaleźć za pomocą $n-1$ porównań.

s. 29
- Jeżeli jednak chcemy szukać jednocześnie największego i najmniejszego elementu, zamiast $2n-2$ można wykonać $3/2*n$ porównań.

s. 31
- Element najczęściej występujący (dominantę) można znaleźć w tablicy za pomocą sortowania w czasie $O(n \log n)$.

s. 32
- Jeżeli szukamy lidera – elementu, który występuje więcej niż $n/2$ razy – można użyć prostego, liniowego algorytmu opartego na wykreślaniu z tablicy par różnych elementów.

6. Tablice dynamiczne i listy wiązane, czyli kiedy tablica nie wystarcza

NA TEJ LEKCJI:

- dowiesz się, jak określić, czym jest tablica – niezależnie od języka programowania;
- przekonasz się, czym jest struktura danych;
- poznasz struktury tablicy dynamicznej (wektora) oraz listy wiązanej.

Uwaga!

Na tej lekcji będziemy porównywać ze sobą konstrukcje używane w różnych językach programowania (C++, Python, będą też inne przykłady języków). Możliwe, że nie znasz dokładnie wszystkich tych języków, ale nie jest to konieczne do zrozumienia treści. Bardzo zalecamy zapoznanie się z całością rozdziału!

Czym właściwie jest tablica?

Na tym etapie nauki prawdopodobnie już dobrze wiesz, czym jest tablica i jak jej używać w programowaniu. Prawie każdy język programowania używa tablic w jakiejś postaci. W C++, aby użyć tablicy, musimy wykonać instrukcję (*deklarację*) podobną do poniższej:

```
int tablica[100];
```

co oznacza „zarezerwuj miejsce w pamięci dla tablicy przechowującej 100 liczb całkowitych (int)". Podobne deklaracje występują w niektórych innych, zarówno popularnych, jak i historycznych, językach programowania:

Java: `int[] tablica = new int[100];`

C#: `int[] tablica = new int[100];`

Pascal: `var tablica : array[1...100] of Integer;`

Trochę inaczej kwestia wygląda np. w języku Python, w którym zmiennych nie trzeba deklarować, a zamiast tablicy używa się nieco bardziej skomplikowanego typu listy. Do Pythona za chwilę wrócimy. Spróbujmy zastanowić się, jakie są wspólne cechy, które mają tablice niezależnie od języka programowania:

(1) są to złożone typy danych, przechowujące więcej obiektów tego samego rodzaju;

(2) elementy są ustawione w kolejności i ponumerowane;

(3) w każdej chwili można odczytać i użyć dowolnego z elementów tablicy (np. `wypisz(tablica[7])`). Taka operacja jest pojedynczą instrukcją, działającą szybko (formalnie – w czasie stałym).

W języku C++ (a także np. w Pascalu) tablica ma stałą długość, której po deklaracji nie można już potem zmienić. Wiąże się to ze sposobem, w jaki tablica jest w praktyce realizowana: w momencie deklaracji program po prostu rezerwuje obszar pamięci tak, żeby pomieścił odpowiednią liczbę komórek, i ta rezerwacja pozostaje niezmieniona do końca „życia" tablicy.

Zwróć uwagę, że są też inne rzeczy, których „zwykła" tablica nie potrafi, takie jak wstawienie elementu w środek. Aby np. z tablicy [1,2,4,5] otrzymać tablicę [1,2,3,4,5], trzeba przesunąć połowę elementów, co oczywiście jest czasochłonne.

W tym rozdziale i kolejnym zostaną omówione różne sposoby przechowywania danych, analogiczne do tablicy, lecz pozwalające na inne sposoby dostępu do nich (np. lista będzie potrafiła wstawić lub usunąć środkowy element, ale ceną za to będzie trudniejszy dostęp do wskazanego miejsca). Takie sposoby przechowywania (często odpowiadające różnym typom danych w różnych językach programowania) nazywa się *strukturami danych*.

Tablica dynamiczna

Tablica dynamiczna to struktura danych bardzo podobna do tablicy, ale dodatkowo pozwalająca na zmiany długości w czasie wywoływania programu – w szczególności na dodawanie i usuwanie elementów z jej końca. Innymi słowy, tablica dynamiczna [1,2,3,4] powinna się łatwo i szybko (w czasie stałym) dać przerobić na tablicę [1,2,3,4,5].

◂ tablica dynamiczna

Uwaga!

Niestety, nigdy nie ustaliło się jednolite i powszechne (prawidłowe) nazewnictwo dla struktur i typów danych. W szczególności nazwa *tablica dynamiczna* jest tylko jedną z możliwych nazw dla tej struktury. Jak zaraz zobaczymy, w różnych językach programowania funkcjonują różne nazwy dla tablicy dynamicznej.

W języku C++ odpowiedni typ danych nazywa się wektorem (ang. *vector*). Zmienną można stworzyć w następujący sposób:

```
vector<int> tab(długość);
```

(np. `vector<int> tab(3)`).

Można też zadeklarować tablicę pustą (długości 0):

```
vector<int> tab;
```

Na zmiennej *tablica* można wykonywać operacje jak na „normalnej" tablicy. Na przykład dla tablicy o długości 3, wykonując:

```
tab[0] = 10; tab[1] = 20; tab[2] = 30;
```

sprawimy, że tablica ma postać [10,20,30]. Mamy też jednak możliwość wykonania instrukcji:

```
tab.push _ back(40);
```

Dostawi ona nowy element 40 na koniec – innymi słowy, zwiększy rozmiar o 1 i dopisze liczbę 40 jako ostatni element. Tablica [10,20,30] zmieni się w ten sposób w [10,20,30,40].

Z kolei funkcja:

```
tab.pop _ back();
```

usuwa z tablicy ostatni element i jednocześnie zmniejsza jej rozmiar (czyli z tablicy [10,20,30] zrobi się tablica [10,20]).

W języku Python sytuacja jest trochę inna: tablice są nazywane listami i są w szczególności tablicami dynamicznymi – pozwalającymi na zmianę rozmiaru. Jednak pełnią też inne funkcje, które chwilowo pozwolimy sobie pominąć.

Pustą listę w Pythonie tworzymy prostą instrukcją:

```
tab = []
```

Aby dopisać element na koniec, wykonujemy instrukcję `append()`, działającą analogicznie do push_back, np. `tab.append(10)`. Z kolei `pop()` usuwa ostatni element listy.

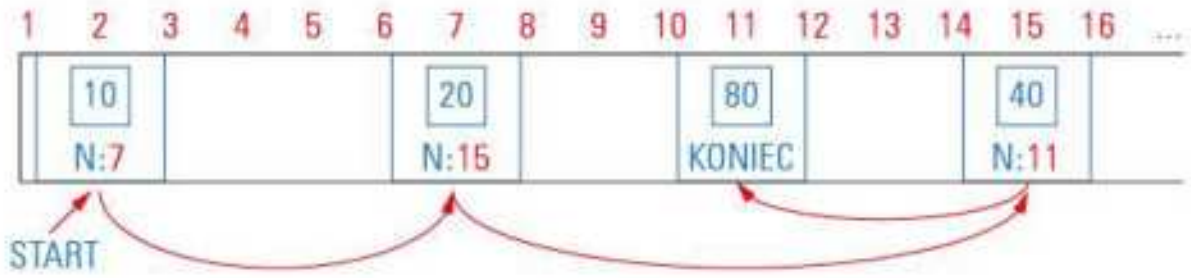
Warto zaznaczyć, że zarówno wektor w C++, jak i lista w Phytonie (a zwłaszcza ta ostatnia) mają zaimplementowanye nawet więcej operacji, np. wydzielenie fragmentu listy, ściąganie więcej niż jednego elementu itd. W tym rozdziale poprzestaniemy jednak na wyjaśnieniu najważniejszych, które mogą przydać się w rozwiązywaniu zadań na poziomie szkolnym.

Wspomnimy jeszcze o tym, jak tablicę dynamiczną realizuje się w praktyce. Przy tworzeniu tablicy rezerwuje się nieco więcej pamięci niż zadeklarował użytkownik. Kiedy tablica jest bliska wypełnienia, program tworzy nową, odpowiednio większą (typowo: dwa razy większą) i przenosi całą zawartość w nowe miejsce. To oczywiście zajmuje czas, przez co tablice dynamiczne zawsze działają trochę wolniej od standardowych. Jednak jeśli uważnie dobierze się moment, w którym jest realizowana operacja przeniesienia, strata czasowa nie będzie duża.

Lista wiązana

lista wiązana

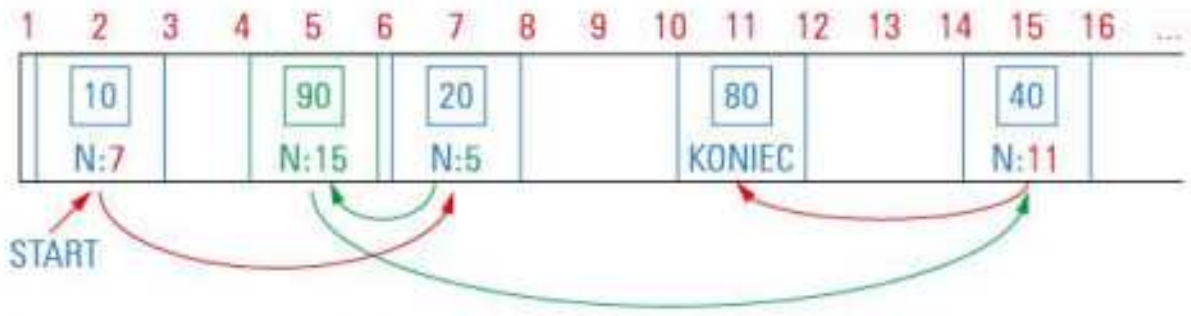
Alternatywnym podejściem do przechowywania danych jest *lista wiązana*. Wyobraźmy sobie, że przechowujemy w pamięci komputera ciąg elementów (np. [10, 30, 20, 40]), ale zamiast być w jednym miejscu, elementy są „rozrzucone" po pamięci operacyjnej. Aby dostęp do nich był możliwy, do każdego elementu dołączamy adres w pamięci następnego:



Przykładowa lista w pamięci komputera (przedstawiona poglądowo). Pierwszy element jest w komórce 2, ma wartość 10 oraz wskazuje, gdzie szukać następnego elementu (w komórce 7). Następny element ma wartość 20 i adres następnego (15). Pod adresem 15 jest kolejny element – 40 – oraz adres następnego (11), wreszcie w komórce 11 jest element 80 i informacja, że to koniec listy.

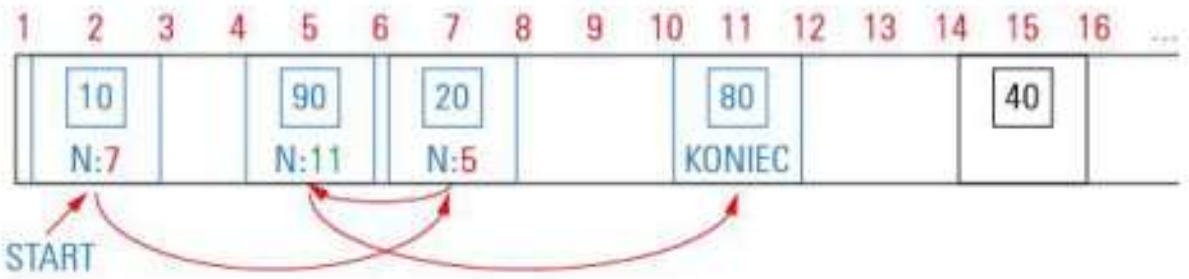
W bardziej zaawansowanej (i w praktyce częściej używanej) wersji każdy element pamięta nie jeden, lecz dwa adresy: następnego i poprzedniego elementu. Zauważmy, że niektóre operacje, które na tablicy były trudne, na liście przychodzą bardzo prosto i w naturalny sposób. Na przykład: jeśli do przedstawionej na powyższym rysunku listy [10,20,40,80] chcemy dodać nowy element 90 do środka (między 20 a 40), wystarczy:

- zarezerwować na niego nową komórkę pamięci (powiedzmy, że będzie to komórka 5);
- do komórki 5 wpisać wartość 90 oraz adres (15) komórki, która ma być następna;
- ustawić w komórce poprzedniej (7, która ma element 20) wskaźnik do następnego elementu na komórkę 5.



Do listy [10,20,40,80] został dodany nowy element (90) między 20 a 40.

W podobny sposób łatwo jest, za pomocą kilku operacji i bez ruszania większości pozostałych, usunąć jeden element – po prostu „przepinamy" wskaźniki tak, żeby ominąć go w tablicy, a następnie zwalniamy zajmowaną przez niego komórkę pamięci.



Z listy [10,20,90,40,80] usunięto element 40.

Widać zatem, że operacje „wstaw element w środek" i „usuń element ze środka" są stosunkowo łatwe. W odróżnieniu od tablicy lista nie potrafi szybko podać zadanego elementu (np. 20. w kolejności). Najszybszym sposobem jest ustawienie się na początku listy i przejście odpowiednią liczbę razy do kolejnego elementu, co oczywiście zajmuje znacznie więcej czasu.

Implementacja listy wiązanej wymaga użycia bezpośrednich adresów w pamięci (wskaźników), co wykracza poza programowanie obecne w szkole średniej. Zatem poprzestaniemy na wyjaśnieniu idei jej działania oraz rozwiązania przykładowego zadania za jej pomocą – tzw. zagadki Flawiusza.

Przykład: problem Flawiusza

Zagadka zwana *problemem Józefa Flawiusza* wygląda następująco:

Pewna liczba osób N ustawia się w koło i odlicza do 3 (pierwsza osoba mówi 1, druga 2, trzecia 3, czwarta 1 i tak dalej). Każda osoba, która powie „3", odpada z gry i wychodzi z koła (które odpowiednio się zmniejsza). Kto pozostanie jako ostatni?

Istnieje kilka wersji zagadki, z różnymi sposobami odliczania (np. do 5 lub do 7) albo z szukaniem dwóch ostatnich osób zamiast jednej. Niektórym towarzyszą różne legendy odwołujące się do faktów historycznych (zwykle dość makabrycznych, dlatego ich tutaj nie cytujemy). Sformułujmy problem w wersji bardziej formalnej: nadajmy osobom numery od 1 do *N* i ustalmy, że przechodzimy przez ciąg elementów, wykreślając co trzeci napotkany, tak jak w zagadce. Naszym celem jest znalezienie elementu, który pozostanie jako ostatni.

Struktura listy, możliwość szybkiego przechodzenia do następnego jej elementu oraz wykreślenia elementu dają nam stosunkowo łatwy algorytm rozwiązujący ten problem. Będziemy potrzebować:

- listy wiązanej *L*;
- zmiennej *a*, która w każdym momencie będzie przechowywać adres jednej z komórek *L*, oraz dwóch funkcji:
  - `następny(a)`: podaje element listy następny po wskazywanym przez *a*;
  - `usuń(a)`: usuwa element listy pod adresem *a* i przechodzi do następnego.

```
L - lista wiązana
dla i = 1, 2, ..., N
```

```
        dodaj i na koniec listy L
a = początek listy L
powtarzaj N-1 razy:
        a = następny(a)
        a = następny(a)
        usuń (a)
wypisz pierwszy element L
```

ZADANIA DO WYKONANIA

1. Przeprowadź symulację problemu Flawiusza dla $N = 10$. Który element pozostanie jako ostatni?

2. Dane są dwie listy wiązane *L1* i *L2*. W jaki sposób można najefektywniej połączyć je ze sobą, czyli stworzyć listę przechowującą elementy ich obu?

PODSUMOWANIE LEKCJI

- Tablica dynamiczna ma zmienny rozmiar – pozwala dopisywać i usuwać elementy z końca tablicy. s. 35
- Lista wiązana przechowuje elementy w różnych miejscach pamięci, a każdy element trzyma adres następnego (i zwykle również poprzedniego). s. 37

7. Stos i kolejka, czyli struktury proste i bardzo użyteczne

NA TEJ LEKCJI:

- nauczysz się, czym są struktury stosu i kolejki;
- poznasz przykłady ich zastosowań;
- dowiesz się, jak użyć ich w swoich programach.

Stos – definicja

Z tematu *Tablice dynamiczne i listy* pamiętamy, że *struktura danych* to pewien sposób ich przechowywania i organizacji. Wyobraźmy sobie bardzo prostą strukturę danych, która przechowuje elementy i organizuje je, „kładąc" jeden na drugim na kształt stosu. Każdy nowy element trafia na szczyt stosu i przykrywa wszystkie poprzednie – tak, że zawsze mamy dostęp tylko do najpóźniej położonego elementu. Mówiąc bardziej informatycznym językiem, mamy do dyspozycji tylko trzy operacje:

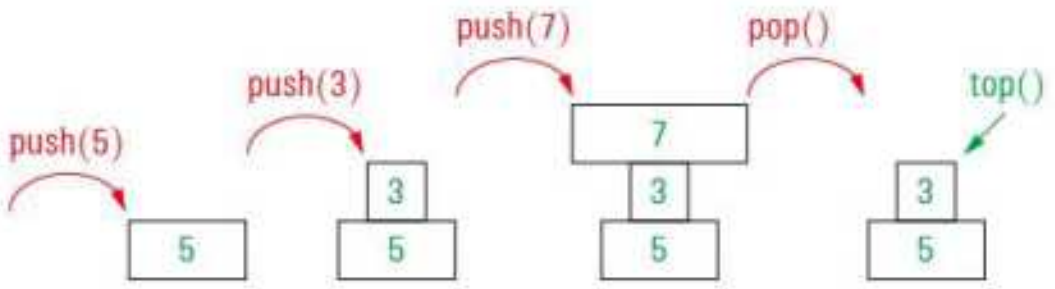
◂ struktura stosu

- `push(x)`: dorzuć nowy element do przechowywanych;
- `top()`: podaj najpóźniej dorzucony element (na szczycie stosu);
- `pop()`: usuń najpóźniej dorzucony element.

Na przykład jeśli zaczniemy od pustej struktury, sekwencja:

- `push(5)`
- `push(3)`
- `push(7)`
- `pop()`
- `wypisz(top())`

wypisze liczbę 3, jak widać na rysunku poniżej:



Przykładowa sekwencja operacji na stosie

Taka struktura nazywana jest *stosem* (ang. *stack*). Czasami jest nazywana też LIFO, od angielskiego akronimu *Last In, First Out* (czyli „przyszedł ostatni, wychodzi pierwszy"). Typowo od stosu oczekuje się (i praktycznie wszystkie implementacje tak mają), że złożoność wszystkich operacji będzie stała.

Zauważmy, że rozmiar stosu (liczba przechowywanych elementów) oczywiście zmienia się w czasie jego działania – struktury, które mają taką własność, nazywamy *dynamicznymi* (poznane wcześniej tablica dynamiczna i lista wiązana oczywiście też są strukturami dynamicznymi). Bywają różne warianty stosu – czasem dopuszczamy jeszcze czwartą operację – `empty()`, która zwraca „prawda"/„fałsz" w zależności od tego, czy stos jest pusty, czy pełny. Czasem operacje `pop()` i `top()` łączy się w jedną i zakłada, że `pop()` zarówno zwraca ostatni element, jak i usuwa ten element ze stosu.

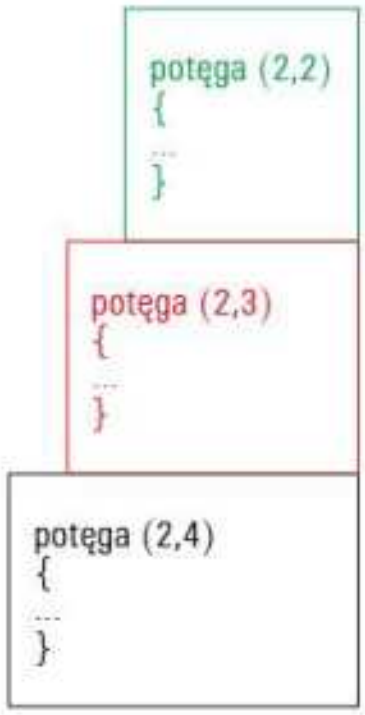
Stos – zastosowania

stos – zastosowania

Stos jest bardzo prostą strukturą danych, ale ma zaskakująco wiele praktycznych zastosowań. Na przykład każdy progr am, jaki napiszesz i uruchomisz, ma tak zwany stos wywołań, na którym zapisuje, które funkcje/procedury są aktualnie wywoływane. Jest to szczególnie ważne przy wywołaniach rekurencyjnych – przypomnij sobie taką (znaną z rozdziału o rekursji) funkcję:

```
funkcja potęga(a, n)
        jeżeli n = 0, zwróć 1
        w przeciwnym razie:
                zwróć a * potega(a, n-1)
```

Jeśli na przykład wywołamy `potega(2,4)`, ta funkcja w pewnym momencie wywoła samą siebie jako `potega(2,3)`, a ta z kolei wywoła `potega(2,2)`. W tym momencie działania program ma zapamiętane wszystkie te wywołania:



Stos wywołań funkcji w przypadku wywołania rekurencyjnego

Oczywiście na rysunku zaznaczyliśmy tylko samo wywołanie, ale razem z nim muszą być zapisane inne informacje (np. w którym miejscu programu nastąpiło, czyli gdzie należy potem powrócić). Tym sposobem można odtworzyć, że po zakończeniu aktualnie wykonywanej funkcji `potega(2,2)` program ma ściągnąć ją ze stosu i wrócić do poprzednio wywoływanej `potega(2,3)`.

Być może pamiętasz z poprzednich lat nauki odwrotną notację polską, czyli alternatywny (i wygodniejszy dla maszyn) sposób zapisu działań arytmetycznych. W skrócie: zamiast *x* + *y* zapisuje się *x y* +, zamiast *x* * *y* zapisuje się *x y* * i podobnie dla innych działań. Tak więc (1+5)*(2+3) zapisujemy jako 1 5 + 2 3 + *.

Mając dane wyrażenie w odwrotnej notacji polskiej, łatwo obliczyć jego wartość z użyciem stosu następującym algorytmem: przechodzimy wyrażenie od lewej do prawej, każdą napotkaną liczbę odkładamy na stos, natomiast przy każdym działaniu bierzemy dwie ostatnie liczby ze stosu, wykonujemy na nich działanie, a wynik odkładamy z powrotem na stos. Na przykład: dla 1 5 + 2 3 + * algorytm wykona następujące kroki:

- napotka 1 i odłoży na stos (aktualny stan stosu: [1]);
- liczbę 5 również odłoży na stos (stan stosu: [1, 5]);
- napotka +, więc zdejmie ze stosu dwie ostatnie liczby (5 i 1), doda je, a wynik 6 odłoży na stos (stan stosu: [6]);
- liczbę 2 odłoży na stos ([6, 2]);
- liczbę 3 odłoży na stos ([6, 2, 3]);
- napotka +, więc ściągnie dwie ostatnie liczby ze stosu (3, 2), doda je, a wynik 5 trafi na stos ([6, 5]);
- napotka *, więc ściągnie dwie ostatnie liczby ze stosu i je pomnoży (6*5 = 30), wynik odłoży na stos ([30]).

Na końcu działania algorytmu na stosie powinna pozostać jedna liczba – ostateczny wynik działania. Dla podanego przykładu będzie to 30.

Stos – implementacja

Jak użyć stosu w swoim programie? Po pierwsze zauważmy, że bardzo łatwo go zrobić, jeśli ma się tablicę dynamiczną z poprzedniego rozdziału – o niej już wiemy, że pozwala na dopisanie elementu na koniec, a także na usunięcie elementu z końca. Podobną własność ma lista wiązana – w niej również da się łatwo dostawić element na koniec lub też usunąć ostatni.

W języku Python stos najłatwiej uzyskać właśnie w ten sposób: dla listy S operacja `S.append(x)` działa jak `push()`, wywołanie *S*[–1] działa jak `top()`, a `S.pop()` – zgodnie ze swoją nazwą – działa tak, jak powinno działać `pop()` stosu, i ściąga ostatni element.

Z kolei w C++ można w analogiczny sposób użyć wektora, ale jest też specjalny typ `stack<int>` (lub `stack<string>`, `stack<double>`,..., jeśli na stosie mają być napisy, liczby rzeczywiste lub inne dane), który jest zaprojektowany właśnie jako implementacja stosu. Na elemencie *S* typu `stack<dowolny_typ>` można wykonywać operacje `S.push()`, `S.pop()` i `S.top()`, a także `S.empty()` sprawdzającą pustość – o znaczeniu takim, jak podano na początku rozdziału.

Kolejka

struktura kolejki

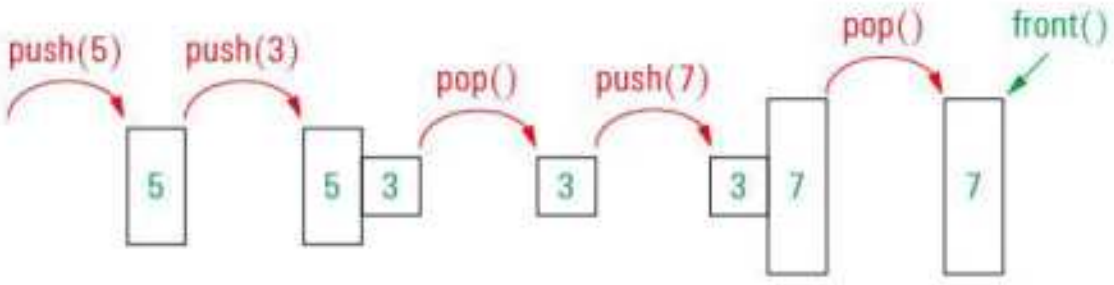
Kolejka (ang. *queue*) to struktura, do której również dokładamy elementy, podobnie jak do stosu. Reguła dostępu do nich jest jednak odwrotna niż dla stosu: zawsze można obejrzeć (i zdjąć) tylko najwcześniejszy z dołożonych elementów. Innymi słowy, elementy schodzą z kolejki w tej samej kolejności, w jakiej na nią trafiły, co uzasadnia nazwę struktury. Bardziej formalnie – mamy do dyspozycji trzy operacje:

- `push(x)`: dokłada element do kolejki;
- `front()`: podaje element, który przyszedł najwcześniej (na początku kolejki);
- `pop()`: usuwa najwcześniejszy element z kolejki.

Na przykład sekwencja:

- `push(5)`
- `push(3)`
- `pop()`
- `push(7)`
- `pop()`
- `wypisz(front())`

wypisze liczbę 7:



Przykładowa sekwencja operacji na kolejce

Podobnie jak w przypadku stosu, funkcjonują też inne warianty kolejki – na przykład często spotykana jest operacja `empty()` zwracająca *prawda*, jeśli kolejka jest pusta, i *fałsz* – w przeciwnym wypadku. Czasem nie ma operacji `front()`, a zamiast niej `pop()` zwraca jako wynik pierwszy (właśnie usunięty element). Czasem też spotykane są, zamiast *push* i *pop*, nazwy *enqueue* i *dequeue* (czyli dosłownie „zakolejkuj" i „odkolejkuj"). Dla samej struktury kolejki funkcjonuje też alternatywna nazwa FIFO, co jest akronimem od *First In, First Out* („pierwszy przyszedł, pierwszy wychodzi"). Tak samo jak przy stosie, zwykle oczekujemy, że podstawowe operacje kolejki będą działać w czasie stałym.

Zastosowania kolejki

Kolejki również mają wiele praktycznych zastosowań. Na przykład: wyobraźmy sobie serwer WWW, który musi obsługiwać żądania od łączących się z nim klientów – na ogół żądania w postaci „prześlij mi stronę internetową". Żądania nie przychodzą równomiernie: czasem kilka z nich przyjdzie naraz, praktycznie w jednym momencie, a czasem jest chwilowa przerwa. Typowo w takiej sytuacji serwer ma kolejkę żądań, których jeszcze nie zrealizował. Każde przychodzące żądanie trafia na koniec tej kolejki (co działa błyskawicznie), a jeśli serwer ma chwilę, obsługuje żądania z kolejki, poczynając od najwcześniejszych. Dla każdego z nich przygotowuje odpowiednią stronę, wysyła ją do klienta, po czym usuwa to żądanie z kolejki i czyta następne.

W algorytmice najbardziej klasycznym zastosowaniem kolejki jest algorytm szukania najkrótszej drogi (jego ideę poznasz w rozdziale poświęconym grafom).

Jak użyć kolejki w programie?

W języku C++ jest typ `queue<dowolny_typ>` (np. `queue<int>`). Zmienne tego typu działają tak jak opisane wyżej: mają operacje `push()`, `front()` i `pop()` (a także `empty()`). Podobna klasa `queue` występuje też w języku Python.

ZADANIA DO WYKONANIA

1. Załóżmy, że masz do dyspozycji tablicę o stałej długości $S[1 \ldots 100]$. Zapisz funkcje `push()`, `pop()` i `top()` tak, aby działały jak operacje stosu (`push()` dokładało nowy element, `pop()` ściągało ostatni włożony element itd.).

2. Zaimplementuj w analogiczny sposób kolejkę – mając do dyspozycji tylko tablicę $K[1 \ldots 100]$, w której przechowywane będą elementy kolejki. Użyj dwóch pomocniczych zmiennych *głowa* i *ogon* – *głowa* będzie przechowywać miejsce, gdzie jest początek kolejki, *ogon* – miejsce w tablicy, gdzie jest jej ostatni element.

PODSUMOWANIE LEKCJI

s. 39
- Struktura stosu pozwala na dokładanie i zdejmowanie elementów – zawsze jest zdejmowany najpóźniej włożony element.

s. 40
- Stos używany jest m.in. do przechowywania wywołań funkcji, a także w algorytmie obliczania wyrażeń w odwrotnej notacji polskiej.

s. 42
- Struktura kolejki działa analogicznie, ale zawsze jest zdejmowany najwcześniej włożony element. Kolejka jest używana m.in. do obsługi żądań serwera, a także w algorytmie szukania najkrótszej ścieżki.

8. Grafy, czyli co mają wspólnego sieć społecznościowa i paryskie metro

NA TEJ LEKCJI:

- dowiesz się, czym jest graf;
- poznasz różne przykłady grafów, w tym wiele praktycznych;
- dowiesz się, jak reprezentować graf w pamięci komputera.

Sieci połączeń

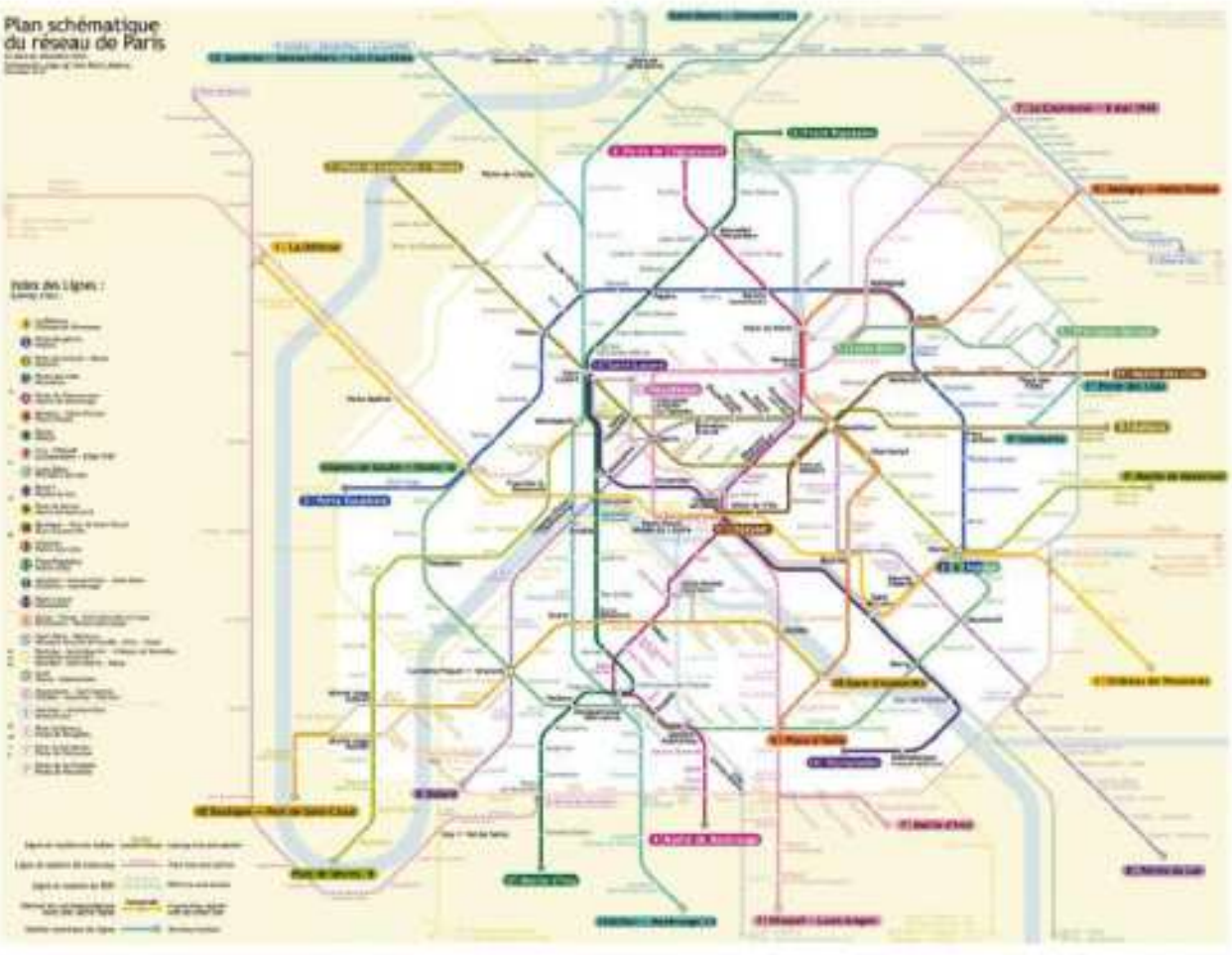
Prawie każdy miał do czynienia z jakąś siecią społecznościową albo przynajmniej wie, jak taka sieć wygląda. W sieci jest zarejestrowana pewna liczba osób, których część deklaruje się jako znajomi.

Przykładowa sieć społecznościowa

Co taki obrazek ma wspólnego np. z siecią połączeń lotniczych w Stanach Zjednoczonych?

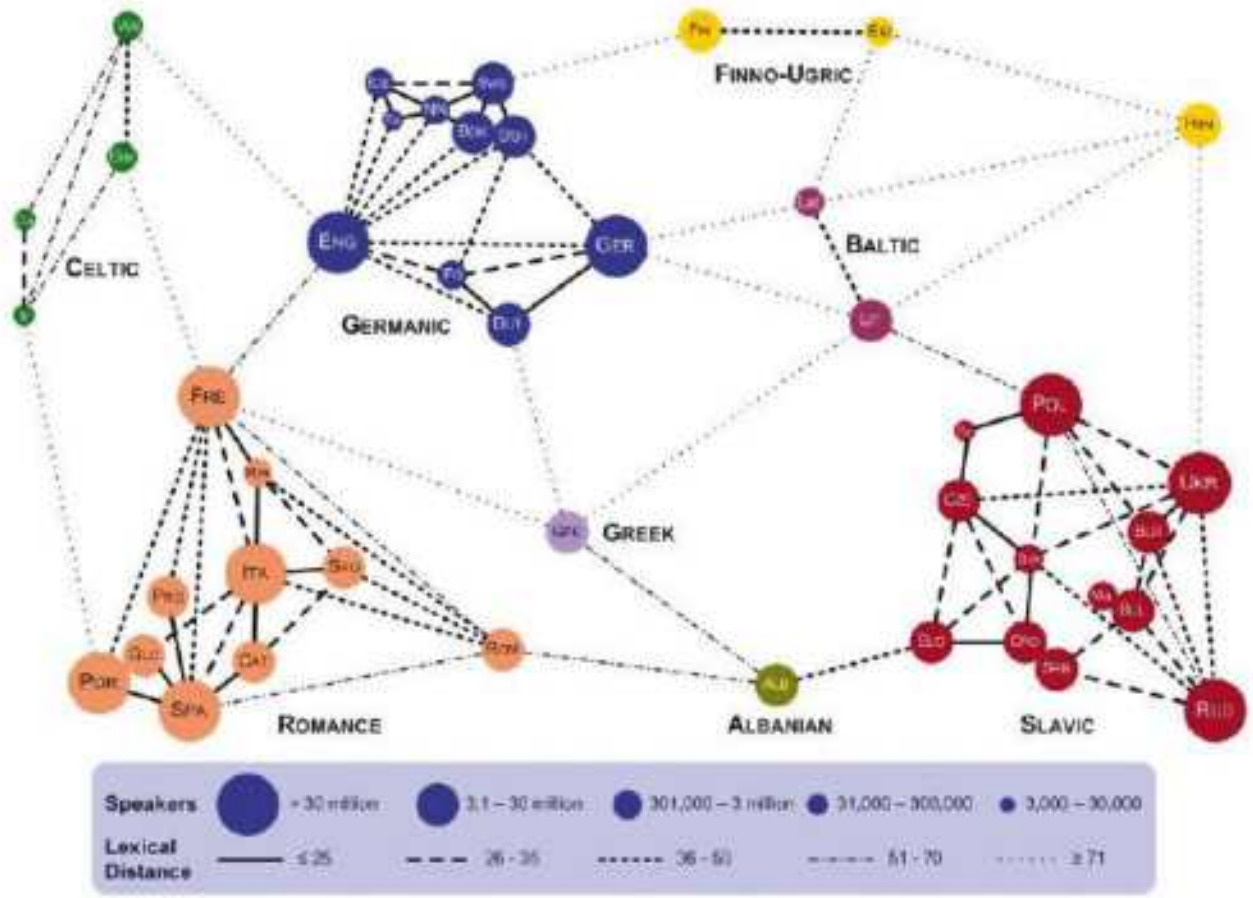
Połączenia lotnicze (nie wszystkie) w USA, 2002. Źródło: airlinemaps

Albo z mapą paryskiego metra?



Mapa sieci metra w Paryżu, źródło: Wikipedia

Albo z siecią pokrewieństwa między różnymi językami używanymi w Europie?



Obliczone odległości między różnymi językami używanymi w Europie (np. polski i czeski są blisko siebie, ale np. niemiecki i węgierski – bardzo odległe). Źródło: Wpcomstaging

We wszystkich tych sytuacjach mamy do czynienia z jakimś obiektami (osoby, miasta, stacje metra, języki...). Niektóre z obiektów są ze sobą połączone, a inne – nie. Taką strukturę nazywamy *grafem*. Mówiąc bardziej formalnie, graf to zbiór obiektów (nazywanych *wierzchołkami* grafu) oraz połączeń między nimi, które tradycyjnie nazywa się *krawędziami* grafu.

graf

Czasem umawiamy się, że ważny jest kierunek połączenia – może istnieć połączenie A→B, ale nie istnieć B→A. W takiej sytuacji mówimy, że graf jest *skierowany*. Przykładami mogą być drogi jednokierunkowe albo sieć społecznościowa, taka jak Instagram, w której możemy kogoś obserwować, ale to nie znaczy, że on obserwuje nas.

Dla danego wierzchołka *X* wszystkie wierzchołki, do których prowadzi krawędź z *X*, nazywamy *sąsiadami X* (w sieci społecznościowej sąsiedzi osoby *X* to po prostu jej znajomi).

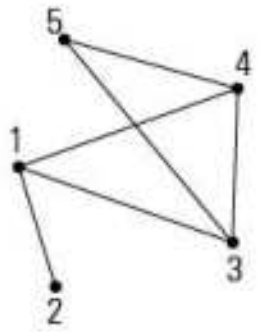
Z podanych powyżej przykładów widać, że grafy bardzo często pojawiają się w problemach informatycznych. Serwisy społecznościowe muszą szybko odpowiadać na pytania typu „Czy podane dwie osoby są znajomymi? Ilu mają wspólnych znajomych?". W sieci połączeń lotniczych lub drogowych możemy z kolei pytać „Jaka jest najkrótsza droga z jednego miasta do drugiego?".

Grafy w pamięci komputera

Załóżmy przez chwilę, że uruchomiliśmy własną sieć społecznościową. Musimy oczywiście przechowywać dane użytkowników. Jednak – co w tym rozdziale interesuje nas najbardziej – musimy też pamiętać, które pary użytkowników są znajomymi.

grafy w pamięci komputera

Innymi słowy, musimy przechować w pamięci komputera cały graf, wszystkie jego wierzchołki i krawędzie. Dwa najbardziej rozpowszechnione sposoby to *macierz sąsiedztwa* i *listy sąsiedztwa*.



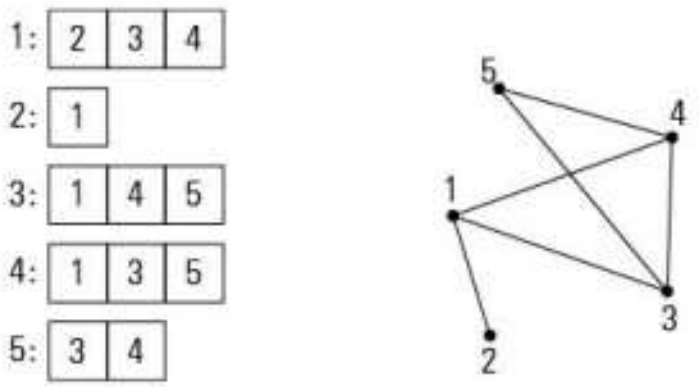
Macierz sąsiedztwa to duża, dwuwymiarowa tablica, w której każdy wierzchołek grafu ma swój wiersz i swoją kolumnę. Na przecięciu a-tego wiersza oraz b-tej kolumny piszemy 1, jeśli wierzchołki a i b są połączone krawędzią, 0 – w przeciwnym wypadku:

	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	0	0
3	1	0	0	1	1
4	1	0	1	0	1
5	0	0	1	1	0

Przykładowy graf i jego macierz sąsiedztwa. Dla każdego wierzchołka, w jego wierszu są jedynki w tych kolumnach, dla których ten wierzchołek ma połączenia (np. sąsiadami 3 są 1, 4 i 5, a więc w wierszu 3 jedynki są w pierwszej, czwartej i piątej kolumnie).

Jest to prosty sposób przechowywania grafu, ale ma bardzo istotną wadę: duże zużycie pamięci. Dla 10 wierzchołków potrzebujemy tablicy 10×10, czyli o 100 komórkach, a ogólnie, jeżeli graf ma N wierzchołków, potrzebne będzie $N \times N = N^2$ komórek. Dla większych N jest to w praktyce zupełnie nieosiągalne. Na przykład: serwis Facebook ma w momencie tworzenia tej książki ponad 2 miliardy użytkowników, dla których pełna tablica $N \times N$ zużyłaby całą dostępną na świecie pamięć.

Znacznie wygodniej jest skorzystać z faktu, że przeciętny użytkownik ma stosunkowo niewielu znajomych (od kilkuset do kilku tysięcy). Struktura list sąsiedztwa polega na pamiętaniu, dla każdego wierzchołka grafu, wszystkich jego sąsiadów przechowywanych w strukturze o zmiennej długości (na liście wiązanej lub – obecnie częściej – w tablicy dynamicznej).



Ten sam graf przechowywany jako lista sąsiedztwa – dla każdego wierzchołka jest stworzona tablica dynamiczna, w której są przechowywani jego sąsiedzi (np. sąsiadami wierzchołka 4 są 1, 3 i 4).

Skoro każdy wierzchołek przechowuje „swoje" krawędzie, widać, że taka struktura potrzebuje łącznie tyle pamięci, ile w grafie znajduje się krawędzi. Jest więc pod tym względem najbardziej efektywna, jak to możliwe. Trudniej jednak odpowiedzieć na pytanie, czy konkretne dwa wierzchołki (a,b) są połączone krawędzią – trzeba do tego w jakiś sposób przeszukać listę sąsiadów *a* albo listę sąsiadów *b*.

Jaki sposób najlepiej wybrać do przechowywania grafu – macierz czy listy sąsiedztwa? Oczywiście, to zależy od sytuacji. Jeżeli – podobnie jak w sieci społecznościowej – wierzchołki mają raczej niewielu sąsiadów, zdecydowanie należy użyć list sąsiedztwa. Jeżeli z kolei w grafie jest bardzo dużo krawędzi (blisko maksymalnej możliwej wartości *N*), macierz sąsiedztwa może być lepszym (szybszym w działaniu) rozwiązaniem. W praktyce częściej spotyka się grafy z małą liczbą krawędzi, a więc prawdopodobnie częściej będziesz używać list sąsiedztwa niż macierzy.

ZADANIA DO WYKONANIA

1. Narysuj graf, w którym wierzchołki to państwa: Polska, Niemcy, Francja, Czechy, Słowacja i Austria, a krawędź między wierzchołkami jest wtedy, kiedy dwa państwa graniczą ze sobą. Stwórz jego macierz sąsiedztwa i listę sąsiedztwa.
2. Oszacuj (może być z dużym przybliżeniem), ile pamięci potrzeba, aby przechowywać w pamięci graf połączeń Facebooka, przy założeniu, że są 2 miliardy użytkowników, a przeciętny użytkownik ma 300 znajomych. Graf chcemy przechowywać jako listy sąsiedztwa w tablicach dynamicznych.

PODSUMOWANIE LEKCJI

s. 46

- Graf to zbiór wierzchołków oraz zbiór krawędzi łączących wierzchołki. Grafy mogą być skierowane (krawędzie są jednokierunkowe) lub nieskierowane (krawędzie są dwukierunkowe).

s. 47

- Graf można przechowywać w pamięci komputera w postaci macierzy sąsiedztwa (kwadratowa tablica, w której 0/1 oznacza brak/istnienie krawędzi) lub list sąsiedztwa (dla każdego wierzchołka w liście lub tablicy dynamicznej są przechowywani wszyscy jego sąsiedzi).

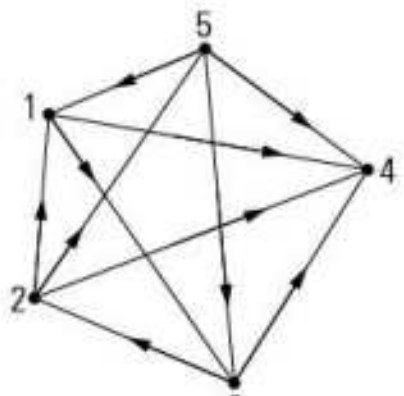
9. Wyszukiwanie idola, czyli o tym, jak spóźniliśmy się na turniej tenisowy

NA TEJ LEKCJI:

- dowiesz się, czym jest turniej i jaki specjalny wierzchołek nazywa się idolem;
- poznasz algorytm wyszukiwania idola.

Grafy-turnieje, czyli czym jest idol

Przeanalizujmy graf, który jest skierowany, co oznacza, że wiemy, w jakim kierunku prowadzi każda krawędź. Co więcej, załóżmy, że każdą parę wierzchołków łączy dokładnie jedna zorientowana krawędź:



Przykładowy graf-turniej. Na przykład: z rysunku wynika, że 1 wygrał z 2 i 5, a przegrał z 3 i 4. Idolem jest wierzchołek 4, do którego wszystkie krawędzie wchodzą, a żadna nie wychodzi.

Możemy sobie wyobrazić, że wierzchołki grafu to zawodnicy, którzy rozegrali między sobą mecze w tenisa (warcaby, Starcraft lub jakąkolwiek inną grę, w której nie ma remisów), przy czym każdy z każdym grał dokładnie raz. Krawędź A→B w grafie oznacza teraz, że A przegrał z B. (Ciekawostka: w informatyce faktycznie taki graf nazywa się *turniejem*).

◂ turniej

Zagadka, którą chcemy rozwiązać w tym rozdziale, jest następująca: spóźniliśmy się na turniej i nie znamy jego wyników (czyli nie wiemy, jakie są krawędzie w grafie). Słyszeliśmy plotkę, że pewien zawodnik wygrał wszystkie swoje spotkania, czyli pokonał wszystkich pozostałych zawodników (innymi słowy, szukamy wierzchołka w grafie takiego, że wszystkie krawędzie prowadzą „do niego", a żadna nie prowadzi „od niego"). Jesteśmy na spotkaniu z zawodnikami po zakończeniu turnieju i jedyne pytanie, jakie wypada nam im zadać, to: „Czy A wygrał, czy przegrał z B?". Aby nie zdradzić się ze swoją niewiedzą, chcemy takich pytań zadać możliwie najmniej. Jak znaleźć zawodnika, który pokonał wszystkich?

Czasem taki problem nazywa się *wyszukiwaniem idola* (jeśli zamiast „A wygrał z B" użyjemy metafory „B jest fanem A w mediach społecznościowych", widać, skąd nazwa się wzięła). W literaturze naukowej częściej mówi się o nim

„szukanie w grafie *ujścia*" (ang. *sink*), czyli wierzchołka takiego, że wszystkie krawędzie „płyną w jego kierunku".

idol, ujście

Algorytm znajdowania idola

Wróćmy do zagadki: jak znaleźć idola, zadając jak najmniej pytań? Oczywiście możemy zapytać o wszystkie możliwe pary. Jak napisaliśmy wcześniej, jest ich $n(n-1)/2$. Niestety to bardzo dużo pytań, np. dla $n = 50$ będzie ich 1225. Można jednak rozegrać całą sytuację znacznie lepiej, tak aby zadanych pytań nie mogło być więcej niż $2n$. Algorytm opiera się na bardzo prostym pomyśle, podobnym do wyszukiwania lidera opisanego w rozdziale 5. (*Specjalne elementy...*): dowolna odpowiedź na każde możliwe pytanie o parę zawodników (x, y) wyklucza jednego z nich z dalszych rozważań – jeśli wygrał x, to y nie może być idolem. Jeśli wygrał y, idolem nie może już być x.

Aby zamienić to na prawidłowo opisany algorytm, oznaczmy w wyobraźni zawodników liczbami $1, 2, \ldots, n$. Zapytajmy najpierw, czy 1 pokonał 2. Jeśli tak, to wiemy, że 2 nie jest idolem, za to 1 wciąż nim może być. Przetestujmy teraz parę (1, 3), a jeśli znowu 1 będzie zwycięzcą, to (1, 4) i tak dalej. Jeżeli uda się w ten sposób ustalić, że 1 pokonał wszystkich, wiemy już, że to on jest idolem. Z reguły jednak tak nie będzie i znajdziemy zawodnika numer p, z którym 1 przegrał, np. weźmy $p = 6$. Czyli 1 wygrał z 2, 3, 4, 5, ale przegrał z 6. Teraz wiemy już, że 1 nie jest idolem (bo przegrał z 6), ale też nie są nim 2, 3, ..., 5 (bo wiemy, że przegrali z 1).

Możemy więc chwilowo pominąć wszystkich graczy od 1 do 5 i kontynuować cały algorytm, poczynając od gracza 6 – będziemy teraz pytać o (6, 7), potem (6, 8) itd. Jeśli 6 przegra z jakimś innym graczem o numerze r, będziemy wiedzieli, że możemy pominąć wszystkich zawodników do $r-1$ włącznie i zacząć dalsze działanie od r.

algorytm znajdowania idola

Całość algorytmu będzie wyglądała następująco: przez cały czas utrzymujemy numer k aktualnego kandydata na idola i kolejno sprawdzamy go z wierzchołkami o wyższych numerach. Zmienna p to zawodnik, z którym porównujemy k. Na początku p jest równe 2, potem 3 itd., aż znajdziemy p, z którym k przegra, albo też dojdziemy do końca. Jeśli odkryjemy, że kandydat k z kimś przegrał, zastępujemy go przez jego pogromcę i postępujemy dalej tak samo.

```
funkcja znajdz_idola()
        k = 1
        p = 2
        dopóki p <= n wykonuj:                          [*]
                jeżeli k wygrał z p
                        p = p+1
                w przeciwnym razie
                        k = p
                        p = k+1
        dla j = 1, 2,..., k-1                                  [**]
                jeśli k przegrał z j
                        zakończ i podaj wynik „NIE MA IDOLA"
        podaj wynik k
```

Pętla, którą opisaliśmy powyżej, jest oznaczona przez [*]. Zakończy się w sytuacji, w której dla pewnego kandydata k dojdziemy z p do końca pętli, czyli dowiemy się, że k wygrał z $k+1$, $k+2$, ..., n. To jeszcze nie oznacza, że k jest idolem! Ale na pewno wiemy, że żaden inny wierzchołek nim nie jest – wszyscy o numerach mniejszych od k z kimś przegrali (bo zwiększamy wartość k wtedy, kiedy wiemy, że z kimś przegrał), a wszyscy o numerach większych przegrali z k.

Aby więc zweryfikować, czy k faktycznie jest idolem, na koniec algorytmu musimy jeszcze sprawdzić (część [**]), czy k na pewno zwyciężył wszystkich zawodników o mniejszych numerach. Robi to ostatnia pętla w algorytmie – jeżeli któryś zawodnik o numerze $j < k$ wygrał z k, wiemy ostatecznie, że nikt nie jest idolem. Jeżeli k pokonał wszystkich, idolem musi być on.

Pewnym wyzwaniem mogłoby się wydawać policzenie, ile operacji wykonał ten algorytm – ustaliliśmy już na początku, że jako operacje liczymy zapytania do par zawodników („czy x wygrał z y"). Intuicyjnie widać, że skoro każde zapytanie eliminuje jednego z zawodników, to część [*] może wykonać ich co najwyżej $n-1$ (po tylu zapytaniach musi zostać już tylko jeden zawodnik). Bardziej formalnie – zauważmy, że każde okrążenie pętli [*] zwiększa wartość zmiennej p o 1, a więc okrążeń będzie co najwyżej $n-1$.

Z kolei druga część algorytmu [**] to zwykła pętla od 1 do $k-1$, a więc wykonuje w najgorszym razie $n-1$ zapytań. Całkowita liczba zapytań nie przekroczy zatem $2n-2$.

ZADANIA DO WYKONANIA

1. Narysuj przykładowy turniej dla 6 graczy – wybierz wyniki meczów przypadkowo – i zbuduj na nim symulację powyższego algorytmu. Ile dokładnie wykonasz zapytań?
2. Rozważmy wersję algorytmu znajdowania idola, w której zamiast turnieju sportowego mówimy o śledzeniu w mediach społecznościowych. Jest n osób, dla każdej pary (A, B) zarówno A może śledzić B, jak i niezależnie B może (lub nie) śledzić A. Innymi słowy, między każdą parą wierzchołków może być krawędź w dowolnym kierunku, w obu naraz albo w żadnym. W takim układzie idol to taka osoba, którą wszyscy śledzą, a ona nie śledzi nikogo. Zmodyfikuj podany algorytm tak, aby za pomocą $3n$ (lub mniej) zapytań postaci „Czy A śledzi B?" znajdował idola w takiej sytuacji. Uwaga: odpowiedź na pytanie „Czy A śledzi B?" ani nie potwierdza, ani nie wyklucza tego, że B śledzi A.

PODSUMOWANIE LEKCJI

- Graf skierowany, w którym między każdą parą wierzchołów jest jedna krawędź w jedną albo drugą stronę, nazywamy *turniejem*. s. 49
- Wierzchołek, do którego krawędzie tylko wchodzą (a żadna nie wychodzi), nazywamy *idolem* lub *ujściem*. s. 50
- Aby znaleźć wierzchołek będący idolem, nie trzeba znać wszystkich krawędzi grafu. Istnieje algorytm znajdowania idola, który wykonuje tylko $2n$ zapytań postaci „w którą stronę jest poprowadzona krawędź?". s. 50

10. Najkrótsze ścieżki w grafie, czyli jak komputery znajdują drogę

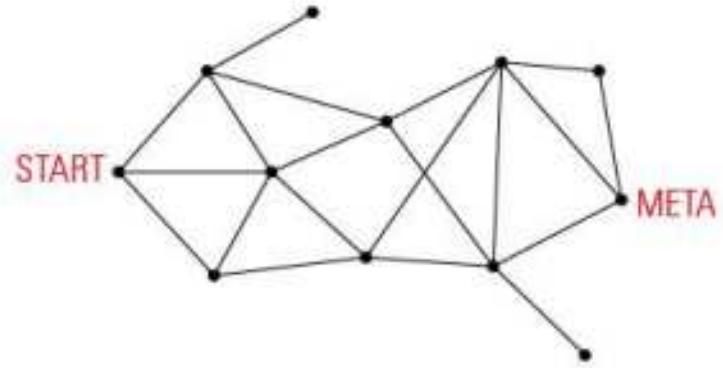
NA TEJ LEKCJI:

- poznasz algorytm BFS (przeszukiwania grafu wszerz), służący do znajdowania najkrótszych ścieżek w grafie;
- dowiesz się, jak znaleźć najszybsze połączenie lotnicze albo zweryfikować teorię o sześciu stopniach oddalenia.

Najkrótsze ścieżki

Jak opowiadaliśmy w rozdziale *Grafy*, wiele praktycznych, życiowych sytuacji modeluje się jako grafy. W szczególności wiele takich, w których grafem jest sieć połączeń drogowych lub lotniczych.

Rozważmy zatem taki graf. Oznaczmy pewne dwa z jego wierzchołków *start* i *meta*, a na polu *start* ustawmy pionek. Pionek może teraz poruszać się po grafie, w jednym ruchu przechodząc wzdłuż krawędzi z jednego wierzchołka do innego.



Przykładowy graf z zaznaczonymi wierzchołkami: startowym (*start*) i docelowym (*meta*)

Naturalne pytania, które można postawić, brzmią:

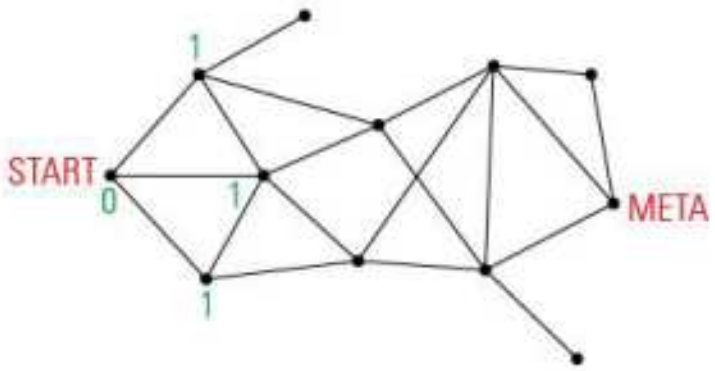
- Czy da się dojść od *startu* do *mety*?
- Ile co najmniej ruchów potrzeba, aby dojść do *mety*?
- Na które pola da się dojść ze *startu*, a na które nie?

Jeśli nasz graf to np. sieć połączeń kolejowych czy lotniczych, to tak naprawdę pytamy, czy da się dostać z jednego miasta do drugiego, a jeśli tak, to ile przejazdów/lotów trzeba w tym celu wykonać. Nawet w sieci społecznościowej można znaleźć pewną analogię – popularną tzw. legendą miejską jest teoria o „sześciu stopniach oddalenia", czyli stwierdzenie, że każde dwie osoby na świecie łączy łańcuch co najwyżej 6 znajomości. Wygląda to na niemożliwe do sprawdzenia, ale gdyby ktoś był w posiadaniu pełnych danych sieci społecznościowej i zastosował poniższy algorytm..., jak najbardziej mógłby przekonać się, że to prawda.

przeszukiwanie grafu wszerz (BFS)

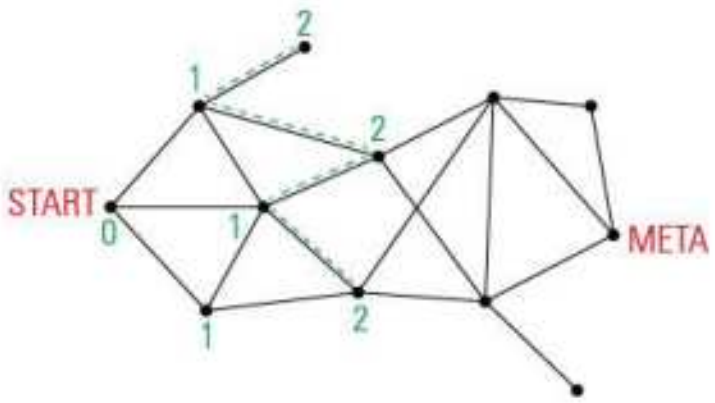
Algorytm, który opiszemy, potrafi znaleźć odpowiedź na wszystkie te pytania. Po polsku nazywa się **przeszukiwaniem grafu wszerz**, ale bardziej jest znany pod angielską nazwą **BFS** (ang. *breadth-first search*). Jego zadaniem

jest znaleźć dla każdego wierzchołka *x* jego odległość od wierzchołka *start*, czyli minimalną liczbę ruchów pionka, jakiej potrzeba, aby ze *start* dostać się do *x*. Oparty jest na zadziwiająco prostym pomyśle: po pierwsze, odległość do samego wierzchołka *start* wynosi 0 (nie trzeba w ogóle ruszać pionkiem), a do wszystkich sąsiadów startu odległość wynosi 1 – ponieważ możemy się dostać do nich w jednym ruchu pionkiem.



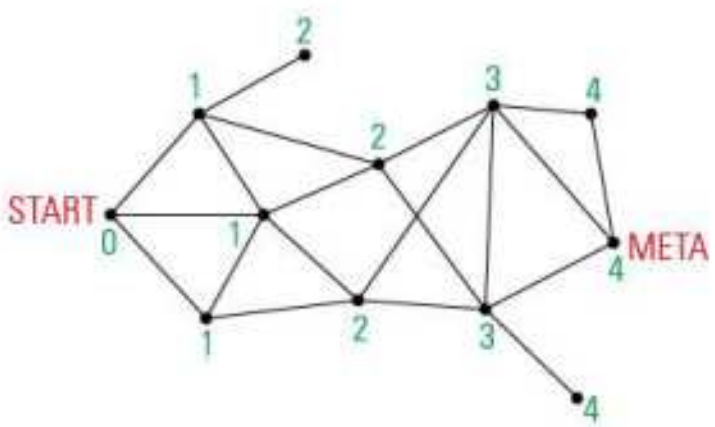
Pierwszy krok algorytmu (wartości tablicy *odległość* zaznaczone na zielono)

Teraz zauważmy, że jeśli jakiś wierzchołek *x* ma już przypisaną odległość 1, to wszyscy sąsiedzi *x*, którzy nie mieli jeszcze numeru, muszą mieć odległość 2. To dlatego, że można do nich dojść z *x* w jednym ruchu, a więc w dwóch ruchach ze startu; skoro nie mieli wcześniej numeru, nie mogą mieć odległości mniejszej niż 2. Procedurę możemy więc powtórzyć kolejno dla wszystkich wierzchołków, które mają już odległość 1...



Kolejny krok algorytmu

Następnie możemy zrobić to samo dla wierzchołków z odległością 2: każdy taki wierzchołek „daje" swoim sąsiadom odległość 3, o ile nie mają jeszcze numeru. Procedurę kontynuujemy, aż skończą się wierzchołki, których jeszcze nie rozważaliśmy (nie badaliśmy, nie analizowaliśmy).



Stan po wykonaniu całego algorytmu

Całą procedurę można opisać następującym – na razie jeszcze niecałkowicie kompletnym – algorytmem:

```
odległość[start] = 0
dla k = 0, 1, 2, ...
(*) rozważ wszystkie wierzchołki x w odległości k
              dla każdego rozważ wszystkich sąsiadów
(**)   i nadaj im odległość k+1
              (jeżeli wcześniej nie mieli nadanej)
```

Pozostaje pytanie, jak algorytm zaimplementować efektywnie i w sposób zrozumiały dla komputera. Możemy zauważyć, że każdy wierzchołek *x* pojawia się w algorytmie dwa razy:

- pierwszy raz, kiedy „otrzymuje" swoją wartość odległości *k* od jednego ze swoich sąsiadów w linii (**);
- drugi raz, kiedy sam jest rozważany i „nadaje" odległość $k+1$ wszystkim własnym sąsiadom (*).

Zanim jednak nastąpi druga wizyta i wierzchołek z odległością *k* będzie mógł przeglądać swoich sąsiadów, musimy najpierw skończyć przeglądać wierzchołki o odległościach $k-1$ i mniejszych. Innymi słowy, po otrzymaniu swojej odległości *k* wierzchołek musi „poczekać", aż zostaną „obsłużone" wierzchołki, które otrzymały odległość wcześniej. Z pomocą przychodzi poznana w rozdziale 7. (*Stos i kolejka...*) struktura *kolejki*. Pomysł jest następujący: utrzymujemy przez cały czas działania algorytmu kolejkę wierzchołków, które już otrzymały odległość (**), ale wciąż czekają na rozważenie w (*). Na początku w kolejce jest tylko wierzchołek startowy, a docelowo pojawią się w niej wszystkie wierzchołki rozważane przez algorytm. Sama procedura jest prosta: ściągamy z kolejki pierwszy wierzchołek, podobnie jak w pierwszej wersji rozważamy (analizujemy) wierzchołki sąsiednie, nadajemy im odległość o 1 większą i dorzucamy je na koniec kolejki, żeby w swoim czasie zostały rozpatrzone. Ostatecznie algorytm wygląda następująco:

algorytm dla każdego wierzchołka

```
Q - kolejka wierzchołków, na początku pusta

// włóż start na początek kolejki
odległość[start] = 0
wrzuć start na początek kolejki Q
dopóki Q nie jest pusta, wykonuj:
// zdejmujemy pierwszy wierzchołek x z kolejki
      x = pierwszy wierzchołek z kolejki Q
      usuń x z początku kolejki
// rozważamy sąsiadów x
      dla każdego y, który jest sąsiadem x wykonaj:
            jeśli odległość[y] nie jest ustawiona
                  odległość[y] = odległość[x]+1
                  wrzuć y na koniec kolejki
```

Po zakończeniu algorytmu wszystkie wierzchołki, do których udało się dojść ze startu, mają przypisane odległości. Jeśli potrzebujemy wiedzieć, czy da się dojść do wierzchołka *meta* lub ile ruchów do tego potrzeba, wystarczy sprawdzić wartość `odległość[meta]`. Możemy też np. sprawdzić, do ilu wierzchołków w ogóle da się dojść, analizując tablicę `odległość[]`.

Na zakończenie jeszcze jedno ciekawe pytanie: co się dzieje, jeśli krawędzie grafu mają przypisane „długości" lub „ceny", a pionek musi ponosić koszty przesuwania się po nich? Wciąż szukamy najkrótszej ścieżki, ale nowy algorytm musi uwzględniać również ceny: najkrótsza ścieżka to teraz ta, która ma najmniejszą łączną cenę. Takim grafem jest np. klasyczna sieć dróg, a szukanie w niej najkrótszej ścieżki to (w dużym uproszczeniu) problem, który za każdym razem musi rozwiązywać nawigacja samochodowa. Okazuje się, że istnieje algorytm pokrewny do BFS, zwany **algorytmem Dijkstry** (wym. dejikstry), który potrafi znaleźć drogę również w takim przypadku. Używa on też bardziej zaawansowanego wariantu kolejki – tzw. kolejki priorytetowej, przechowującej wierzchołki nie w kolejności rozważania, a według najkrótszej aktualnie znanej odległości.

ZADANIA DO WYKONANIA

1. Przeanalizuj działanie algorytmu BFS na przykładowym grafie z rysunku, krok po kroku, zwracając za każdym razem uwagę na to, co aktualnie znajduje się w kolejce.
2. Zauważ, że jeśli potrzebujemy tylko odległości między danymi wierzchołkami *start* i *meta*, to algorytm niepotrzebnie działa za długo, pracując nawet po tym, jak już tę odległość znaleźliśmy. Zaproponuj modyfikację algorytmu, która w takim wypadku przyśpieszy jego zakończenie.
3. Zaproponuj algorytm, który mając daną sieć społecznościową, sprawdzi teorię sześciu stopni oddalenia, czyli rozstrzygnie, czy rzeczywiście każde dwie osoby są od siebie oddalone o co najwyżej 6 znajomości.

PODSUMOWANIE LEKCJI

- Algorytm BFS, czyli przeszukiwanie grafu wszerz, służy do szukania najkrótszej ścieżki w grafie, przy ustalonym wierzchołku startowym. s. 52
- Algorytm dla każdego wierzchołka wyznacza jego odległość od startu i opiera się na powtarzaniu procedury, w której każdy wierzchołek nadaje sąsiadom odległość o 1 większą od swojej własnej. Do przeglądania wierzchołków w odpowiedniej kolejności, gwarantującej poprawność, algorytm używa struktury kolejki. s. 54

11. Programowanie strukturalne kontra obiektowe, czyli dane w centrum uwagi

NA TEJ LEKCJI:

- poznasz nowy sposób myślenia o pisaniu programów: programowanie obiektowe, w którym centralnym pojęciem nie są procedury, lecz typy danych (klasy) i zmienne (obiekty);
- poznasz podstawowe pojęcia programowania obiektowego: *klasa*, *obiekt*, *pole*, *metoda*, a także – w dużym skrócie – mechanizmy dziedziczenia i polimorfizmu.

Programowanie strukturalne

Wszystkie dotychczas rozważane przez nas algorytmy, a także (pseudo)kody pojawiające się w treściach rozdziału, miały jedną wspólną cechę: były, jak na standardy programowania, krótkie – zamykały się w kilkunastu, kilkudziesięciu linijkach kodu. Dla porównania: kody do współczesnej gry komputerowej mają nawet kilkaset *tysięcy* linii kodu, nie licząc nawet rozmaitych gotowych wcześniej bibliotek, z których korzystają. Kod systemu operacyjnego, takiego jak Windows czy Linux, liczy się w dziesiątkach milionów linii i oczywiście jest dziełem setek lub tysięcy programistów, z których jedni są oryginalnymi twórcami, a inni stopniowo go ulepszają, wyszukują i usuwają błędy czy rozwijają i dodają nowe funkcjonalności.

Jak nie stracić kontroli nad tak długim programem? Jest oczywiste, że nie można go pisać tak, jak pisaliśmy algorytmy – jako jedną długą główną procedurę z ewentualnymi kilkoma funkcjami pomocniczymi. Żaden programista nie będzie mógł w tak długim kodzie znaleźć błędu, nie wspominając już o przerobieniu tego kodu. Trzeba znaleźć sposób podziału programu na mniejsze części tak, aby programiści mogli je tworzyć, pracować nad nimi i testować je oddzielnie.

Dla ilustracji wyobraźmy sobie bardzo prostą grę, w której musimy unikać duchów poruszających się po prostokątnej planszy. Duchy poruszają się w przypadkowy sposób, o jedno pole na sekundę, pionowo lub poziomo.

Zrzut ekranu (fragment) z gry FreePacman

Oczywiście nie będziemy w tym przykładzie pisać programu ani nawet wybierać języka programowania, ale poprzestaniemy na umownym pseudojęzyku. Musimy zatem pamiętać pozycję każdego ducha (wiersz i kolumna), na której się znajduje. W jednej sekundzie gry powinniśmy więc, dla każdego ducha, wykonać następujące kroki:

- wymazać go z ekranu komputera z pozycji, na której się znajduje;
- ustalić dla niego nową pozycję;
- narysować go na ekranie w nowej pozycji.

Nie wchodząc w szczegóły techniczne naszej wymyślonej gry, niech funkcja `rysuj _ ducha(i,j)` służy do narysowania na ekranie ducha w *i*-tym wierszu i *j*-tej kolumnie planszy, a funkcja `usuń _ ducha(i,j)` – do wyczyszczenia takiego pola. Duchów niech będzie *N*. Musimy pamiętać pozycję każdego ducha. Użyjemy do tego tablic *W*[1 ... *N*] i *K*[1 ... *N*] – duch numer *s* niech znajduje się w wierszu *W*[*s*] i w kolumnie *K*[*s*]. Teraz jedna sekunda gry w naszym programie wyglądałaby w sposób podobny do poniższego:

```
funkcja rysuj_ducha(i,j)
        (instrukcje odpowiadające za narysowanie
        ducha na ekranie)
funkcja usuń_ducha(i,j)
dla s = 1, 2, ..., N
        usuń_ducha(W[s],K[s])
        wybierz nową pozycje - nowe wartości W[s], K[s]
        rysuj_ducha(W[s],K[s])
```

Programowanie obiektowe: klasy, obiekty, pola, metody

Powyższy program – jak i wszystkie dotychczasowe – napisaliśmy w tzw. sposób *strukturalny*. W **programowaniu strukturalnym** dane (czyli zmienne, takie jak pozycje duchów) są oddzielone od procedur i funkcji. Program to ciąg procedur operujących na danych.

W **programowaniu obiektowym** zmieniamy sposób myślenia: procedury i funkcje mają być przyporządkowane do danych i od nich zależne. W rozważanym przykładzie chcielibyśmy – mówiąc w uproszczony sposób – żeby każdy duch *„sam" pamiętał swoją pozycję* i „sam" *rysował się na ekranie*. Możemy o tym myśleć tak, że obok liczb naturalnych czy napisów tworzymy nowy typ danych – zmienne, które nazywamy tu duchami . Zmienna typu „duch" byłaby nieco bardziej skomplikowana niż dotychczas poznane: składałaby się (co najmniej) z dwóch liczb całkowitych *wiersz* i *kolumna*. W języku C++ definicja takiej nowego typu zmiennej wygląda tak:

```
class duch
{
        int wiersz, kolumna;
}
```

Co oznacza: zmienna typu „duch" to po prostu (na razie) para liczb całkowitych. Tajemnicze słowo *class* tłumaczy się na polski jako „klasa" – w naszym nowym sposobie myślenia takie zdefiniowane przez nas typy danych nazywa się właśnie **klasami**. Wartości przechowywane wewnątrz klasy – w tym wypadku *wiersz* i *kolumna* – nazywa się **polami** tej klasy.

klasa, pole

Uwaga!

Jeśli nie znasz C++, nie przejmuj się – będziemy używać przykładów tylko najprostszych konstrukcji, które nie powinny być trudne do zrozumienia. W większości języków składnia dla klas i obiektów jest podobna, z tym że np. w Pythonie nie trzeba w klasie deklarować pól (w Pythonie zmienne w ogóle nie wymagają deklaracji). Do wyjaśnienia użyjemy C++, dlatego że sporo innych języków opiera na nim swoją składnię – zwłaszcza języków projektowanych z myślą o programowaniu obiektowym (więcej o tym na samym końcu rozdziału).

Teraz możemy używać zmiennych typu „duch" w naszym programie, analogicznie jak używalibyśmy innych zmiennych. Jeśli *X* jest taką zmienną, to przez `X.wiersz` oznaczamy wartość pola `wiersz` (nie tylko w C++, lecz także w Pythonie, Javie i większości języków programowania), a przez `X.kolumna` – wartość pola `kolumna`. Zmienna typu będącego klasą nazywa się *obiektem*. Podsumowując: wszystkie duchy, które pojawią się w naszej grze, będą obiektami (zmiennymi) klasy `duch`.

Ale to nie koniec: funkcje takie jak `rysuj_ducha()` czy `usuń_ducha()` również można uczynić częścią klasy `duch`. Można to nieformalnie określić tak, że obiekty tej klasy „same odpowiadają za narysowanie siebie na ekranie".

W C++ mogłoby to wyglądać tak:

```
class duch
{
        int wiersz, kolumna;        // pola klasy
        void rysuj() // funkcja rysująca ducha
        {
                [...]
        }
        void usun()   // funkcja usuwająca ducha
        {
                [...]
        }
        void ruch()
        {
                znajdź pole, na które przesunie się duch
                wiersz = nowa wartość wiersza
                kolumna = nowa wartość kolumny
        }
}
```

Funkcje będące częścią klasy nazywają się *metodami* klasy. Jak wyglądałoby przesuwanie duchów na planszy po wprowadzeniu pól i metod? Prawdopodobnie użylibyśmy tablicy *D*[1...*n*], której elementy teraz są obiektami klasy `duch`, a główna pętla wyglądałaby tak samo.

```
dla j = 1, 2, ..., N
        D[j].usuń()
        D[j].ruch()
        D[j].rysuj()
```

W zasadzie są to te same instrukcje, co przy programowaniu strukturalnym, ale odnieśliśmy już pierwszą korzyść: możemy w dużo łatwiejszy sposób oddzielić od głównego programu samą klasę `duch` i np. odstąpić ją do zaprogramowania komuś innemu. Dość typowe przy większych projektach programistycznych jest podzielenie ich na wiele plików tak, aby każdy odpowiadał jednej klasie. To jednak nie koniec zalet podejścia obiektowego – niezwykle ważną jego część stanowi mechanizm dziedziczenia.

Dziedziczenie i polimorfizm

Odwołajmy się jeszcze raz do przykładu z grą i duchami. Co jeśli na planszy jest kilka rodzajów duchów? Wyobraźmy sobie np., że oprócz zwykłych duchów są jeszcze:

- „zielone" duchy, które poruszają się tak samo jak zwykłe, ale trzeba rysować je inaczej (choć usuwać tak samo);
- „specjalne" duchy, które wyglądają tak jak zwykłe (rysuje się je i usuwa tak samo), ale poruszają się w inny sposób.

Innymi słowy, dla niektórych duchów potrzebujemy osobnej procedury `rysuj()`, a dla innych – osobnej procedury `ruch()`. W dawnym myśleniu programowania strukturalnego musielibyśmy mieć osobne tablice i osobne pętle (pierwsza wywoływałaby się dla „zwykłych" duchów, druga – dla „zielonych", trzecia – dla „specjalnych"). Byłoby wygodniej, gdyby udało się utrzymać zarówno tablicę duchów *D*[1...*n*], jak i główną pętlę w formie takiej jak wcześniej:

```
dla j = 1, 2, ..., N
        D[j].usuń()
        D[j].ruch()
        D[j].rysuj()
```

Z drugiej strony, niektóre z duchów powinny być duchami innego rodzaju, więc nie powinny być w tej samej tablicy. Rozwiązaniem jest mechanizm znany jako dziedziczenie: tworzymy klasy `zielony_duch` oraz `specjalny_duch`, które są tzw. **klasami potomnymi** – zachowują wszystkie własności (pola i metody) klasy `duch`, a dodatkowo możemy, jeśli jest taka potrzeba, nadpisać dowolną z istniejących metod, czyli stworzyć jej nową wersję. W naszym przykładzie zdefiniujemy klasy potomne `zielony_duch` i `specjalny_duch`. Dla klasy `zielony_duch` napiszemy oddzielną metodę `rysuj()`, a dla

dziedziczenie, klasa potomna

`specjalny_duch` – oddzielną metodę `ruch()`. Ten mechanizm jest nazywany **dziedziczeniem** klas.

Co ciekawe, główna pętla programu może pozostać niezmieniona – jednym z założeń programowania obiektowego jest to, że program dla każdego ducha wywoła najlepiej pasującą metodę – dla klas potomnych nowe wersje `rysuj()` bądź `ruch()`, a tam, gdzie nie ma takiego wariantu – starą wersję tych metod. Ten z kolei mechanizm – wywoływania odpowiedniej wersji metod – nazywa się *polimorfizmem*.

Programowanie obiektowe to bardzo potężne narzędzie – dość powiedzieć, że w zasadzie wszystkie projekty programistyczne są tworzone w ten sposób. Toteż praktycznie wszystkie używane komercyjne języki programowania wspierają programowanie obiektowe. Niektóre z popularnych języków – takich jak Java czy C# – zostały w całości zaprojektowanych „obiektowo", czyli z założeniem, że każdy kod jest w ten sposób pisany.

ZADANIA DO WYKONANIA

1. W małej grupie wybierzcie jakąś grę lub prostą aplikację (szachy, Tetris, komunikator internetowy, kalkulator) i zastanówcie się wspólnie, z jakich obiektów składałby się taki projekt. Jak byście go podzielili między siebie?
2. Przeanalizuj struktury danych omówione w rozdziałach „Stos i kolejka" oraz „Tablice dynamiczne i listy wiązane" i zastanów się, które konstrukcje w wybranym języku programowania są klasami, polami i metodami.

PODSUMOWANIE LEKCJI

s. 58

- Zarówno w programowaniu strukturalnym, jak i obiektowym mamy do czynienia z danymi oraz z funkcjami/procedurami. Różnica między tymi dwoma podejściami jest na poziomie organizacji programu.

s. 60

- W programowaniu strukturalnym procedury są odrębne od danych i na nich operują. W programowaniu obiektowym procedury (metody) są traktowane jako część danych (klas, obiektów).

12. Jak rozwiązywać zadania programistyczne, czyli podstawowe porady na proste kody

NA TEJ LEKCJI:

- dowiesz się, jak wyglądają typowe zadania algorytmiczno-programistyczne (które pojawiają się na wszelkiego rodzaju konkursach i olimpiadach);
- poznasz kilka podstawowych porad, jak podchodzić do takich zadań i ogólnie – do programowania.

Jak zwykle wyglądają zadania i ich rozwiązania

Zadania programistyczne, które omówiono w tym rozdziale, wyglądają na ogół w sposób podobny do następującego:

W pliku liczby.txt dane jest 100 liczb z przedziału [2, 1 000 000]. Oblicz, ile jest wśród nich liczb pierwszych.

albo:

W pliku napisy.txt dane jest 200 napisów składających się z małych liter (co najmniej jednej, co najwyżej 100). Znajdź najdłuższy napis, który jest palindromem (czytany od przodu i od tyłu jest taki sam).

Takie zadania są elementarzem początkującego programisty, można je spotkać nawet na rozmowach rekrutacyjnych na staż, a bardzo podobne można spotkać też na Olimpiadzie Informatycznej, Olimpiadzie Informatycznej Juniorów czy na innych konkursach algorytmicznych. W konkursach, co prawda, częściej zamiast z plików czyta się ze standardowego wejścia (czyli „z klawiatury"), ale same zadania oparte są na podobnym schemacie: wczytaj dane, przeanalizuj, rozwiąż dla nich jakiś (prostszy lub trudniejszy) problem algorytmiczny, zapisz odpowiedź.

Na tym etapie znasz już wszystkie potrzebne ci algorytmy. Skoncentrujmy się zatem na poradach, jak postępować z zadaniami i jak pisać kod.

Porady

Uwaga!

1. Przygotuj się! Rozwiązuj zadania, pisz programy.

jak podchodzić do rozwiązywania zadań

Z umiejętnością programowania jest trochę jak z jazdą samochodem: po przepisowych kilkudziesięciu godzinach jazdy na kursie dalej masz poczucie, że to za mało. Dopiero kiedy przejedziesz odpowiedną liczbę kilometrów, nabierasz pewności siebie, kręcisz kierownicą odruchowo i bez niepotrzebnego

kombinowania, a także przestajesz popełniać błędy. Tak samo jest z programowaniem. Aby umieć pisać programy, pisz programy. Aby umieć rozwiązywać zadania, rozwiązuj zadania! Zadania można znaleźć na stronach z kursami algorytmicznymi czy na serwerach typu *online judge* (dobrym przykładem jednego i drugiego jest serwis Szkopuł). Jednak jeśli dopiero zaczynasz przygodę z programowaniem, autorzy polecają zadania z Olimpiady Informatycznej Juniorów. Na jej stronie można znaleźć zadania z poprzednich edycji, które stanowią dobry materiał do ćwiczeń na każdym poziomie początkującego programisty algorytmika.

Uwaga!

2. Przeczytaj dokładnie treść zadania.

Niełatwo jest zachować spokój. W warunkach egzaminu, testu kwalifikacyjnego na staż czy konkursu stres pojawia się u prawie wszystkich, a czasem zachęca do pośpiechu i nierozważnych kroków, z których jednym z najczęstszych jest pobieżne i niedokładne przeczytanie treści zadań. Uważaj na to! Jeżeli w treści napisane jest „sprawdź, czy słowa są anagramami", a ty przeczytasz „palindromami" i poprawnie rozwiążesz inne zadanie, nic ci to nie da.

Na samym początku, kiedy jest najwięcej czasu i najmniejsza presja, przeczytaj od początku do końca treści wszystkich zadań. Upewnij się, że je rozumiesz – nie musisz jeszcze wiedzieć, jak je rozwiązać, ale ułóż sobie w głowie, co w zadaniu jest wejściem (co wczytuje program), a co – wyjściem (co masz wypisać). Dopiero kiedy masz poczucie, że rozumiesz wszystkie zadania, zacznij myśleć nad rozwiązaniami. Może się zdarzyć, że nie wiesz, jak wykonać wszystkie zadania. Zrozumienie wszystkich pozwoli ci podjąć decyzję, którymi się zająć, a które odłożyć na później.

Uwaga!

3. Dobrze przemyśl algorytm i pisz tak, żeby się potem zrozumieć.

Czyli „myśl dwa razy, żeby nie kodzić trzy razy". Autorzy książki nie wiedzą, skąd to powiedzenie pochodzi, ale się z nim identyfikują. Często zdarzało nam się rzucić do pisania programu, napisać kilkadziesiąt (czy wręcz kilkaset) linijek kodu, utknąć (zagrzebać się), poprawić kilka razy, po czym dojść do wniosku, że cały pomysł na program nie ma sensu i trzeba zacząć od nowa. Nie powtarzaj naszych błędów – przed napisaniem pierwszej linijki **zastanów się** (**a może i naszkicuj na kartce**), **jak będzie wyglądała główna struktura programu**, jakich zmiennych i pętli będziesz potrzebować. Bardzo często trzeba zwrócić uwagę na szczegóły typu „czy ta pętla ma być od 1 do n, czy od 0 do $n-1$?" – napisanie jej wcześniej na kartce bardzo pomaga.

Naczelną zasadą powinno być **pisanie swojego programu tak, by zawsze mieć nad nim całkowitą kontrolę** – mieć pełną świadomość, do czego służyła dana zmienna/instrukcja. Co ciekawe, pisanie prosto i zrozumiale nie zawsze oznacza pisanie jak najkrócej. Zalecamy np. używanie opisowych nazw zmiennych

(*dzielnik, dlugosc_slowa, czy_palindrom*) – nazwy jednoliterowe (*a, b, c...*) stosuj tylko dla „krótko żyjących" zmiennych, np. zmiennej kontrolnej pętli.

Uwaga!

4. Program musi być elastyczny, czyli bez kopiuj–wklej.

Być może znasz tę regułę, ale... mamy wrażenie, że nasi uczniowie nie zawsze biorą ją na poważnie: unikaj w programie stałych wartości zapisanych „na sztywno". Bardzo typowy przykład: nawet jeśli wiesz, że w zadaniu masz plik wejściowy zawierający 100 liczb, przez co w twojej tablicy powinno przez cały czas być 100 liczb, nie projektuj swoich pętli tak:

(C++)

```
for(int i=1; i<=100; i++)
        (...)
```

(Python)

```
for i in range(1,100):
        (...)
```

Zamiast tego pisz tak:

(C++)

```
const int ile = 100;
(...)
for(int i=1; i<=ile; i++)
        (...)
```

(Python)

```
ile = 100
(...)
for i in range(1,ile):
        (...)
```

Głównym powodem tej rady jest fakt, że dane lubią się zmieniać. Może się okazać, że za chwilę będziesz pracować na pliku z nie 100, a np. 20 liczbami. Albo źle przeczytasz treść za pierwszym razem. Jest też drugi powód – jeśli musisz napisać tę samą liczbę (np. 1000) kilka, kilkanaście razy, to statystycznie jeden raz napiszesz o jedno zero za mało lub za dużo. Każdy robi takie błędy – łącznie z zawodowymi programistami, o czym więcej za chwilę. Problem leży w tym, że błąd w stałej jest dość ciężki do odszukania i poprawienia. Zatem: **pisz tak, żeby nie prosić się o kłopoty**. Szczególnie ważna jest druga część omawianej tu (czwartej) zasady: nigdy nie używaj w programie techniki kopiuj–wklej! Jeśli masz kilka bardzo podobnych fragmentów, wydziel je w osobną procedurę/funkcję albo użyj pętli. Kilkukrotne skopiowanie kodu powoduje, że każdą przyszłą modyfikację czy każde poprawienie pomyłki trzeba będzie powtarzać wielokrotnie, a dodatkowo przy takim wielokrotnym powtarzaniu łatwo wprowadzić nowe błędy do programu.

testowanie programu

Uwaga!

5. Dobrze przetestuj program, czyli mały koszt, duży zysk.

Jak napisaliśmy wyżej: błędy w programowaniu zdarzają się wszystkim, również zawodowym programistom. Każda firma, tworząc projekt informatyczny, od razu zakłada, że błędy się pojawią. Najczęściej zatrudnia więc specjalny dział do ich sprawdzania, wyszukiwania, lokalizowania w kodzie programu i usuwania.

Zawsze więc warto zakładać, że kiedy kończymy program, jakiś błąd może się w nim znajdować. Jak się zatem o tym przekonać?

Rozważmy przykład podany na początku: program rozwiązujący zadanie o liczbach pierwszych:

W pliku liczby.txt dane jest 100 liczb z przedziału [2, 1 000 000]. Oblicz, ile jest wśród nich liczb pierwszych.

Uruchamiamy nasz świeżo napisany program, który podaje odpowiedź – niech to będzie 37. Aby teraz przetestować nasz program, np. zróbmy, co następuje:

- stwórzmy plik tekstowy o nazwie *test.txt*, wyglądający następująco:

```
2
3
4
5
6
7
8
9
10
11
```

- zmieńmy (na chwilę!) nasz program tak, żeby zamiast z pliku *liczby.txt* czytał z pliku *test.txt* i żeby zamiast 100 liczb pracował na 10 (korzystasz z porad z punktu 4, żeby wystarczyło zmienić w jednym miejscu?);
- sprawdźmy, jaką odpowiedź nasz program daje dla tego pliku – oczekujemy odpowiedzi 5, jako że w pliku jest 5 liczb pierwszych (2,3,5,7,11); jeśli dostaniemy taką odpowiedź, program zaliczył test, więc czas na stworzenie kolejnych (np. z większymi liczbami); jeśli jednak dostaniemy inną odpowiedź, musimy szukać błędu w naszym programie;
- najprostszą metodą szukania błędu jest dopisanie do programu instrukcji „raportujących" jego aktualny stan – w naszym przykładzie mogłaby to być instrukcja, która wypisuje na ekranie każdą analizowaną liczbę i wynik analizy (2: TAK, 3: TAK, 4: NIE...). Możemy zobaczyć, dla której liczby nasz program się pomylił, i zastanowić się krok po kroku, dlaczego to zrobił.

ZADANIA DO WYKONANIA

1. Rozważmy szachownicę, na której każde pole jest oznaczone przez swój numer wiersza i kolumny (np. pole [2,3] jest w drugim wierszu i trzeciej kolumnie). Napisz w wybranym języku programowania funkcję, która przyjmuje numer wiersza i kolumny pewnego pola i wypisuje dla niego pola sąsiednie – te, które stykają się z nim bokiem lub rogiem (np. dla pola [2,3] są to pola [1,2], [1,3], [1,4], [2,2], [2,4] itd.) Pól sąsiednich jest 8, chyba że pole leży na brzegu lub w rogu szachownicy – uwzględnij ten fakt przy wypisywaniu. **Napisz funkcję tak, aby nie powtórzyć 8 razy tego samego kodu.**
2. Przetestuj napisany w poprzednim punkcie program – czy działa dobrze dla pól w środku szachownicy? A dla pól na brzegu i w rogu?

PODSUMOWANIE LEKCJI

- Przygotuj się! Rozwiązuj zadania, pisz programy, nabieraj wprawy w kodowaniu i poprawianiu błędów. Czytaj dokładnie treści zadań. s. 61
- Przetestuj swój program: zrób własne testy, dla których odpowiedź będzie znana i sprawdź wyniki. Najprostszą metodą szukania błędu jest wypisywanie „raportu działania" przez program. s. 64

PODSUMOWANIE PRZED SPRAWDZIANEM

1. Więcej o liczbach, czyli sito Eratostenesa i schemat Hornera

s. 11
- Algorytm sita Eratostenesa służy do obliczania naraz wszystkich liczb pierwszych mniejszych od *N*. Opiera się na wykreślaniu spośród wszystkich liczb kolejnych wielokrotności, a działa w czasie bliskim liniowemu.

s. 12
- Schemat Hornera to technika, która pozwala w prosty sposób, w czasie liniowym, obliczyć wartość wielomianu.

2. Rekursja raz jeszcze, czyli dziel i zwyciężaj

s. 15
- Rekursja w algorytmach działa, jeśli są spełnione trzy warunki: wywołanie rekurencyjne następuje zawsze na mniejszych danych, jest określony początek rekursji oraz wynik jest prawidłowo odtwarzany z wyników wywołań rekurencyjnych.

s. 15
- Szybkie potęgowanie – działające w czasie $O(\log N)$, gdzie N jest wykładnikiem potęgi do obliczenia – można łatwo zaimplementować za pomocą rekursji.

s. 16
- Metoda *dziel i zwyciężaj* działa przez podzielenie danych na dwie (czasem więcej) części, wywołanie się na każdej osobno, a potem połączenie uzyskanych wyników w jeden.

3. Sortowanie przez scalanie, czyli pierwszy efektywny algorytm sortowania

s. 18
- Dwie posortowane tablice możemy scalić w czasie liniowym procedurą, która za każdym razem wybiera mniejszy z dwóch początkowych elementów, a następnie wykreśla go z tej tablicy.

s. 21
- Sortowanie przez scalanie dzieli tablicę na dwie mniejsze, sortuje każdą z nich rekurencyjnie, po czym łączy w jedną za pomocą algorytmu scalania.

s. 21
- Algorytm sortowania przez scalanie działa w czasie $O(n \log n)$.

4. Sortowanie szybkie, czyli jak się sortuje w praktyce

s. 23
- Algorytm sortowania szybkiego (*QuickSort*) wybiera klucz (jeden z elementów tablicy), przestawia elementy tablicy i dzieli ją tak, aby w lewej części były elementy mniejsze od klucza, a w prawej – większe. Potem wywołuje się rekurencyjnie na obu częściach.

s. 27
- Algorytm *QuickSort* ma złożoność pesymistyczną $O(n^2)$, ale średnią $O(n \log n)$ w wersji z losowaniem klucza.

s. 27
- Algorytm *QuickSort* jest w praktyce najszybszym i najczęściej stosowanym z dotychczas poznanych algorytmów.

5. Specjalne elementy w tablicy, czyli największy, najmniejszy i lider

s. 29
- Standardowo element największy lub najmniejszy w tablicy można znaleźć za pomocą $n-1$ porównań.

- Jeżeli jednak chcemy szukać jednocześnie największego i najmniejszego elementu, zamiast $2n-2$ można wykonać $3/2*n$ porównań. s. 29
- Element najczęściej występujący (dominantę) można znaleźć w tablicy za pomocą sortowania w czasie $O(n \log n)$. s. 31
- Jeżeli szukamy lidera – elementu, który występuje więcej niż $n/2$ razy – można użyć prostego, liniowego algorytmu opartego na wykreślaniu z tablicy par różnych elementów. s. 32

6. Tablice dynamiczne i listy wiązane, czyli kiedy tablica nie wystarcza

- Tablica dynamiczna ma zmienny rozmiar – pozwala dopisywać i usuwać elementy z końca tablicy. s. 35
- Lista wiązana przechowuje elementy w różnych miejscach pamięci, a każdy element trzyma adres następnego (i zwykle również poprzedniego). s. 37

7. Stos i kolejka, czyli struktury proste i bardzo użyteczne

- Struktura stosu pozwala na dokładanie i zdejmowanie elementów – zawsze jest zdejmowany najpóźniej włożony element. s. 39
- Stos używany jest m.in. do przechowywania wywołań funkcji, a także w algorytmie obliczania wyrażeń w odwrotnej notacji polskiej. s. 40
- Struktura kolejki działa analogicznie, ale zawsze jest zdejmowany najwcześniej włożony element. Kolejka jest używana m.in. do obsługi żądań serwera, a także w algorytmie szukania najkrótszej ścieżki. s. 42

8. Grafy, czyli co mają wspólnego sieć społecznościowa i paryskie metro

- Graf to zbiór wierzchołków oraz zbiór krawędzi łączących wierzchołki. Grafy mogą być skierowane (krawędzie są jednokierunkowe) lub nieskierowane (krawędzie są dwukierunkowe). s. 46
- Graf można przechowywać w pamięci komputera w postaci macierzy sąsiedztwa (kwadratowa tablica, w której 0/1 oznacza brak/istnienie krawędzi) lub list sąsiedztwa (dla każdego wierzchołka w liście lub tablicy dynamicznej są przechowywani wszyscy jego sąsiedzi). s. 47

9. Wyszukiwanie idola, czyli o tym, jak spóźniliśmy się na turniej tenisowy

- Graf skierowany, w którym między każdą parą wierzchołków jest jedna krawędź w jedną albo drugą stronę, nazywamy *turniejem*. s. 49
- Wierzchołek, do którego krawędzie tylko wchodzą (a żadna nie wychodzi), nazywamy *idolem* lub *ujściem*. s. 50
- Aby znaleźć wierzchołek będący idolem, nie trzeba znać wszystkich krawędzi grafu. Istnieje algorytm znajdowania idola, który wykonuje tylko $2n$ zapytań postaci „w którą stronę jest poprowadzona krawędź?". s. 50

10. Najkrótsze ścieżki w grafie, czyli jak komputery znajdują drogę

s. 52

- Algorytm BFS, czyli przeszukiwanie grafu wszerz, służy do szukania najkrótszej ścieżki w grafie, przy ustalonym wierzchołku startowym.

s. 54

- Algorytm dla każdego wierzchołka wyznacza jego odległość od startu i opiera się na powtarzaniu procedury, w której każdy wierzchołek nadaje sąsiadom odległość o 1 większą od swojej własnej. Do przeglądania wierzchołków w odpowiedniej kolejności, gwarantującej poprawność, algorytm używa struktury kolejki.

11. Programowanie strukturalne kontra obiektowe, czyli dane w centrum uwagi

s. 58

- Zarówno w programowaniu strukturalnym, jak i obiektowym mamy do czynienia z danymi oraz z funkcjami/procedurami. Różnica między tymi dwoma podejściami jest na poziomie organizacji programu.

s. 60

- W programowaniu strukturalnym procedury są odrębne od danych i na nich operują. W programowaniu obiektowym procedury (metody) są traktowane jako część danych (klas, obiektów).

12. Jak rozwiązywać zadania programistyczne, czyli podstawowe porady na proste kody

s. 61

- Przygotuj się! Rozwiązuj zadania, pisz programy, nabieraj wprawy w kodowaniu i poprawianiu błędów. Czytaj dokładnie treści zadań.

s. 64

- Przetestuj swój program: zrób własne testy, dla których odpowiedź będzie znana i sprawdź wyniki. Najprostszą metodą szukania błędu jest wypisywanie „raportu działania" przez program.

II. Arkusz kalkulacyjny

- Podstawowe typy danych, czyli co możemy wpisać w komórkę arkusza
- Trójkąt Sierpińskiego, czyli do czego przydaje się adresowanie względne i adresowanie bezwzględne oraz wypełnienie serią danych
- Podstawowe funkcje tekstowe. Korzystanie z gotowych funkcji, czyli jak łatwo modyfikować dane tekstowe
- Podstawowe funkcje dotyczące daty i czasu. Korzystanie z gotowych funkcji, czyli jak łatwo pracować z datą
- Jak szukać w Excelu, czyli gdzie to jest
- Co ukrywa numer PESEL, czyli do czego przydaje się adres mieszany
- Tabela przestawna, czyli wygodne grupowanie danych
- Wykresy, czyli jak atrakcyjnie przedstawiać dane
- Pobieranie danych z pliku, czyli z notatnika do Excela

13. Podstawowe typy danych, czyli co możemy wpisać w komórkę arkusza

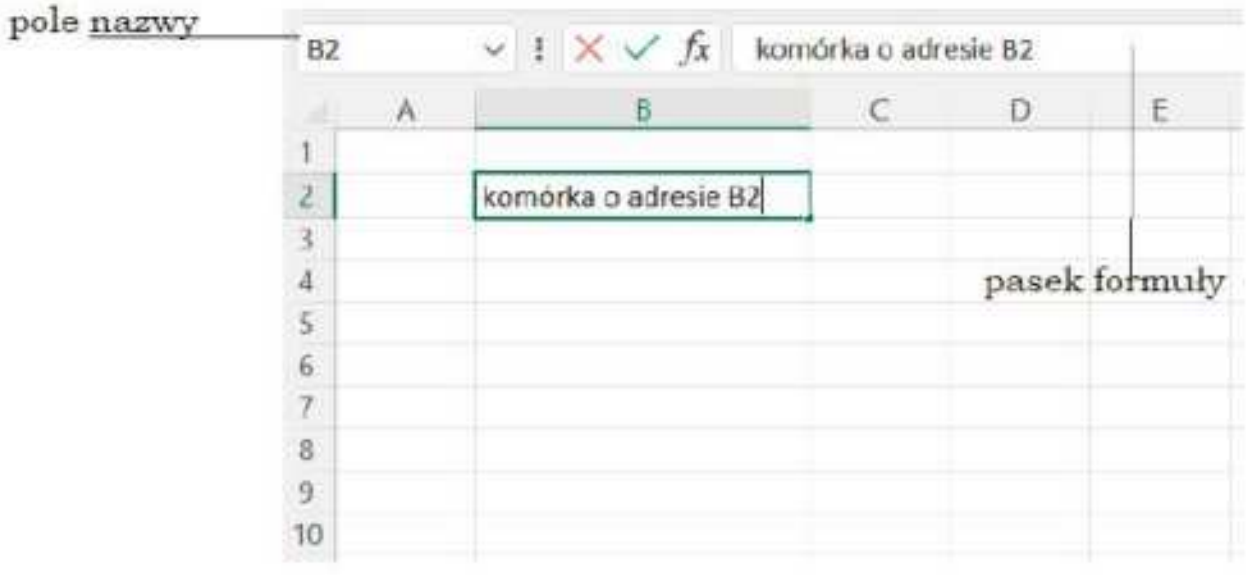
NA TEJ LEKCJI:

- przypomnisz sobie, jakie dane można wpisać w komórkę arkusza;
- przypomnisz sobie rodzaje adresowania komórek;
- przypomnisz sobie podstawowe funkcje używane w formułach.

Co możemy wpisać do komórki

wpisy do komórki ▸ Komórkę, do której można wpisywać dane lub je edytować, wyznacza kursor. Wiemy, że jesteśmy w trybie wpisywania danych do komórki bieżącej, gdy kursor przyjmuje postać obramowania bieżącej komórki.

Adres komórki składa się z nazwy kolumny i numeru wiersza, na przecięciu których się znajduje, np. B2.

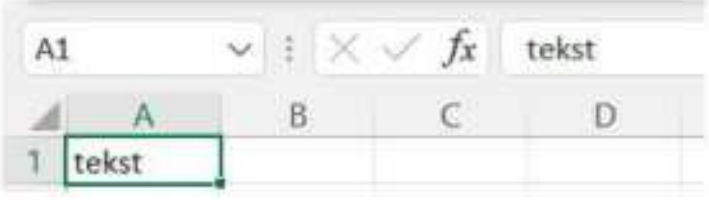


Adres komórki pojawia się w polu nazwy, które znajduje się nad roboczym arkuszem kalkulacyjnym, a wpisane dane pojawiają się w pasku formuły.

Do komórki arkusza możemy wpisać następujące typy danych:

- tekst,
- liczbę,
- formułę.

1. Tekst standardowo jest wyrównywany do lewej krawędzi komórki.



2. Liczba jest wyrównywana do prawej krawędzi komórki.

	A	B
1	13	
2	12.01.2022	
3	133,00 zł	
4		

3. Jeśli chcemy wpisać liczbę jako tekst, poprzedzamy ją znakiem '.

A1 '0123

	A	B	C	D	E
1	'0123				
2					
3					
4					

4. Formułę zawsze poprzedzamy znakiem =.

B1 =A1+B1

	A	B	C	D
1	23	45	=A1+B1	
2				

Sposoby adresowania komórki

adresowanie do komórek

Wyróżniamy trzy sposoby adresowania komórek:

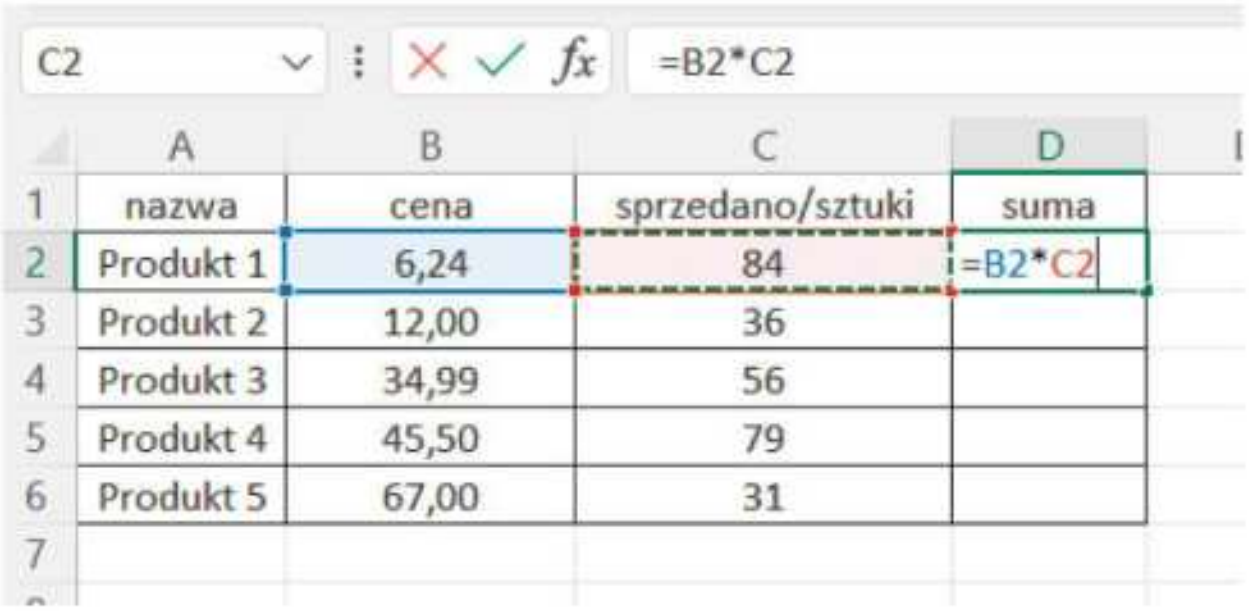
- względny,
- bezwzględny,
- mieszany.

Adresowanie względne

Adresowanie względne, np. A1, C3, powoduje zmianę adresu po przekopiowaniu formuły. Jeśli w komórkę D2 wpiszemy formułę `=B2*C2`, oznacza to, że w komórce D2 zostanie wstawiona wartość iloczynu komórki znajdująca się w tym samym wierszu, co komórka D2 (wiersz 2), o dwie kolumny na lewo (kolumna B), przez komórkę, która znajduje się w tym samym wierszu, co komórka D2 (wiersz 2), o jedną kolumnę na lewo (kolumna C).

Po przekopiowaniu formuły z D2 do komórki D3 otrzymamy formułę, która w D3 wstawi wartości iloczynu komórki znajdującej się w tym samym wierszu, co

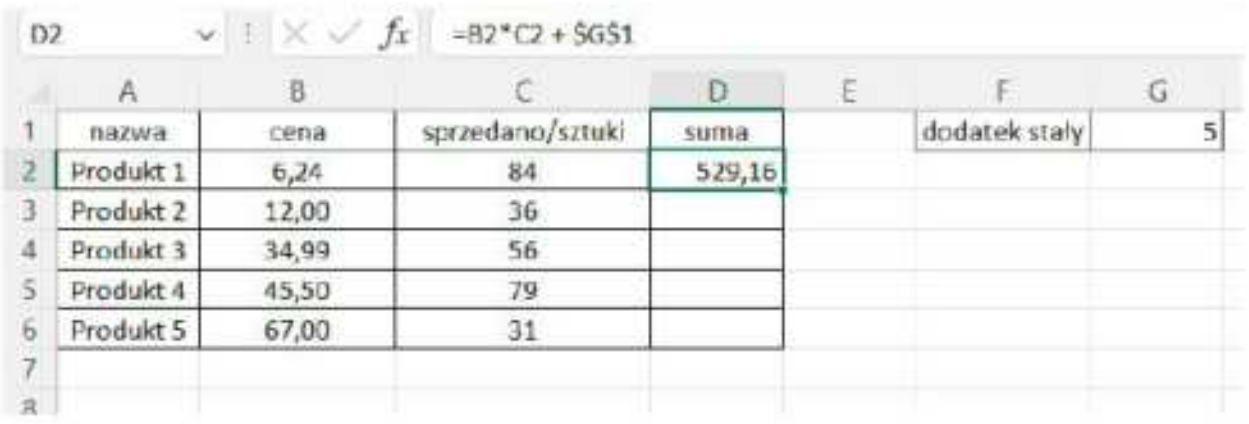
komórka D3 (wiersz 3), o dwie kolumny na lewo (kolumna B), przez komórkę, która znajduje się w tym samym wierszu, co komórka D3 (wiersz 3), o jedną kolumnę na lewo (kolumna C). Formuła w komórce D3 będzie wyglądać następująco: `=B3*C3`.



Przeciągnięcie formuły o wiersz niżej spowodowało utworzenie formuły `=B3*C3`.

Adresowanie bezwzględne

Adresowanie bezwzględne, np. $A$1, $C$3, powoduje „zakotwiczenie się" w danej komórce. Po przekopiowaniu formuły ten adres się nie zmieni.



Przeciągnięcie formuły o wiersz niżej spowodowało utworzenie formuły `=B3*C3+$G$1`.

Adresowanie mieszane

Adresowanie mieszane pozwala zablokować wiersz $A1 lub kolumnę A$1. Blokowanie odbywa się przez postawienie znaku $ przed wierszem lub kolumną. Po przekopiowaniu zmienia się niezablokowany wymiar.

Uwaga!

Szybkim sposobem na zamianę trybu adresowania jest naciśnięcie F4.

Formuły z użyciem funkcji

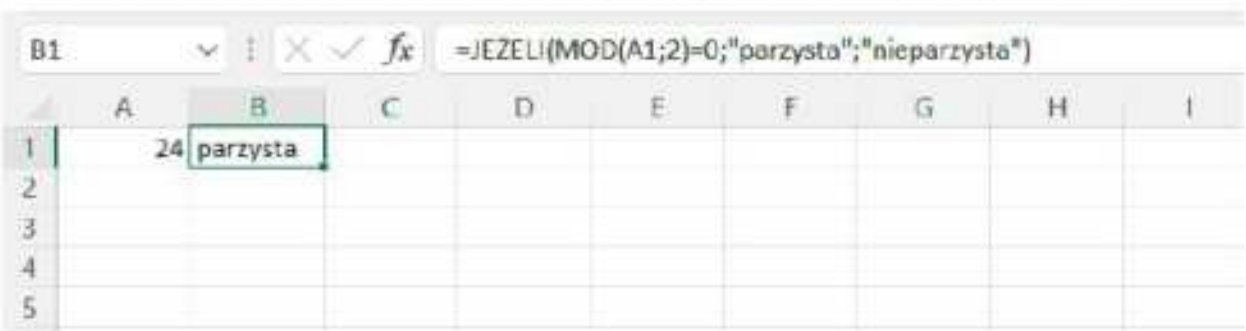
Do tworzenia formuł, oprócz operatorów matematycznych i adresów komórek, możemy używać dostępnych funkcji wbudowanych Formuły/Wstaw Funkcję.

Lista przydatnych funkcji

Nazwa	Opis
SUMA	Dodaje wartości.
MIN	Zwraca najmniejszą liczbę w zbiorze wartości.
MAX	Zwraca największą liczbę w zbiorze wartości.
ŚREDNIA	Zwraca średnią (średnią arytmetyczną) argumentów.
JEŻELI	JEŻELI (jakieś wyrażenie jest prawdziwe, to wykonaj określone działanie, w przeciwnym razie wykonaj inne działanie).
ORAZ	Sprawdza, czy wszystkie warunki w teście mają wartość PRAWDA.
LUB	Sprawdza, czy chociaż jeden z warunków w teście ma wartość PRAWDA
JEŻELI.BŁĄD	Zwraca wartość podaną w przypadku, gdy wynikiem formuły jest błąd, w przeciwnym razie zwraca wynik formuły.
LICZ.JEŻELI	Podaje liczbę komórek, które spełniają dany warunek.
SUMA.JEŻELI	Sumuje wartości z zakresu spełniającego określony warunek.
SUMA.WARUNKÓW	Dodaje wszystkie argumenty, które spełniają wiele kryteriów.
LICZ.WARUNKI	Podaje liczbę komórek, które spełniają podane warunki.
MOD	Wyznacza resztę z dzielenia.

Przykład:

Formuła `=JEŻELI(MOD(A1;2)=0;"parzysta";"nieparzysta")`



Daje napis „parzysta", jeżeli wartość w komórce A1 ma resztę 0 przy dzieleniu przez 2, i „nieparzysta" w przeciwnym przypadku.

ZADANIA DO WYKONANIA

1. Utwórz tabliczkę mnożenia 5 x 5.

	A	B	C	D	E	F
1		**1**	**2**	**3**	**4**	**5**
2	**1**	=				
3	**2**					
4	**3**					
5	**4**					
6	**5**					
7						

W komórkę B2 wpisz formułę, używając adresowania mieszanego, która po przekopiowaniu na prawo i w dół da prawidłowy wynik.

PODSUMOWANIE LEKCJI

s. 70 • W komórkę możemy wpisać tekst, liczbę i formułę.

s. 71 • Do komórki odwolujemy się przez adres względny, bezwzględny lub mieszany.

14. Trójkąt Sierpińskiego, czyli do czego przydaje się adresowanie względne i adresowanie bezwzględne oraz wypełnienie serią danych

NA TEJ LEKCJI:

- wykorzystasz w praktyce adresowanie komórek;
- dowiesz się, jak szybko wypełniać komórki podobnymi wartościami;
- przypomnisz sobie, jak korzystać z funkcji `JEŻELI`;
- poznasz funkcję `LOS.ZAKR()`;
- nauczysz się szybko kopiować i zaznaczać dane;
- przypomnisz sobie, jak wstawić wykres punktowy.

Wykonaj następujące polecenie. Użyj trzech podanych odwzorowań:

1. $\begin{cases} x' = d * x \\ y' = d * y \end{cases}$ 2. $\begin{cases} x' = d * x + 2 \\ y' = d * y \end{cases}$ 3. $\begin{cases} x' = d * x + 1 \\ y' = d * y + \sqrt{3} \end{cases}$

$d = 0{,}5$, do wygenerowania zbioru 5000 punktów i przedstaw go na wykresie punktowym.

Aby wykonać polecenie, wykorzystaj algorytm:

1. Wybierz dowolne wartości początkowe x i y.
2. Powtórz wiele razy (5000):
   - oblicz nową wartość x' i y' – aby to zrobić, wybierz jeden z trzech powyższych układów;
   - zaznacz na wykresie otrzymany punkt.

Aby wykonać zadanie w arkuszu kalkulacyjnym, wykorzystamy:

1. szybkie wypełnianie komórek podobnymi wartościami, żeby wypełnić kolumnę liczbami od 1 do 5000;
2. funkcję `LOS.ZAKR(1;3)` do wylosowania liczby 1, 2 lub 3;
3. funkcję `JEŻELI` do wybrania jednego z trzech podanych przekształceń w zależności od tego, która liczba: 1, 2 lub 3, została wylosowana;
4. wstawianie wykresu punktowego.

Zatem zróbmy to:

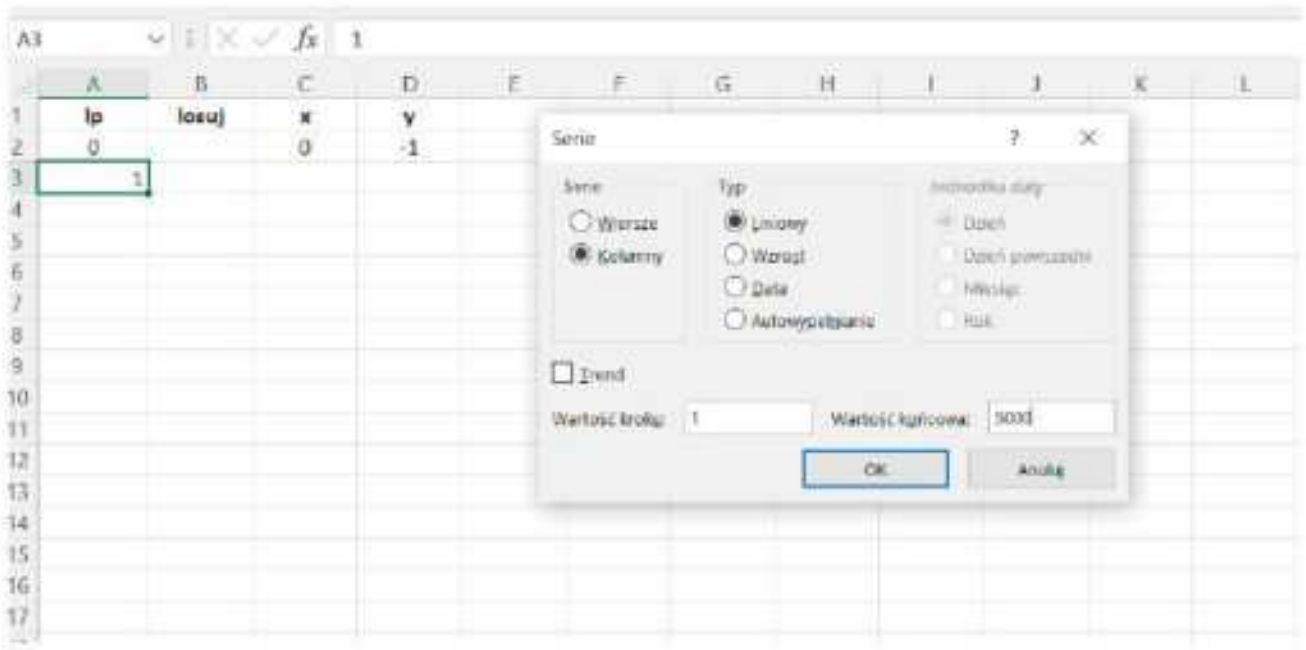
Jako punkt początkowy przyjmijmy $x = 0$ i $y = -1$.

	A	B	C	D	E
1	**lp**	**losuj**	**x**	**y**	
2	0		0	-1	
3					
4					
5					

szybkie wypełnianie komórek

Aby wygenerować liczby od 1 do 5000, wpisz w komórkę A3 wartość początkową, czyli 1. Mając wybraną komórkę A3, wybierz z **Narzędzi głównych** opcję **Wypełnij**.

Z opcji **Wypełnij** wybierz **Seria danych** i ustaw wartość kroku 1 oraz wartość końcową 5000. Chcesz wypełnić dane w kolumnie, więc pamiętaj o wybraniu opcji **Serie/Kolumny**.

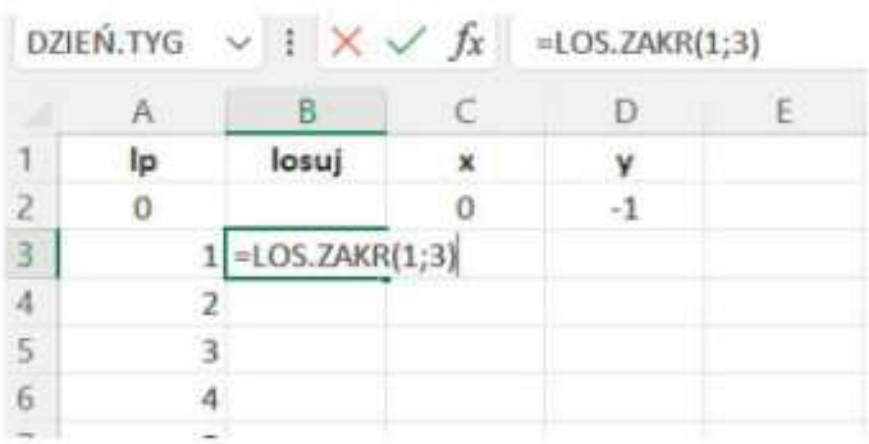


Po kliknięciu OK w kolumnie A zostaną wygenerowane liczby od 1 do 5000 z krokiem 1.

Podane powyżej trzy odwzorowania są tak samo prawdopodobne. Oznacza to, że jeśli rzucimy kostką trzyścienną i wypadnie 1, to użyjemy przekształcenia 1, jeśli wypadnie 2, to – przekształcenia 2, a jeśli 3, to – przekształcenia 3.

generowanie liczb losowych

Do symulacji rzutu kostką użyjemy funkcji `LOS.ZAKR(góra; dół)`, która losuje dowolną liczbę całkowitą z przedziału góra–dół. U nas ten zakres to 1–3.



Teraz obliczymy wartość x punktu 1 dzięki wykorzystaniu naszego odwzorowania. Jeżeli wylosowała się 1, użyjemy przekształcenia $x'=d*x$, jeżeli 2, to $-x'=d*x+2$, w przeciwnym przypadku (czyli wylosowała się 3) użyjemy przekształcenia $x'=d*x+1$.

Wykorzystamy do tego funkcje `JEŻELI` oraz adresowania względnego i bezwzględnego. W komórkę G1 wpiszmy wartość stałej d, czyli 0,5.

G1 | 0,5

	A	B	C	D	E	F	G
1	**lp**	**losuj**	**x**	**y**		**d**	0,5
2	0		0	-1			
3	1	2					
4	2						
5	3						
6	4						
7	5						

W komórkę D3 wpiszemy formułę:
`JEŻELI(B3=1;$G$1*C2;JEŻELI(B3=2;$G$1*C2+2;$G$1*C2+1))`.

Zwróć uwagę, że adres odwołujący się do stałej *d* piszemy $G$1, ponieważ nie chcemy, aby się zmieniał. Pozostałe adresy piszemy bez „$", ponieważ chcemy przekopiować formułę, tak aby do obliczania punktu drugiego używała wartości z punku pierwszego.

Uwaga!

Po wpisaniu do formuły adresu komórki G1 wciśnij klawisz **F4**, zrób to jeszcze raz. Klawisz **F4** pozwala nam szybko przechodzić między różnymi sposobami adresowania komórki G1 (G1, $G$1, G$1, $G1).

Analogicznie wyznaczamy wartość *y* punktu 1. Do obliczenia wartości $\sqrt{3}$ wykorzystamy funkcję `PIERWIASTEK)`.

D3 | =JEŻELI(B3=1;$G$1*D2;JEŻELI(B3=2;$G$1*D2;$G$1*D2+PIERWIASTEK(3)))

	A	B	C	D	E	F	G	H	I	J
1	**lp**	**losuj**	**x**	**y**		**d**	0,5			
2	0		0	-1						
3	1	1	0	-0,5						
4	2									
5	3									

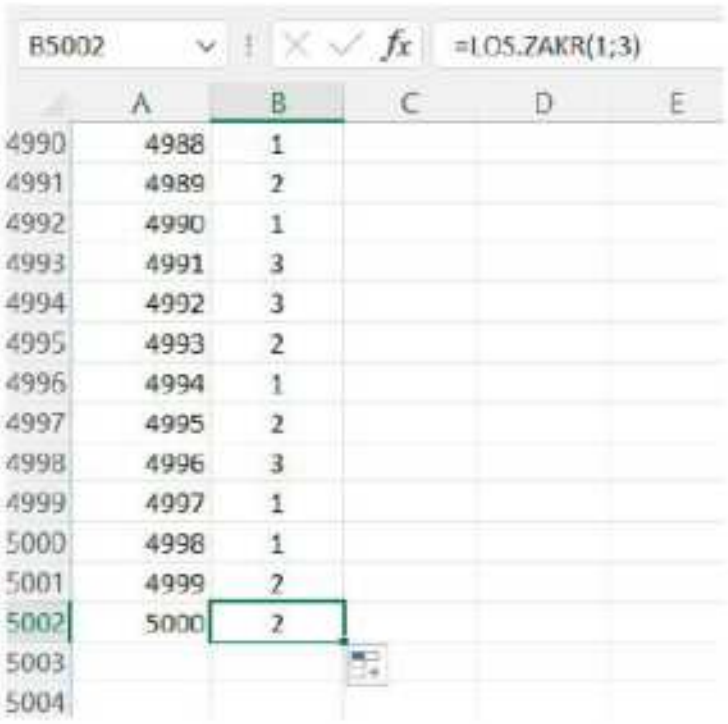
Teraz możemy przekopiować nasze formuły dla wszystkich pozostałych punktów.

Do zrobienia tego użyj szybkiego kopiowania. Wypróbuj szybkiego kopiowania na funkcji `LOS.ZAKR`. Ustaw kursor w prawym dolnym rogu komórki B3 i kliknij dwukrotnie.

B3

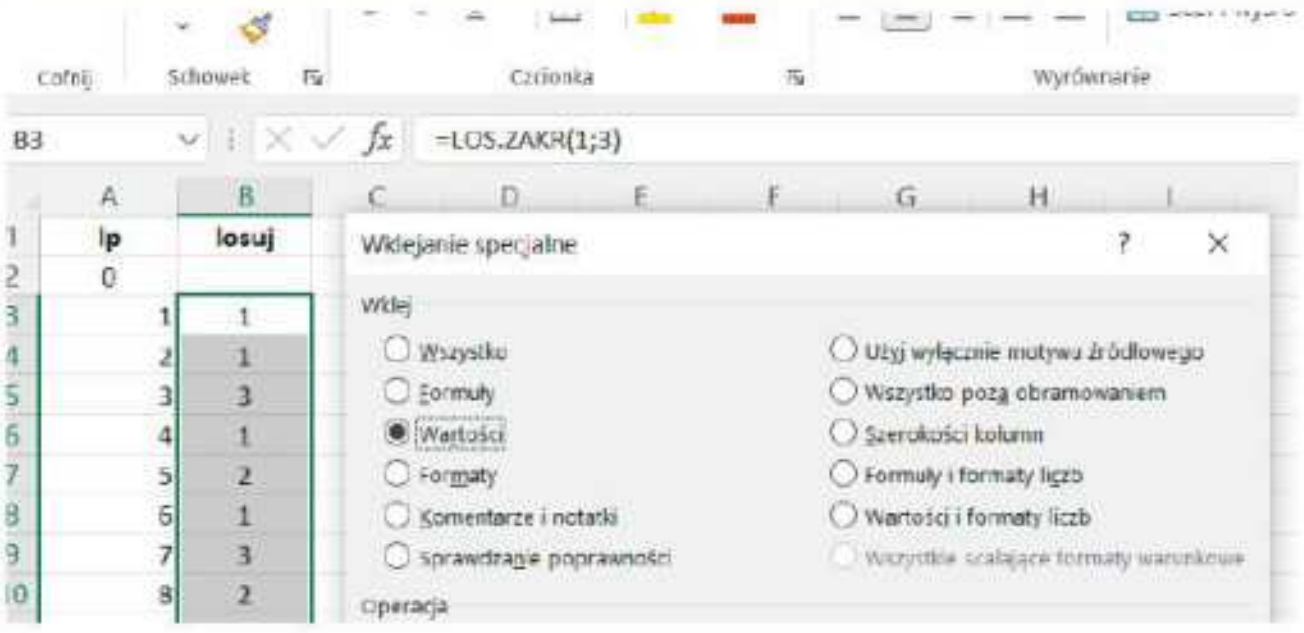
	A	B	C
1	**lp**	**losuj**	x
2	0		0
3	1	1	0
4	2		
5	3		
6	4		

Formuła zostanie przekopiowana na wszystkie komórki od B4 do B5002, czyli do tej „głębokości", do której są wypełnione dane w kolumnie A.



Uwaga!

W przypadku ponownego obliczania arkusza przez wprowadzenie formuły lub danych w innej komórce albo ręcznego ponownego obliczania dla każdej formuły korzystającej z funkcji `LOS.ZAKR` jest generowana nowa liczba losowa. Nie ma to wpływu na nasze zadanie, ale możesz przekopiować wylosowane liczby w to samo miejsce dzięki skopiowaniu wartości, a nie formuły. Aby to zrobić, zaznacz wszystkie komórki od B3 do B502 oraz wybierz opcję **Kopiuj** (**Ctrl+C**). Następnie ustaw się w komórce B3 i wybierz opcję **Wklejanie specjalnie/Wartości**.



Teraz w komórkach od B# do B5002 będą znajdowały się liczby 1, 2 lub 3.

Uwaga!

Do szybkiego zaznaczania komórek możesz użyć kombinacji klawiszy **Shift** + (←↑↓→) lub **Shift+Ctrl+**(←↑↓→). Ustaw się w komórce B3 i wciśnij klawisze **Shift+Ctrl+↓**. Obszar zaznaczenia powinien objąć komórki od B3 do B5002.

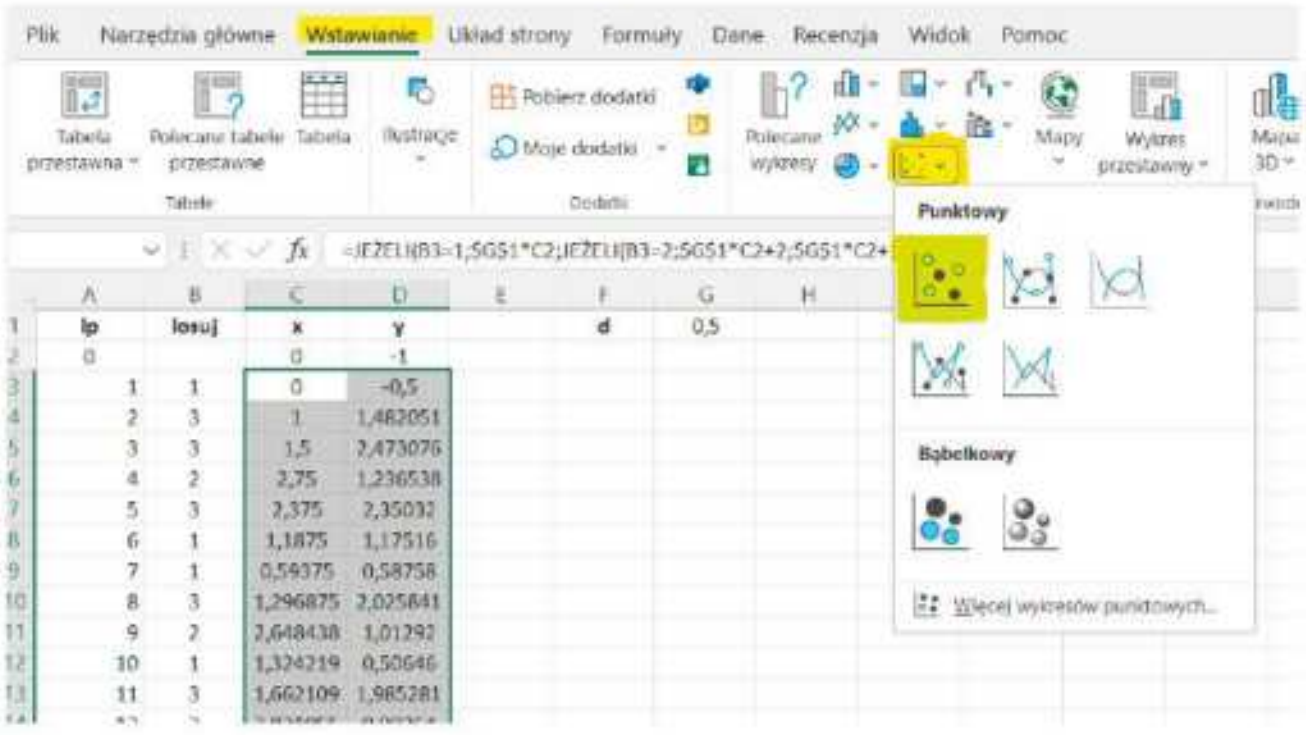
Teraz skorzystaj z szybkiego kopiowania, aby przekopiować formuły z komórek C3 i D3. Możesz jednocześnie przekopiować formuły z obu komórek:

Cofnij | Schowek | Czcionka | Wyrównanie

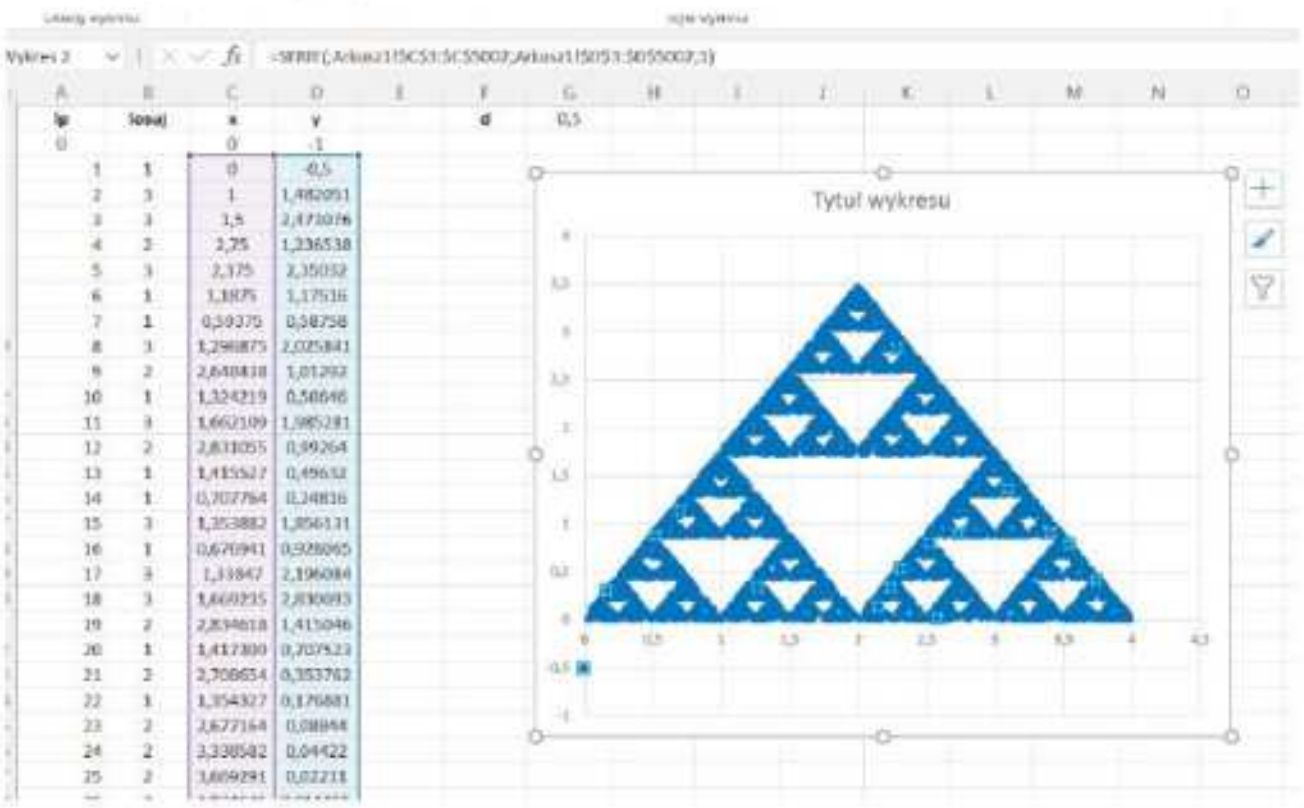
C3 | =JEŻELI(B3=1;$G$1*C2;JEŻELI(B3=2;$G$1*C2+2;$G$1*C2+1))

	A	B	C	D	E	F	G	H	I
1	**lp**	**losuj**	**x**	**y**		**d**	0,5		
2	0		0	-1					
3	1	1	0	-0,5					
4	2	1							
5	3	3							
6	4	1							
7	5	2							

Mamy już obliczone wszystkie 5000 punktów. Teraz wystarczy zaznaczyć je w układzie współrzędnych. Do tego użyjemy wykresu punktowego. Zaznaczamy blok komórek C3:D5002, używając szybkiego zaznaczania, tzn. zaznaczamy komórki C3 i D3, po czym – trzymając jednocześnie wciśnięte klawisze **Shift+Ctrl** – wciskamy ↓. Po zaznaczeniu całego bloku komórek wybieramy opcję **Wstawianie/Wykres punktowy**.



I tak otrzymaliśmy wykres:



lp	losuj	x	y	d	0,5
0		0	-1		
1	1	0	-0,5		
2	3	1	1,482091		
3	3	1,5	2,473076		
4	2	2,25	1,236538		
5	3	2,375	2,35032		
6	1	1,1875	1,17516		
7	1	0,59375	0,08758		
8	3	1,296875	2,025841		
9	2	2,648438	1,012921		
10	1	1,324219	0,50646		
11	3	1,662109	1,985281		
12	2	2,831055	0,99264		
13	1	1,415527	0,49632		
14	1	0,707764	0,24816		
15	3	1,353882	1,856131		
16	1	0,676941	0,928065		
17	3	1,33847	2,196084		
18	3	1,669235	2,830693		
19	2	2,834618	1,415346		
20	1	1,417309	0,707523		
21	2	2,708654	0,353762		
22	1	1,354327	0,176881		
23	2	2,677164	0,08844		
24	2	2,338582	0,04422		
25	2	1,669291	0,02211		

Powstała figura to trójkąt Sierpińskiego.

iterowane przekształcenia

Do wygenerowania tej figury wykorzystaliśmy układ iterowanych przekształceń funkcji liniowych. Jest to matematyczna metoda rozwiązania równania, w tym wypadku dwóch zmiennych $x=f(x,y)$ i $y=g(x,y)$:

$$x'=a*x+b*y+c$$
$$y'=d*x+e*y+f$$

polegająca na wyznaczaniu kolejnych przybliżeń rozwiązania:

$$x_0, y_0, x_1, y_1, \ldots\ldots, x_n, y_n$$

x_0, y_0 – wybieramy dowolnie

x_1, y_1 – obliczmy dzięki przekształceniu x_0, y_0

x_2, y_2 – dzięki przekształceniu x_1, y_1

x_{n+1}, y_{n+1} – dzięki przekształceniu x_n, y_n

Do skonstruowania trójkąta Sierpińskiego użyliśmy trzech układów równań. Każdy nowo obliczony punkt jest obrazem poprzedniego przekształconego przez jedno z trzech wylosowanych odwzorowań. Kolejne punkty pojawiają się w sposób losowy w zbiorze punktów tego układu przekształceń. Zbiór ten wykazuje cechy samopodobieństwa. Tak powstała figura jest jednym z najbardziej znanych fraktali. Fraktal to w znaczeniu potocznym obiekt samopodobny. Układ iterowanych przekształceń jest jedną z najprostszych metod generowania fraktali.

ZADANIA DO WYKONANIA

1. Poszukaj informacji, kim był Wacław Sierpiński i w którym roku podał konstrukcję zbioru nazwanego później trójkątem Sierpińskiego.
2. W arkuszu kalkulacyjny wygeneruj zbiór dwóch odwzorowań:

a) $\begin{cases} x' = -0{,}4 * x - 1 \\ y' = -0{,}4 * y + 0{,}1 \end{cases}$ b) $\begin{cases} x' = -0{,}76 * x - 0{,}4 * y \\ y' = -0{,}4 * x + 0{,}76 * y \end{cases}$

3. Sprawdź, jak działa funkcja `LOS()`.

PODSUMOWANIE LEKCJI

- Do szybkiego wypełniania komórek podobnymi wartościami służy opcja **Narzędzia główne/Wypełnij/Seria danych**. s. 75
- Do generowania losowych wartości służą funkcje `LOS()`, `LOS.ZAKR()`. s. 80

15. Podstawowe funkcje tekstowe. Korzystanie z gotowych funkcji, czyli jak łatwo modyfikować dane tekstowe

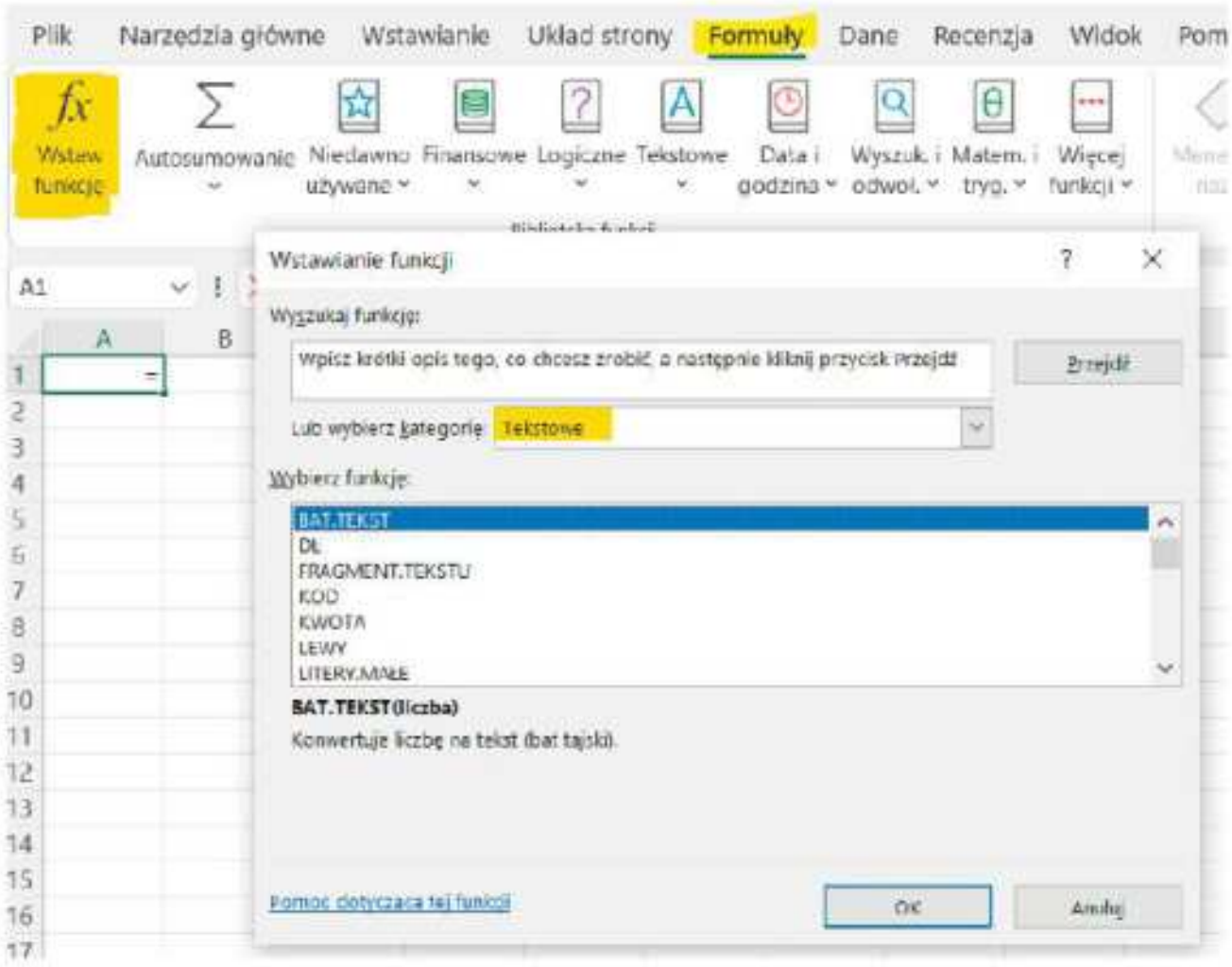
NA TEJ LEKCJI:

- nauczysz się, jak korzystać z funkcji operujących na tekście.

Funkcje na tekście

W poprzednich rozdziałach wskazano, że jednym z typów danych wpisywanych do arkusza jest tekst.

W arkuszu kalkulacyjnym mamy funkcje, które pozwalają nam wykonać wiele operacji na tekście. Możesz je przejrzeć po wybraniu z **Formuły/Wstaw funkcję** kategorii **Tekstowe**.

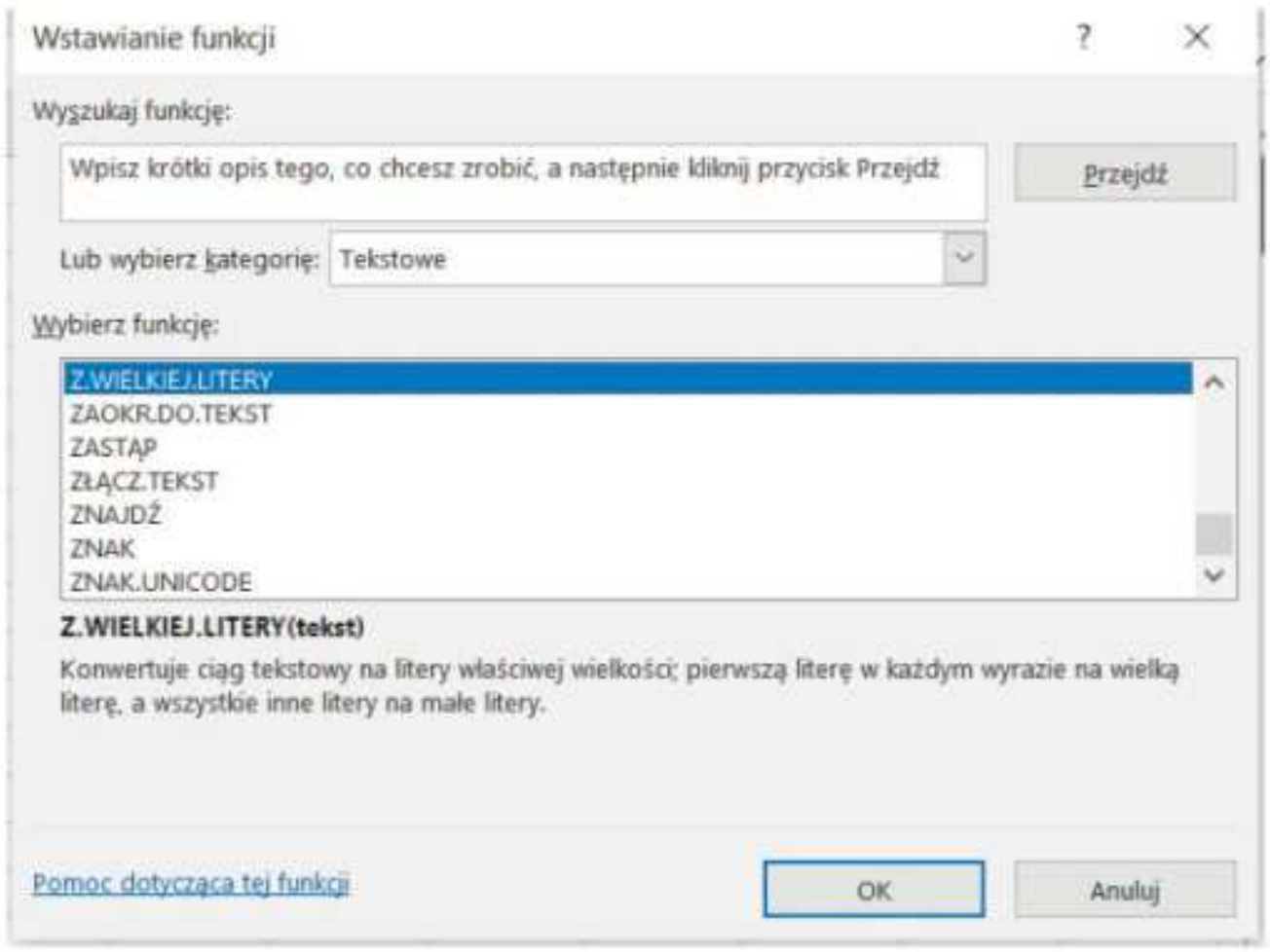


Na potrzeby tej lekcji plik obrazy.xlsx przygotuje ci nauczyciel.

W tym pliku zawarto informacje o obrazach. Są to: *Tytuł*, *Imię Malarza*, *Nazwisko Malarza*, *Miasto* (w którym na stałe znajduje się obraz) i *Stan* (informacja o tym, czy obraz jest w magazynie, wypożyczony czy na ekspozycji).

	A	B	C	D	E
1	Tytuł	Imie Malarza	Nazwisko Malarza	Oddział	Stan
2	Lanie Skarg	Jan	Matejko	Krakow	ekspozycja czasowa
3	Konrad Wallenrod	Jan	Matejko	Krakow	ekspozycja stała
4	Stanczyk w czasie balu	Jan	Matejko	Warszawa	ekspozycja czasowa
5	Astronom Kopernik	Jan	Matejko	Wroclaw	w magazynie
6	Batory pod Pskowem	Jan	Matejko	Warszawa	w magazynie
7	Unia Lubelska	Jan	Matejko	Wroclaw	ekspozycja stała
8	Hołd pruski	Jan	Matejko	Warszawa	wypożyczony
9	sobieski pod Wiedniem	Jan	Matejko	Krakow	ekspozycja stała
10	Wernyhora	Jan	Matejko	Warszawa	ekspozycja stała
11	smierc Zygmunta Augusta	Jan	Matejko	Krakow	w magazynie
12	Bitwa pod Racławicami	Jan	Matejko	Nieborow	ekspozycja stała
13	Konstytucja 3 maja 1791 roku	Jan	Matejko	Warszawa	ekspozycja stała
14	sluby Jana Kazimierza	Jan	Matejko	Warszawa	ekspozycja czasowa
15	Bitwa pod Grunwaldem	Jan	Matejko	Warszawa	ekspozycja stała
16	Chrzest Litwy	Jan	Matejko	Wroclaw	ekspozycja stała
17	Bitwa pod Pskowem	Jan	Matejko	Wroclaw	wypożyczony
18	Traktat chocimski	Marcello	Bacciarelli	Nieborow	ekspozycja stała
19	Sobiński pod Wiedniem	Marcello	Bacciarelli	Wroclaw	w magazynie

Każdy tytuł powinien zaczynać się wielką literą. Wprowadzający dane w niektórych przypadkach się pomylił. Poprawmy dane tak, aby tytuł zaczynał się wielką literą. Popatrzmy na funkcje tekstowe dostępne w arkuszu. Jedna z nich to funkcja `Z.WIELKIEJ.LITERY()`, która zamieni pierwszą literę w każdym słowie na wielką literę, a wszystkie pozostałe litery – na małe litery. Nie o to nam chodzi w tym wypadku, ponieważ tytuł „Stanczyk w czasie balu" zostanie przekonwertowany przez tę funkcję na „Stanczyk W Czasie Balu".



Przeglądając dostępne funkcje tekstowe, widzimy dostępne funkcje:

podstawowe funkcje na tekście

- `LITERY.WIELKIE(tekst)` – która konwertuje ciąg tekstowy na wielkie litery;
- `LEWY(tekst; liczba znaków)` – która zwraca określoną liczbę znaków od początku ciągu liczbowego. Np. `LEWY(„domek";3)` zwróci „dom"
- `PRAWY(tekst; liczba znaków)` – która zwraca określoną liczbę znaków od końca ciągu liczbowego. Np. `PRAWY(„domek";3)` zwróci „mek";
- `DŁ(tekst)` – zwraca liczbę znaków w tekście, np. `DŁ(„Ala ma kota")` da nam wynik 11;
- `FRAGMENT.TEKSTU(tekst;nr_poz_pocz;liczba_znaków)` – która zwraca znaki ze środka ciągu tekstowego przy danej pozycji początkowej i długości. Np. `FRAGMENT.TEKSTU(„Ala ma kota";4;2)` zwróci „ma";

Uwaga!

Znaki w ciągu numerujemy od 1.

- `ZŁĄCZ.TEKSTY(tekst1; tekst2; tekst3;...)` – łączy listę ciągów tekstowych.

Teraz wystarczy, że korzystając z tych funkcji, podzielimy tytuł na dwa podsłowa – jedno złożone tylko z jednej pierwszej litery, a drugie złożone z pozostałych liter tytułu.

Przekopiujemy kolumnę *Tytuł* do innego arkusza, który nazwiemy *tytuł*.

B2

	A	B	C
1	Tytul	Podslowo 1	Podslowo 2
2	kazanie Skargi		
3	Konrad Wallenrod		
4	Stanczyk w czasie balu		
5	Astornom Kopernik		
6	Batory pod Pskowem		
7	Unia Lubelska		
8	Hold pruski		
9	sobieski pod Wiedniem		
10	Wernyhora		
11	smierc Zygmunta Augusta		
12	Bitwa pod Raclawicami		
13	Konstytucja 3 maja 1791 roku		
14	sluby Jana Kazimierza		
15	Bitwa pod Grunwaldem		
16	Chrzest Litwy		
17	Bitwa pod Pskowem		
18	Traktat chocimski		
19	Sobieski pod Wiedniem		
20	Hold pruski		
21	Unia lubelska		
22	Krol Wladyslaw Jagiello		
23	Stanislaw Poniatowski z zona		
24	Portret Stanislawa Augusta		
25	Autoportret w konfederatce		
26	Stanislaw August w stroju koronacyjnym		
27	Kolumna Zygmunta III		
28	Widok Warszawy od strony Pragi		

obrazy | tytul | Arkusz2

Teraz wyznaczymy podsłowo 1, używając funkcji `LEWY()`.

Z.WIELKIEJ... =LEWY(A2;1)

	A	B	C
1	Tytul	Podslowo 1	Podslowo 2
2	kazanie Skargi	=LEWY(A2;1)	
3	Konrad Wallenrod		
4	Stanczyk w czasie balu		
5	Astornom Kopernik		

Wyznaczymy także podsłowo 2, wykorzystując funkcję `PRAWY()`. Wytniemy z prawej wszystkie znaki bez pierwszego, czyli `DŁ(tekst) - 1`.

	A	B	C	D
1	Tytul	Podsłowo 1	Podsłowo 2	Zamien 1
2	kazanie Skargi	k	=PRAWY(A2;DŁ(A2)-1)	
3	Konrad Wallenrod			
4	Stanczyk w czasie balu			

Teraz na podsłowie 1 zastosujemy funkcję `LITERY.WIELKIE()`.

	A	B	C	D	E	F
1	Tytul	Podsłowo 1	Podsłowo 2	Zamien 1	Połacz	
2	kazanie Skargi	k	azanie Skargi	=Z.WIELKIEJ.LITERY(B2)		
3	Konrad Wallenrod					
4	Stanczyk w czasie balu					
5	Astornom Kopernik					

Następnie połączymy podsłowo 1 z podsłowem 2 w jeden tekst.

	A	B	C	D	E	F
1	Tytul	Podsłowo 1	Podsłowo 2	Zamien 1	Połacz	
2	kazanie Skargi	k	azanie Skargi	K	=ZŁĄCZ.TEKST(D2;C2)	
3	Konrad Wallenrod					
4	Stanczyk w czasie balu					

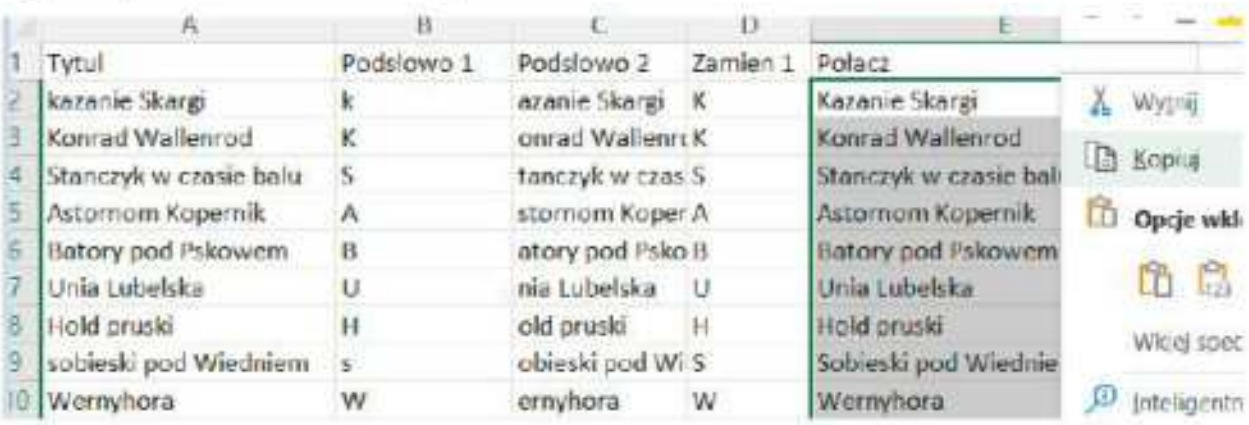
Teraz przekopiujemy te formuły na całą kolumnę *Tytuł*. Pamiętaj o szybkim kopiowaniu, tj. zaznacz komórki B2:E2, ustaw się w prawym dolnym rogu bloku i kliknij dwa razy.

	A	B	C	D	E	F
1	Tytul	Podsłowo 1	Podsłowo 2	Zamien 1	Połacz	
2	kazanie Skargi	k	azanie Skargi	K	Kazanie Skargi	
3	Konrad Wallenrod					
4	Stanczyk w czasie balu					
5	Astornom Kopernik					

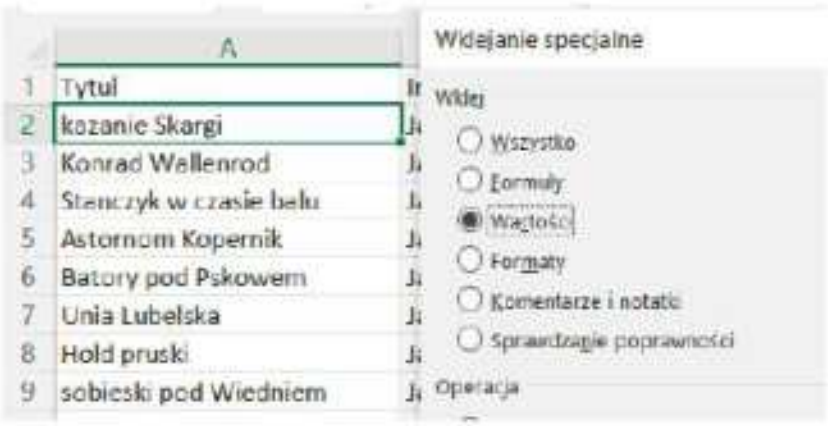
W kolumnie *Połącz* będziemy mieli właściwy tytuł:

	A	B	C	D	E	F
1	Tytul	Podsłowo 1	Podsłowo 2	Zamien 1	Połacz	
2	kazanie Skargi	k	azanie Skargi	K	Kazanie Skargi	
3	Konrad Wallenrod	K	onrad Wallenr	K	Konrad Wallenrod	
4	Stanczyk w czasie balu	S	tanczyk w czas	S	Stanczyk w czasie balu	
5	Astornom Kopernik	A	stornom Koper	A	Astornom Kopernik	
6	Batory pod Pskowem	B	atory pod Psko	B	Batory pod Pskowem	
7	Unia Lubelska	U	nia Lubelska	U	Unia Lubelska	
8	Hold pruski	H	old pruski	H	Hold pruski	
9	sobieski pod Wiedniem	s	obieski pod Wi	S	Sobieski pod Wiedniem	
10	Wernyhora	W	ernyhora	W	Wernyhora	
11	smierc Zygmunta Augusta	s	mierc Zygmunt	S	Smierc Zygmunta Augusta	
12	Bitwa pod Raclawicami	B	itwa pod Racla	B	Bitwa pod Raclawicami	
13	Konstytucja 3 maja 1791 r	K	onstytucja 3 m	K	Konstytucja 3 maja 1791 roku	
14	sluby Jana Kazimierza	s	luby Jana Kazin	S	Sluby Jana Kazimierza	
15	Bitwa pod Grunwaldem	B	itwa pod Grun	B	Bitwa pod Grunwaldem	
16	Chrzest Litwy	C	hrzest Litwy	C	Chrzest Litwy	
17	Bitwa pod Pskowem	B	itwa pod Psko	B	Bitwa pod Pskowem	
18	Traktat chocimski	T	raktat chocims	T	Traktat chocimski	
19	Sobieski pod Wiedniem	S	obieski pod Wi	S	Sobieski pod Wiedniem	
20	Hold pruski	H	old pruski	H	Hold pruski	
21	Unia lubelska	U	nia lubelska	U	Unia lubelska	
22	Krol Wladyslaw Jagiello	K	rol Wladyslaw	K	Krol Wladyslaw Jagiello	
23	Stanislaw Poniatowski z z	S	tanislaw Ponia	S	Stanislaw Poniatowski z zona	
24	Portret Stanislawa August	P	ortret Stanisla	P	Portret Stanislawa Augusta	
25	Autoportret w konfedera	A	utoportret w k	A	Autoportret w konfederatce	
26	Stanislaw August w stroju	S	tanislaw Augus	S	Stanislaw August w stroju korona	yjnym

Teraz wystarczy tylko przekopiować blok E2:E340 do arkusza *obrazy* do kolumny *Tytuł*. Pamiętaj o szybkim zaznaczaniu: ustaw się w komórce E2 i trzymając jednocześnie wciśnięty klawisz **Shift** i **Ctrl**, naciśnij strzałkę w dół ↓. Tak wybrany blok zaznacz do kopiowania.



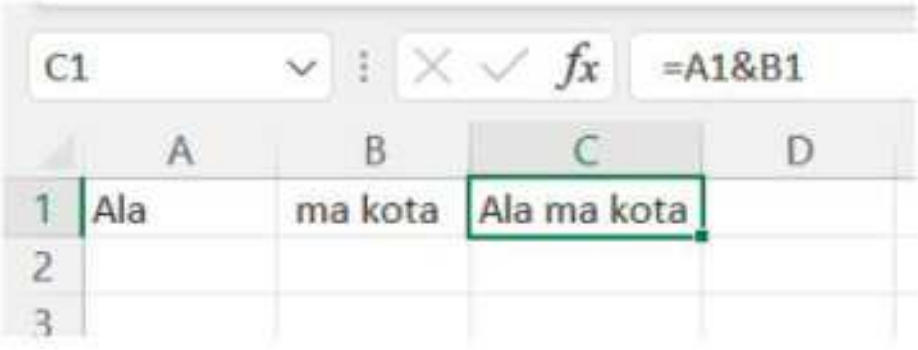
Przejdź do arkusza *obrazy*, ustaw się w komórce A2 i wybierz **Wklejanie specjalne/Wartość**. To spowoduje zastąpienie tytułów z błędem właściwymi tytułami.



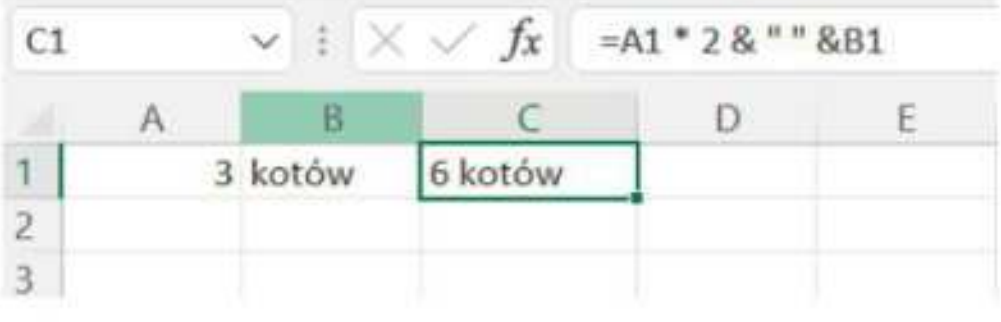
Uwaga!

Do łączenia tekstów możemy użyć „&”.

Przykład 1.



Przykład 2.



ZADANIA DO WYKONANIA

1. Dla każdego obrazu wyznacz kod malarza, który go namalował. Kod malarza składa się z pierwszej litery imienia i pierwszej litery nazwiska.

PODSUMOWANIE LEKCJI

• Podstawowe funkcje na tekście to:

s. 83

Funkcja	Opis
DŁ	Zwraca liczbę znaków ciągu tekstowego.
LEWY	Zwraca określoną liczbę znaków z ciągu tekstowego z lewej strony.
PRAWY	Zwraca określoną liczbę znaków z ciągu tekstowego z prawej strony.
FRAGMENT.TEKSTU	Zwraca określoną liczbę znaków z ciągu tekstowego, zaczynając od zadanej pozycji.
ZNAJDŹ	Znajduje jedną wartość tekstową wewnątrz innej (z uwzględnieniem wielkich i małych liter).
ZŁĄCZ.TEKSTY	Łączy tekst z wielu zakresów i (lub) ciągów, ale nie zapewnia argumentów ignorowania pustych ani ogranicznika.
LITERY.WIELKIE	Konwertuje znaki tekstu na wielkie litery.
TEKST	Formatuje liczbę i konwertuje ją na tekst.

Wszystkie dostępne funkcje z określonych kategorii są widoczne w **Formuły/Wstaw Funkcję/Kategorie**.

16. Podstawowe funkcje dotyczące daty i czasu. Korzystanie z gotowych funkcji, czyli jak łatwo pracować z datą

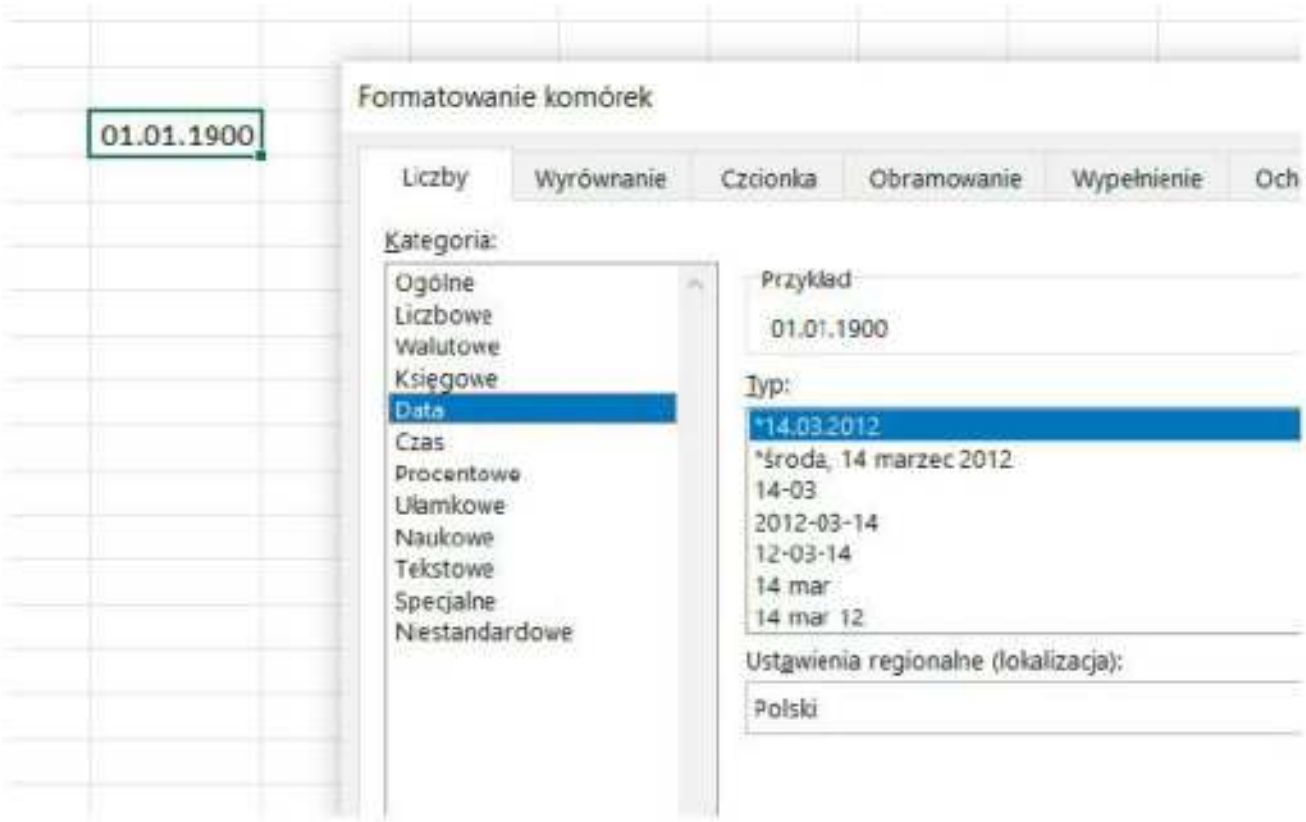
NA TEJ LEKCJI:

- poznasz podstawowe funkcje operujące na dacie i czasie.

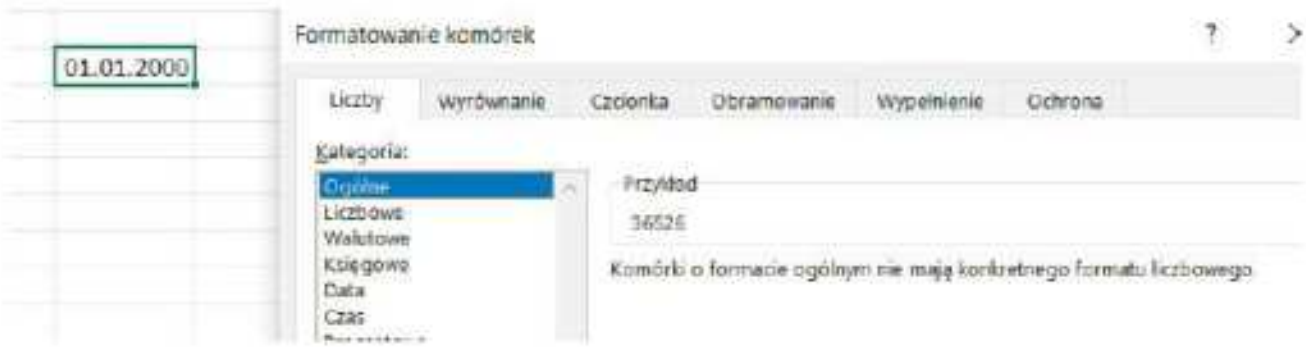
Funkcje operujące na dacie/czasie

podstawowe funkcje

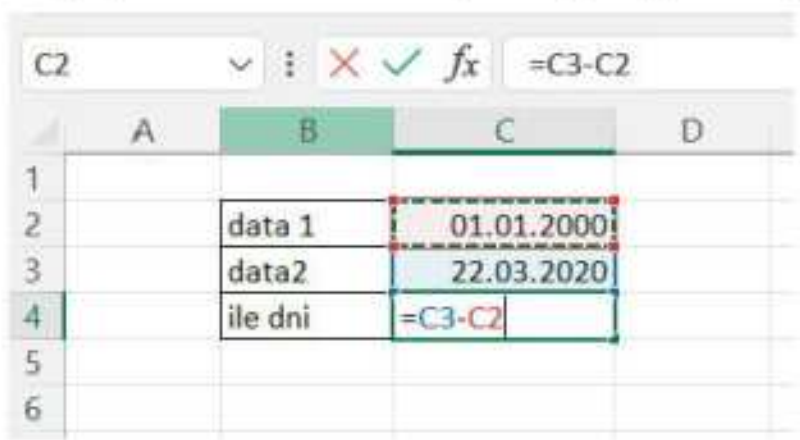
Dla arkusza kalkulacyjnego data to liczba – liczba dni, które upłynęły od 1 stycznia 1900 r. O tym, jaka data będzie wyświetlana, decyduje format komórki, który ustawimy.

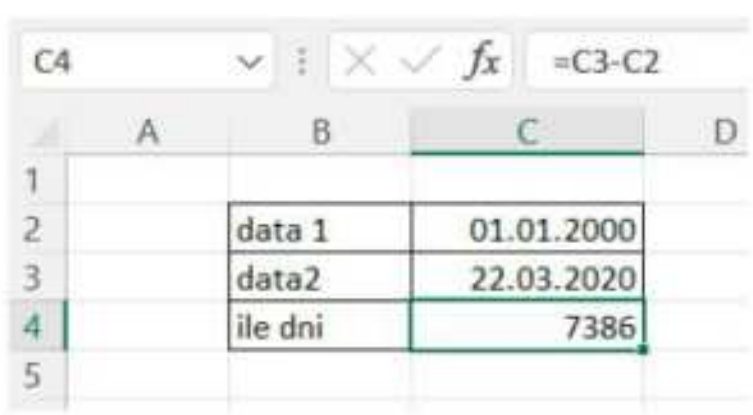


Jeśli wybierzemy format ogólny lub liczbowy, będziemy widzieli liczbę dni, która upłynęła od daty 1.01.1900.



Fakt, że daty to liczby, pozwala nam w łatwy sposób policzyć, ile dni upłynęło między dwiema datami. Wystarczy odjąć te daty:

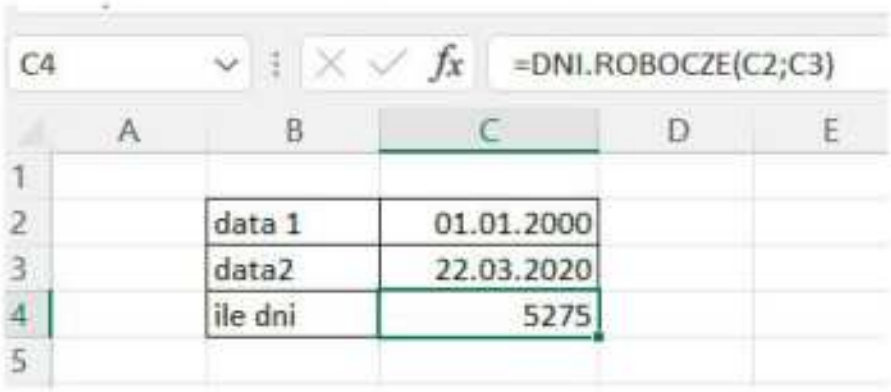




Uwaga!

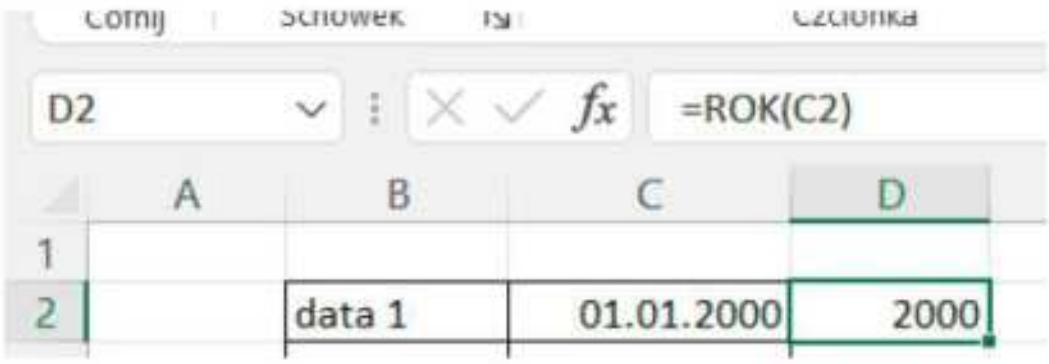
Pamiętaj, że komórka C4 powinna mieć ustawiony format ogólny lub liczbowy.

Jeśli chcemy dowiedzieć się, ile jest dni roboczych między datami, użyjemy funkcji `DNI.ROBOCZE(data1;data2)`. Pomija ona weekendy i dni wolne przypadające w okresie wyznaczonym przed data1 i data2.

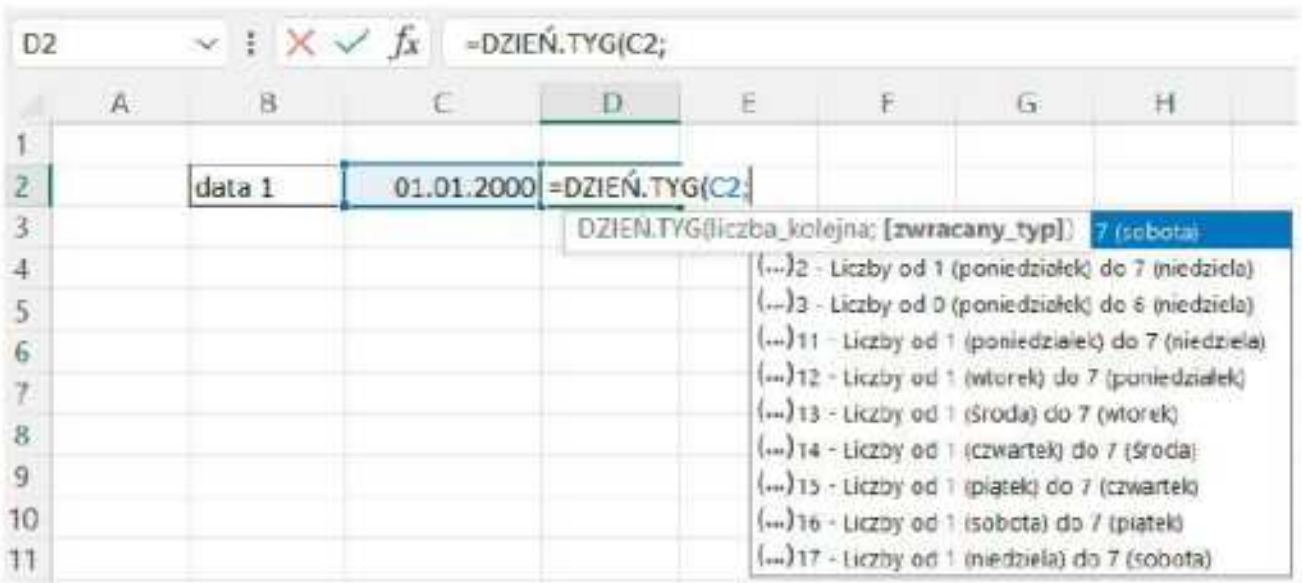


Jeśli chcemy wyznaczyć rok, miesiąc lub dzień, użyjemy funkcji: `ROK()`, `MIESIAC()`, `DZIEŃ()`.

Na przykład:



Do wyznaczenia dnia tygodnia użyjemy funkcji `DZIEŃ.TYG(data;zwracany typ)`.

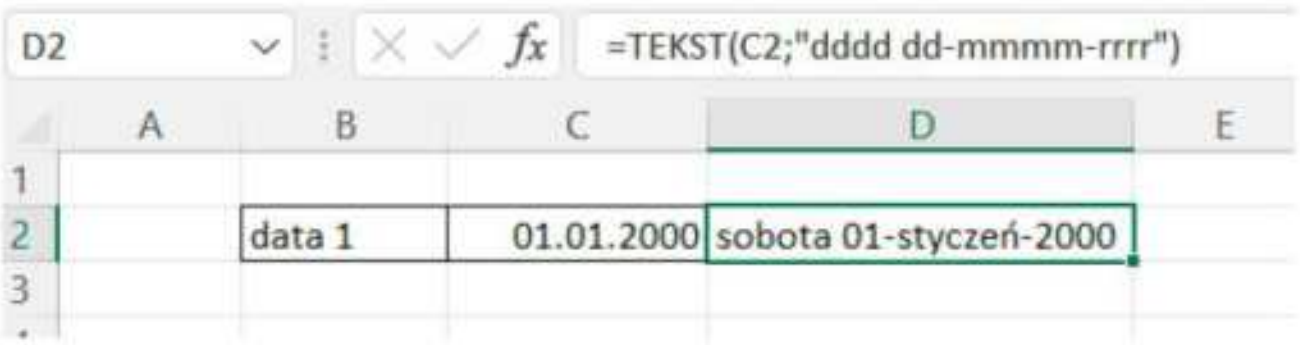


Jeśli jako zwracany typ wybierzemy parametr 1, to niedziela będzie reprezentowana przez 1, poniedziałek – przez 2, a sobota – przez 7.

Jeśli chcemy, aby dzień tygodnia był przedstawiony słownie, możemy użyć funkcji `TEKST()`. Funkcja `TEKST` pozwala na zmianę sposobu wyświetlania liczby przez zastosowanie do niej formatowania za pomocą kodów formatów.

Jest to przydatne w sytuacjach, w których warto wyświetlić liczby w bardziej czytelnym formacie lub połączyć liczby z tekstem czy symbolami.

Formuła `=TEKST(C2;"dddd dd-mmmm-rrrr")` wyświetli nam datę w postaci:



Zatem jeśli dla podanej daty chcemy podać słownie dzień tygodnia, wystarczy napisać `=TEKST(D2;"dddd")`. Jeśli chcemy podać słownie nazwę miesiąca, wystarczy napisać `=TEKST(D2,"mmmm")`.

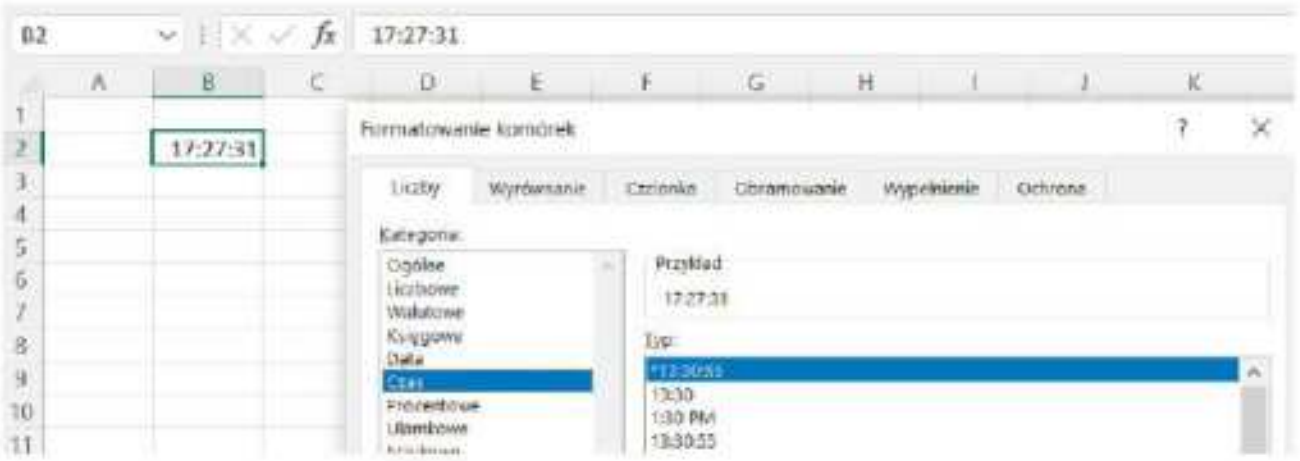
Uwaga!

Funkcja `TEKST` przekonwertuje liczby (data to liczba) na tekst. Jeśli chcemy odwoływać się do nich w dalszych obliczeniach, wtedy można zachować wartość liczbową (np. datę) w jednej komórce, a funkcji `TEKST` użyć w innej komórce.

Funkcje operujące na czasie

W arkuszu kalkulacyjnym godziny są przechowywane jako liczby dziesiętne z zakresu od 0,0 do 0,99999, przy czym 0,0 reprezentuje godzinę 00:00:00, a 0,99999 – godzinę 23:59:59. Jedna godzina to 1/24 doby. Jeśli chcemy zamienić ją na liczbę całkowitą, należy pomnożyć ją przez 24 (liczba godzin w dobie).

Sposób wyświetlania godziny w arkuszu ustawiamy w **Formatowanie komórek/ Czas**.



Funkcje, które pozwalają nam z podanej godziny wyznaczyć minuty, sekundy lub godziny, to `GODZINA()`, `MINUTA()`, `SEKUNDA()`.

E2 =SEKUNDA(B2)

	A	B	C	D	E	F
1			godzina	minuta	sekunda	
2		17:27:31	17	27	=SEKUNDA(B2)	
3						
4						

Uwaga!

Jeśli chcemy połączyć datę i czas, wystarczy je do siebie dodać.

C2 =A2+B2

	A	B	C	D
1	data	czas	+	
2	01.04.2022	17:40:23	01.04.2022 17:40:23	
3				
4				

W ten sposób otrzymamy datę długą zawierającą datę i czas.

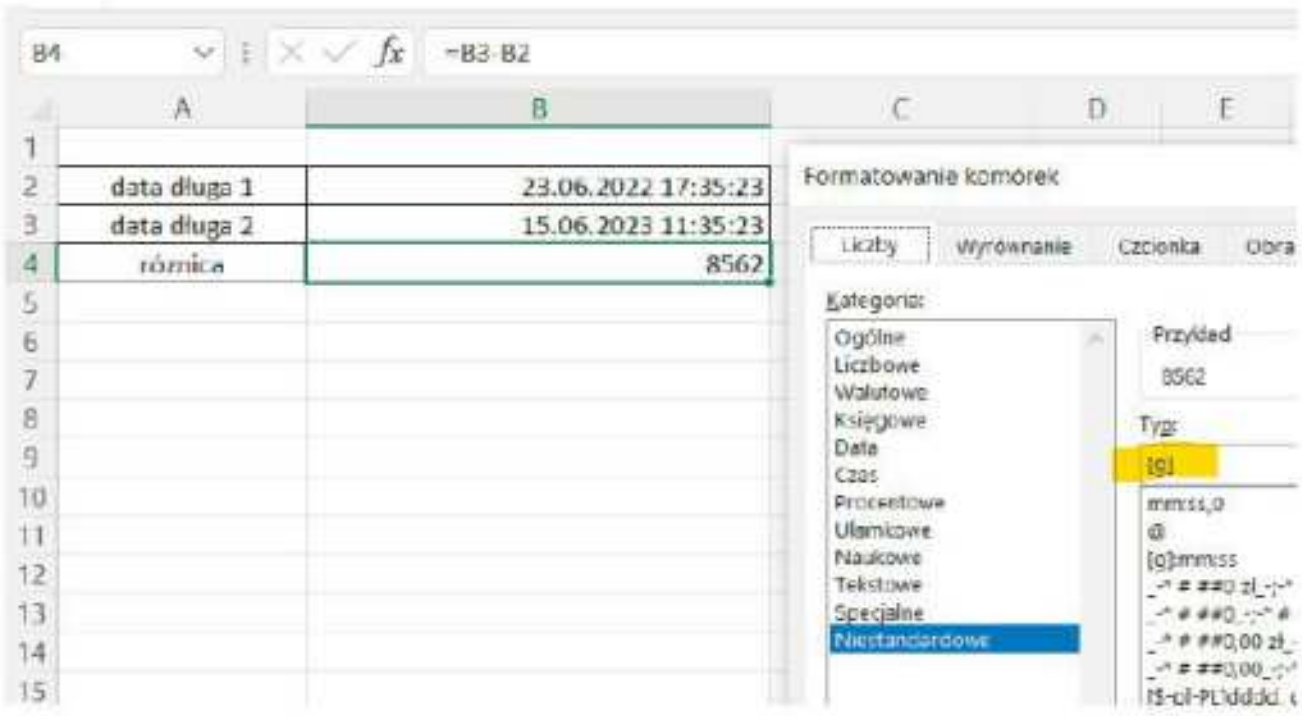
Uwaga!

Po odjęciu od siebie dwóch dat długich otrzymamy liczbę rzeczywistą.

B4 =B3-B2

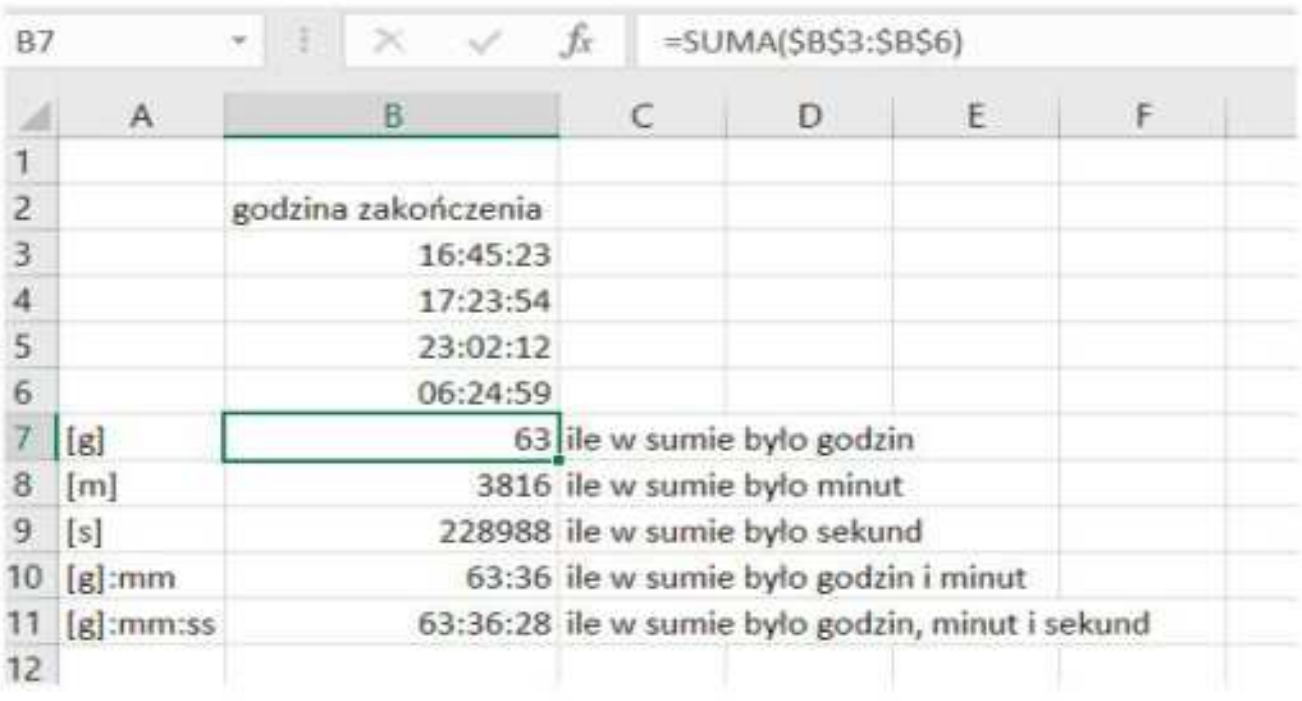
	A	B
1		
2	data długa 1	23.06.2022 17:35:23
3	data długa 2	15.06.2023 11:35:23
4	róznica	356,75
5		
6		

Łatwym sposobem przedstawienia różnicy w godzinach jest formatowanie niestandardowe.



Użycie typu [g] wymusi wyświetlanie wyniku w godzinach. Gdybyśmy chcieli wynik otrzymać w minutach, użylibyśmy typu [m].

W komórkach B3:B6 mamy czas zakończenia aktywności, które rozpoczęły się o godzinie 00:00:00. Jeśli chcemy wiedzieć, ile czasu łącznie zajęły te aktywności, wystarczy zsumować komórki od B3 do B6 (`=SUMA(B3:B6)`).



Następnie trzeba ustawić w **Formatowanie komórek/Niestandardowe**, w jakich jednostkach chcemy dostać wynik.

ZADANIA DO WYKONANIA

1. W pliku data.xlsx (który otrzymasz od nauczyciela) przedstawiono daty w postaci 03052022; 03 oznacza dzień, 05 – miesiąc, a 2022 – rok. Jest to zapis nierozpoznawalny przez arkusz jako data. Użyj funkcji `DATA()`, `LEWY()`, `PRAWY()` i `FRAGMENT.TEKSTU()`, aby przedstawić te daty w formacie traktowanym przez arkusz jako data.

PODSUMOWANIE LEKCJI

• Podstawowe funkcje na dacie i czasie to:

s. 87

Funkcja	Opis
DATA	Zwraca liczbę kolejną dla wybranej daty.
ROK	Konwertuje liczbę kolejną na rok.
MIESIĄC	Konwertuje liczbę kolejną na miesiąc.
DZIEŃ	Konwertuje liczbę kolejną na dzień.
DNI.ROBOCZE	Zwraca liczbę pełnych dni roboczych między dwiema datami.
TERAZ	Zwraca liczbę kolejną bieżącej daty i godziny.
GODZINA	Konwertuje liczbę kolejną na godzinę.
MINUTA	Konwertuje liczbę kolejną na minutę.
SEKUNDA	Konwertuje liczbę kolejną na sekundę.

Wszystkie dostępne funkcje z określonej kategorii widoczne są w **Formuły/Wstaw Funkcję/Kategorie**. Formatowanie komórek pozwala nam przedstawić datę i czas w takiej postaci, jakiej potrzebujemy.

17. Jak szukać w Excelu, czyli gdzie to jest

NA TEJ LEKCJI:

- dowiesz się, jak wyszukiwać wartości, korzystając z funkcji `WYSZUKAJ`;
- samodzielnie nauczysz się używać funkcji `WYSZUKAJ.PIONOWO`, `WYSZUKAJ.POZIOMO`.

Wyszukiwanie wartości

Funkcja `WYSZUKAJ()` umożliwia przeszukanie jednej kolumny lub jednego wiersza w celu znalezienia wartości na tej samej pozycji w drugim wierszu lub drugiej kolumnie.

Składnia:

`WYSZUKAJ`([szukana wartość]; [przeszukiwany wektor]; [wektor wynikowy])

[szukana wartość] – to wartość, której funkcja szuka w zakresie [przeszukiwany wektor].

szukana wartość

Szukana wartość może być liczbą, tekstem, wartością logiczną, nazwą lub odwołaniem do wartości.

[przeszukiwany wektor] – przeszukiwany zakres (wektor). Jest to jeden wiersz lub jedna kolumna.

konieczność sortowania

Wartości w przeszukiwanym wektorze muszą być posortowane rosnąco.

[wektor wynikowy] – zakres (wektor), z którego wartość zostanie zwrócona, jeśli poszukiwana wartość zostanie znaleziona w przeszukiwanym wektorze.

Uwaga!

Zakres zaznaczony jako [wektor wynikowy] musi mieć taki sam rozmiar, jak [przeszukiwany wektor].

Przeanalizujmy przykłady.

Przykład 1

Dla danego kodu produktu chcemy podać jego cenę.

	A	B	C	D	E	F
1	kod	nazwa	cena		kod produktu	cena
2	K1	Produkt 1	6,24		K10	
3	K2	Produkt 2	12			
4	K3	Produkt 3	34,99			
5	K4	Produkt 4	45,5			
6	K5	Produkt 5	67			
7	K6	Produkt 6	39,99			
8	K7	Produkt 7	36,12			
9	K8	Produkt 8	71,20			
10	K9	Produkt 9	76,22			
11	K10	Produkt 10	55,14			
12	K11	Produkt 11	55,07			
13	K12	Produkt 12	83,43			
14	K13	Produkt 13	45,46			
15	K14	Produkt 14	60,37			
16	K15	Produkt 15	29,22			
17	K16	Produkt 16	74,60			
18	K17	Produkt 17	94,81			

W komórce F2 zapiszemy `=WYSZUKAJ(E2;A2:A18;C2:C18)`.

[szukana wartość] to zawartość komórki E2.

[przeszukiwany wektor] – zakres A2:A18, w którym będziemy szukać wartości z komórki E2.

[wektor wynikowy] – zakres C2:C18. Jeśli [szukana wartość] znajduje się z zakresie A2:A18 na pozycji *x*, to zostanie zwrócona wartość na pozycji *x* z zakresu C2:C18.

Uwaga!

Funkcja `WYSZUKAJ()` pozwala na pobieranie danych z zakresu na lewo i powyżej przeszukiwanego obszaru.

F2 =WYSZUKAJ(E2;B2:B18;A2:A18)

	A	B	C	D	E	F	G
1	kod	nazwa	cena		Nazwa	kod	
2	K1	Produkt 1	6,24		Produkt 13	K13	
3	K10	Produkt 10	55,14				
4	K11	Produkt 11	55,07				
5	K12	Produkt 12	83,43				
6	K13	Produkt 13	45,46				
7	K14	Produkt 14	60,37				
8	K15	Produkt 15	29,22				
9	K16	Produkt 16	74,60				
10	K17	Produkt 17	94,81				
11	K2	Produkt 2	12				
12	K3	Produkt 3	34,99				
13	K4	Produkt 4	45,5				
14	K5	Produkt 5	67				
15	K6	Produkt 6	39,99				
16	K7	Produkt 7	36,12				
17	K8	Produkt 8	71,20				
18	K9	Produkt 9	76,22				

Zauważ, że jeśli chcemy szukać w zakresie B2:B18, to zakres ten musi być posortowany.

Przykład 2

Wyjaśnij, co się stanie, jeśli poszukiwana wartość nie znajduje się w przeszukiwanym wektorze.

G2 =WYSZUKAJ(F2;D2:D18;A2:A18)

	A	B	C	D	E	F	G	H
1	kod	nazwa	cena	sprzedano/sztuki		ile sztuk	kod	
2	K12	Produkt 12	83,43	19		50	K6	
3	K17	Produkt 17	94,81	21				
4	K10	Produkt 10	55,14	30				
5	K5	Produkt 5	67	31				
6	K2	Produkt 2	12	36				
7	K8	Produkt 8	71,20	38				
8	K6	Produkt 6	39,99	44				
9	K3	Produkt 3	34,99	56				
10	K13	Produkt 13	45,46	58				
11	K7	Produkt 7	36,12	65				
12	K11	Produkt 11	55,07	66				
13	K9	Produkt 9	76,22	77				
14	K4	Produkt 4	45,5	79				
15	K1	Produkt 1	6,24	84				
16	K14	Produkt 14	60,37	88				
17	K16	Produkt 16	74,60	88				
18	K15	Produkt 15	29,22	91				

Szukamy kodu produktu, którego sprzedano 50 sztuk.

```
=WYSZUKAJ(F2;D2:D18;A2:A18)
```

Jeżeli funkcja `WYSZUKAJ` nie może znaleźć wartości określonej przez [szukana wartość], to zwraca największą wartość w przeszukiwanym bloku [przeszukiwany wektor], która jest mniejsza od argumentu [szukana wartość] lub równa temu argumentowi. Jeśli szukana wartość jest mniejsza od najmniejszej, to funkcja `WYSZUKAJ` zwraca wartość błędu #N/D.

W tym wypadku funkcja `WYSZUKAJ` znalazła wartość 44 w komórce D8 i zwróciła wartość K6 z komórki A8.

Przykład 3

Określmy poziom sprzedaży produktu jako:

- niski, jeśli sprzedano go mniej niż 30 sztuk,
- średni, jeśli sprzedano od 30 do 60 sztuk <30;60),
- wysoki, jeśli sprzedano 60 lub więcej sztuk.

Dla każdego produktu wyznaczmy poziom sprzedaży produktu.

E2 | =WYSZUKAJ(D2;$G$2:$H$4)

	A	B	C	D	E	F	G	H	I
1	kod	nazwa	cena	sprzedano/sztuki	poziom		zakres	poziom	
2	K1	Produkt 1	6,24	84	wysoki		0	niski	<0;30)
3	K10	Produkt 10	55,14	30			30	średni	<30;60)
4	K11	Produkt 11	55,07	66			60	wysoki	<60;...)
5	K12	Produkt 12	83,43	19					
6	K13	Produkt 13	45,46	58					
7	K14	Produkt 14	60,37	88					
8	K15	Produkt 15	29,22	91					
9	K16	Produkt 16	74,60	88					
10	K17	Produkt 17	94,81	21					
11	K2	Produkt 2	12	36					
12	K3	Produkt 3	34,99	56					
13	K4	Produkt 4	45,5	79					
14	K5	Produkt 5	67	31					
15	K6	Produkt 6	39,99	44					
16	K7	Produkt 7	36,12	65					
17	K8	Produkt 8	71,20	38					
18	K9	Produkt 9	76,22	77					
19									

W E2 wpiszmy `=WYSZUKAJ(D2;$G$2:$H$4)`.

W funkcji `WYSZUKAJ` użyliśmy dwóch parametrów:

- szukana wartość,
- tablica.

W tym przypadku funkcja `WYSZUKAJ` przeszuka pierwszą kolumnę tablicy i zwróci odpowiadającą wartość z ostatniej kolumny podanej tablicy.

`WYSZUKAJ(D2;$G$2:$H$4)` będzie szukała wartości 84 w pierwszej kolumnie tablicy $G$2:SHS4. W bloku do $G$2 do $G$4 znajdzie wartość 60, która jest największą wartością mniejszą od 84 lub jej równą, i zwróci odpowiednią wartość z ostatniej kolumny tablicy $G$2:$H$4. Ostatnia kolumna w tej tablicy to blok od $H$2 do $H$4.

Funkcja `WYSZUKAJ()` w takiej formie nie daje możliwości wskazania kolumny lub wiersza, z którego mają zostać zwrócone dane. Taką możliwość dają funkcje `WYSZUKAJ.PIONOWO()`, `WYSZUKAJ.POZIOMO()`.

Funkcja `WYSZUKAJ.PIONOWO` i `POZIOMO`

ZADANIA DO WYKONANIA

1. Zapoznaj się samodzielnie z opisem działania funkcji `WYSZUKAJ.PIONOWO` (pomoc arkusza Excel). Wykonaj wszystkie ćwiczenia z lekcji, używając funkcji `WYSZUKAJ.PIONOWO`. Plik produkty.xlsx otrzymasz od nauczyciela.

PODSUMOWANIE LEKCJI

- Funkcje `WYSZUKAJ`, `WYSZUKAJ.PIONOWO`, `WYSZUKAJ.POZIOMO` umożliwiają szukanie wartości w kolumnie, wierszu lub tablicy. s. 94
- W funkcji `WYSZUKAJ` obszar, w którym szukana jest wskazana wartość, musi być posortowany. s. 94
- Jeżeli funkcja `WYSZUKAJ` nie może znaleźć wartości określonej przez [szukana wartość], to zwraca największą wartość w przeszukiwanym bloku [przeszukiwany wektor], która jest mniejsza od argumentu [szukana wartość] lub równa temu argumentowi. Jeśli szukana wartość jest mniejsza od najmniejszej, to funkcja `WYSZUKAJ` zwraca wartość błędu #N/D. s. 97

18. Co ukrywa numer PESEL, czyli do czego przydaje się adres mieszany

NA TEJ LEKCJI:

- dowiesz się, jak z numeru PESEL wyznaczyć datę urodzenia;
- sprawdzisz, czy dany numer PESEL jest prawidłowy.

Co to jest numer PESEL

Numer PESEL to jedenastocyfrowy symbol numeryczny, który pozwala na łatwą identyfikację osoby, do której go przypisano. Numer PESEL zawiera datę urodzenia, numer porządkowy, oznaczenie płci oraz liczbę kontrolną.

Wyznaczanie daty urodzenia

PESEL jako tekst

Data urodzenia w numerze PESEL zapisana jest za pomocą pierwszych sześciu znaków numeru. Dwa pierwsze to rok urodzenia, znaki trzeci i czwarty to miesiąc urodzenia, a piąty i szósty to dzień urodzenia. Aby odróżnić od siebie numery PESEL z różnych stuleci, przyjęto następującą metodę oznaczania miesiąca urodzenia: numer PESEL traktujemy jak tekst, żeby nie utracić zer na początku numeru.

Miesiąc/Stulecie	1800–1899	1900–1999	2000–2099	2100–2199	2200–2299
styczeń	81	01	21	41	61
luty	82	02	22	42	62
marzec	83	03	23	43	63
kwiecień	84	04	24	44	64
maj	85	05	25	45	65
czerwiec	86	06	26	46	66
lipiec	87	07	27	47	67
sierpień	88	08	28	48	68
wrzesień	89	09	29	49	69
październik	90	10	30	50	70
listopad	91	11	31	51	71
grudzień	92	12	32	52	72

Źródło: https://www.gov.pl/

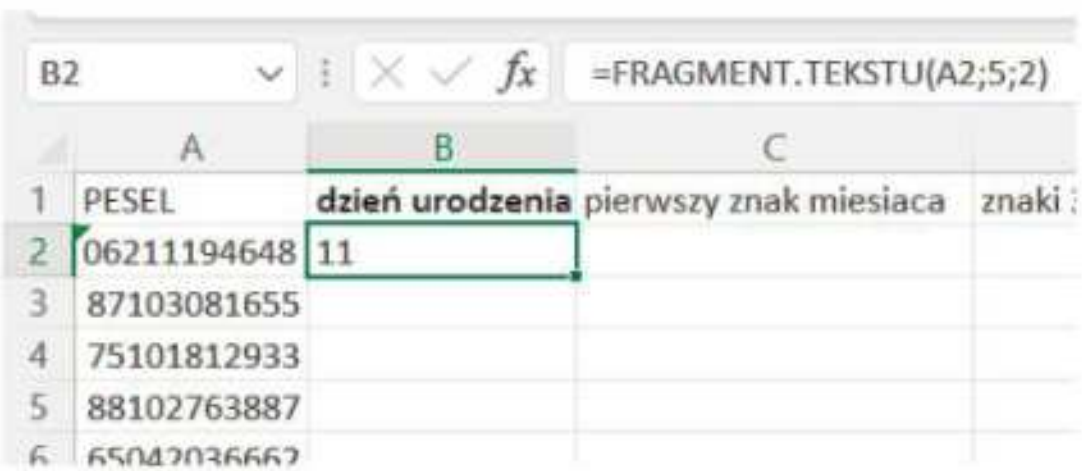
Osoba urodzona 13 stycznia 1921 roku będzie miała numer PESEL 210113xxxxx, a osoba urodzona 13 stycznia 2021 roku – 212113xxxxx.

Aby odczytać datę urodzenia osoby z numeru PESEL, wyznaczymy dzień, miesiąc i rok urodzenia.

fragment tekstu

Do wyznaczenia dnia urodzenia użyjemy funkcji `FRAGMENT.TEKSTU()`, aby wyciąć znaki piąty i szósty z numeru PESEL.

```
=FRAGMENT.TEKSTU(A2;5;2)
```



Funkcja `FRAGMENT.TEKSTU()` zwraca nam tekst.

Aby zamienić tekst na wartość liczbową, możemy skorzystać z funkcji `=WARTOŚĆ(FRAGMENT.TEKSTU(A2;5;2))` lub przemnożyć nasz wynik przez 1 `=FRAGMENT.TEKSTU(A2;5;2)*1`.

B2 =FRAGMENT.TEKSTU(A2;5;2) * 1

	A	B	C	
1	PESEL	**dzień urodzenia**	pierwszy znak miesiaca	znaki 3 i 4
2	06211194648	11		
3	87103081655			
4	75101812933			

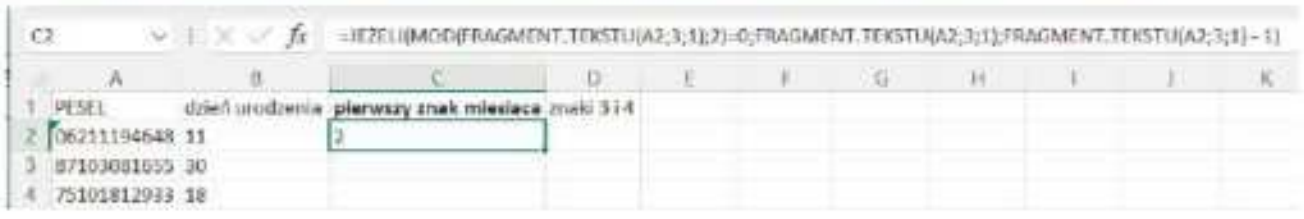
Uwaga!

Tekst domyślnie jest wyrównywany do lewej strony, a liczba – do prawej strony.

Przy wyznaczeniu miesiąca urodzenia należy zwrócić uwagę to, w którym stuleciu urodziła się osoba. Zauważ, że dla osób urodzonych w latach 1800–1899 do numeru miesiąca dodawane jest 80. Dla osób urodzonych w latach 2000–2099 do numeru miesiąca dodawane jest 20.

Aby to zrobić:

- wyznaczymy pierwszą cyfrę miesiąca;
- jeżeli wyznaczona liczba jest nieparzysta, odejmujemy od niej 1;
- `=JEŻELI(MOD(FRAGMENT.TEKSTU(A2;3;1);2)=0; FRAGMENT.TEKSTU(A2;3;1);FRAGMENT.TEKSTU(A2;3;1) - 1);`



	A	B	C	D	E	F	G	H	I	J	K
1	PESEL	dzień urodzenia	pierwszy znak miesiaca	znaki 3 i 4							
2	06211194648	11	2								
3	87103081655	30									
4	75101812933	18									

- potem wyznaczamy znaki trzeci i czwarty odpowiadające za miesiąc;

D2 =FRAGMENT.TEKSTU(A2;3;2)

	A	B	C	D	
1	PESEL	dzień urodzenia	pierwszy znak miesiaca	**znaki 3 i 4**	miesiąc
2	06211194648	11	2	21	
3	87103081655				
4	75101812933				

- jeżeli pierwszy znak numeru miesiąca jest większy od 1, należy od numeru miesiąca odjąć wartość równą „pierwszy znak miesiąca * 10”.

E2 =JEŻELI(C2>1;D2 - C2 * 10; D2 *1)

	A	B	C	D	E
1	PESEL	dzień urodzenia	pierwszy znak miesiaca	znaki 3 i 4	**miesiąc urodzenia** r
2	06211194648	11	2	21	1
3	87103081655				
4	75101812933				
5	88102763887				
6	65042036662				

Aby wyznaczyć rok urodzenia, najpierw wyznaczymy dwa pierwsze znaki z numeru PESEL, a potem dodamy do nich odpowiednie stulecie.

F2 =LEWY(A2;2)

	A	B	C	D	E	F
1	PESEL	dzień urodzenia	pierwszy znak miesiaca	znaki 3 i 4	miesiąc urodzenia	**znak 1 i 2**
2	06211194648	11	2	21	1	06
3	87103081655					
4	75101812933					
5	88102763887					

Następnie użyjemy funkcji `WYSZUKAJ()`, aby wyznaczyć odpowiednie stulecie. Skorzystamy ze spostrzeżenia, że za stulecie odpowiada pierwszy znak miesiąca. Zależność tę przedstawimy w tabeli:

	K	L	M
	pierwszy znak miesiąca	stulecie	
	0	1900	
	2	2000	
	4	2100	
	6	2200	
	8	1800	

Skorzystamy z funkcji `WYSZUKAJ()`, aby wyznaczyć odpowiednie stulecie. Będziemy szukać wskazanego pierwszego znaku miesiąca w $K$2:$K$7, a po znalezieniu wypiszemy odpowiednią wartość z $L$2:$L$7.

Do oznaczenia bloków użyjemy adresu bezwzględnego, ponieważ obszar ten nie będzie się zmieniał.

`=WYSZUKAJ(4;$K$2:$K$7;$L$2$:L$7)` znajdzie komórkę $K$4 i wypisze wartość z komórki $L$4.

=WYSZUKAJ(C2;$K$2:$K$7;$L$2:$L$7)

	B	C	D	E	F	G	
	dzień urodzenia	pierwszy znak miesiaca	znaki 3	miesią	znak 1 i 2	**stulecie**	rok ur
4648	11	2	21	1	06	2000	
31655							
12933							
53887							
36662							
11136							

Teraz wystarczy tylko dodać do wartości określającej rok odpowiednie stulecie.

Czcionka | Wyrównanie

=F2+G2

	C	D	E	F	G	H	
ia	pierwszy znak miesiaca	znaki 3	miesią	znak 1 i 2	stulecie	**rok urodzenia**	d
11	2	21	1	06	2000	2006	

Następnie skorzystamy z funkcji `DATA(rok; miesiąc; dzień)` i mamy wyznaczoną datę urodzenia. Pamiętaj, że najwcześniejsza data, którą obsługuje Excel, to 1 stycznia 1900 roku.

Należy teraz zaznaczyć bloki od B2 do I2 i przekopiować formuły do pozostałych wierszy.

	A	B	C	D	E	F	G	H	I
1	PESEL	dzień urodzenia	pierwszy znak miesiaca	znaki 3 i 4	miesiąc urodzenia	znak 1 i 2	stulecie	rok urodzenia	data urodzenia
2	06211194648	11	2	21	1	06	2000	2006	11.01.2006
3	87103081655								
4	75101812933								
5	88102763887								

	A	B	C	D	E	F	G	H	I
1	PESEL	dzień urodzenia	pierwszy znak miesiaca	znaki 3 i 4	miesiąc urodzenia	znak 1 i 2	stulecie	rok urodzenia	data urodzenia
2	06211194648	11	2	21	1	06	2000	2006	11.01.2006
3	87103081655	30	0	10	10	87	1900	1987	30.10.1987
4	75101812933	18	0	10	10	75	1900	1975	18.10.1975
5	88102763887	27	0	10	10	88	1900	1988	27.10.1988
6	65042036662	20	0	04	4	65	1900	1965	20.04.1965
7	64062811136	28	0	06	6	64	1900	1964	28.06.1964
8	64041293726	12	0	04	4	64	1900	1964	12.04.1964
9	84092457533	24	0	09	9	84	1900	1984	24.09.1984
10	05310191123	01	2	31	11	05	2000	2005	01.11.2005
11	57112988469	29	0	11	11	57	1900	1957	29.11.1957

Sprawdzenie, czy numer PESEL jest prawidłowy

Ostania cyfra w numerze PESEL to liczba kontrolna. Aby sprawdzić, czy numer PESEL jest właściwy, samodzielnie policzymy liczbę kontrolną i przekonamy się, czy jest ona równa ostatniej cyfrze w numerze PESEL.

Aby wyznaczyć liczbę kontrolną:

- mnożymy każdą z dziesięciu pierwszych cyfr numeru PESEL przez odpowiednie wagi:

  1 3 7 9 1 3 7 9 1 3.

- dodajemy otrzymane wyniki, np. dla numeru 06211194648:

  0 * 1 + 6 * 3 + 2 * 7 + 1 * 9 + 1 * 1 + 1 * 3 + 9 * 7 + 4 * 9 + 6 * 1 + 4 * 3

- otrzymujemy 182

Uwaga!

Pomijamy ostatnią cyfrę numeru PESEL.

- wyznaczamy resztę z dzielenia tej sumy przez 10.

Dla sumy 182 reszta wynosi 2.

Jeżeli reszta wynosi 0, to liczba kontrolna wynosi zero; w przeciwnym wypadku liczba kontrolna równa się różnicy 10 – wyznaczona reszta. Tu mamy 10 – 2, czyli 8.

Wyznaczona przez nas liczba kontrolna równa jest ostatniej cyfrze numeru PESEL, a więc numer PESEL jest prawidłowy.

Do wyliczenia liczby kontrolnej musimy wyznaczyć każdą cyfrę z numeru PESEL i przemnożyć ją przez odpowiednią wagę. Użyjemy do tego funkcji `FRAGMENT.TEKSTU()` i adresowania mieszanego.

Przygotujmy tabelę jak poniżej. W wierszu pierwszym wpiszemy wagi odpowiednich cyfr numeru PESEL, w wierszu drugim – numer cyfry w PESEL-u.

N11

	A	B	C	D	E	F	G	H	I	J	K	L
1	**wagi cyfry**	**1**	**3**	**7**	**9**	**1**	**3**	**7**	**9**	**1**	**3**	
2	PESEL/numer cyfry	1	2	3	4	5	6	7	8	9	10	11
3	06211194648											
4	87103081655											
5	75101812933											
6	88102763887											
7	65042036662											
8	64062811136											
9	84092457533											
10	05310191123											

Do wyznaczenia iloczynu cyfry i jej wagi użyjmy formuły:

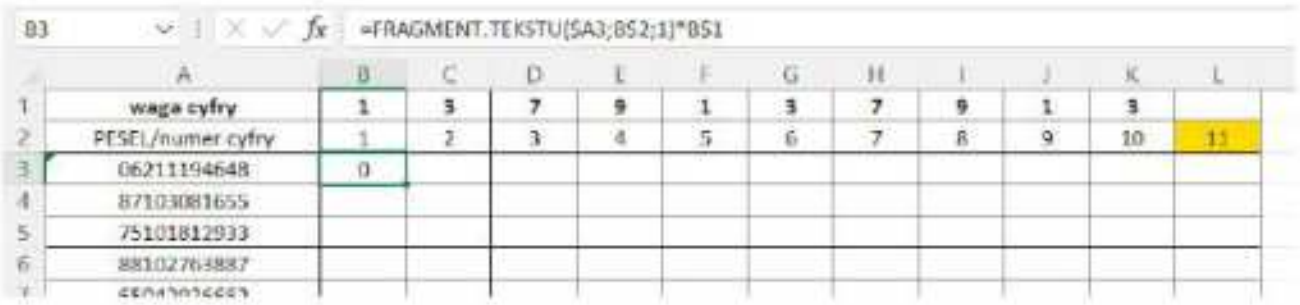
```
=FRAGMENT.TEKSTU($A3;B$2;1)*B$1
```

Formuła ta daje jeden znak na pozycji numer cyfry. Chcemy zmienić numer PESEL w obrębie kolumny A, dlatego użyliśmy adresu mieszanego $A3. Zakotwiczy nas to w kolumnie A, a dzięki temu pozwoli zmienić numer wiersza i przekopiować formułę na pozostałe numery PESEL. Podobnie chcemy zmienić numer cyfry w wierszu 2, dlatego zakotwiczymy się tylko w wierszu 2 i pozwolimy na zmiany kolumn.

```
=FRAGMENT.TEKSTU($A3;B$2;1)
```

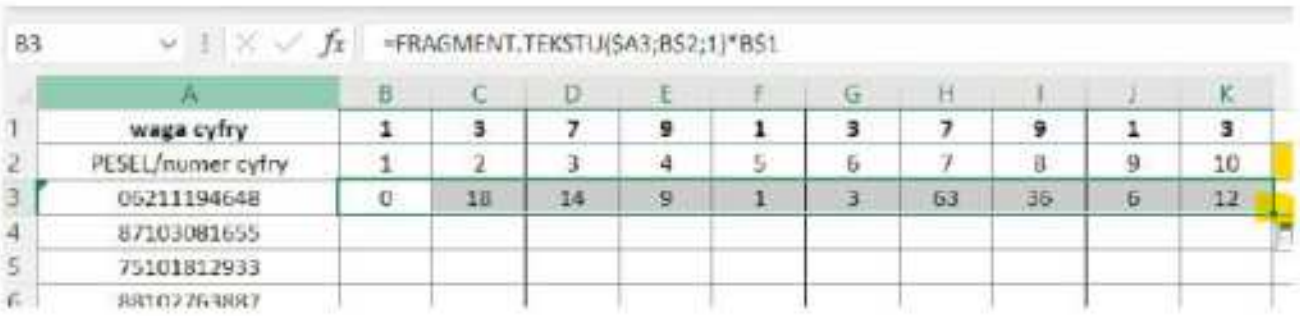
Analogicznie zrobimy dzięki pomnożeniu cyfry przez wagę. Chcemy w wierszu 1 zmieniać kolumny, dlatego zakotwiczymy się tylko w wierszu 1.

```
=FRAGMENT.TEKSTU($A3;B$2;1)*B$1
```



	A	B	C	D	E	F	G	H	I	J	K	L
1	waga cyfry	1	3	7	9	1	3	7	9	1	3	
2	PESEL/numer cyfry	1	2	3	4	5	6	7	8	9	10	11
3	06211194648	0										
4	87103081655											
5	75101812933											
6	88102763887											

Tak napisaną formułę przekopiujemy na pozostałe 9 cyfr numeru PESEL.



	A	B	C	D	E	F	G	H	I	J	K
1	waga cyfry	1	3	7	9	1	3	7	9	1	3
2	PESEL/numer cyfry	1	2	3	4	5	6	7	8	9	10
3	06211194648	0	18	14	9	1	3	63	36	6	12
4	87103081655										
5	75101812933										
6	88102763887										

Następnie przekopiujmy formułę na pozostałe wiersze z numerami PESEL. W tym celu klikniemy w prawym dolnym rogu zaznaczonego pliku B3:K3.

Teraz zostało już tylko wykonać pozostałe kroki do wyznaczenia liczby kontrolnej.

Uwaga!

Funkcja `MOD(liczba; dzielnik)` wyznacza resztę z dzielenia liczby przez dzielnik.

ZADANIA DO WYKONANIA

1. W pliku pesel.xlsx (który otrzymasz od nauczyciela) znajdują się numery PESEL. Dla każdego numeru podaj datę urodzenia i płeć wynikające z numeru PESEL.
2. Dla każdego numeru PESEL (z pliku z zad. 1.) oblicz liczbę kontrolną i sprawdź, czy jest zgodna z liczbą kontrolną w numerze PESEL.

PODSUMOWANIE LEKCJI

- Numer PESEL traktujemy jak tekst, żeby nie utracić zer na początku numeru. s. 98
- Do zamiany tekstu na wartość liczbową można skorzystać z formuły `=WARTOŚĆ(FRAGMENT.TEKSTU(A2;5;2))` lub przemnożyć nasz `=FRAGMENT.TEKSTU(A2;5;2)` razy 1. s. 98

19. Tabela przestawna, czyli wygodne grupowanie danych

NA TEJ LEKCJI:

- dowiesz się, jak korzystać z tabeli przestawnej.

Tabela przestawna krok po kroku

tabela przestawna

Tabela przestawna to narzędzie, którego używamy na danych przedstawionych w tabeli. Tabela przestawna pozwala na grupowanie danych i dokonywanie pewnych obliczeń i analiz w tych grupach.

W pliku pogotowie.xlsx (który otrzymasz od nauczyciela) znajdują się dane dotyczące usług medycznych Szpitalnego Oddziału Ratunkowego SOR z 2021 r. Dane te zawierają numer medyczny pacjenta, datę przyjęcia, kod lekarza, rodzaj pomocy, rodzaj ubezpieczenia oraz koszt usługi.

	A	B	C	D	E	F
1	Nr_medyczny	Data_przyjęcia	Kod Lekarza	Pomoc	Ubezpieczenie	Koszt leczenia
2	86M	12.02.2021	L3	Pomoc laryngologiczna	NFZ	230
3	1M	01.01.2021	L5	Intensywna terapia	Polisa zdrowotna	790
4	636K	28.08.2021	L6	Pomoc kardiologiczna	Polisa zdrowotna	420
5	212M	22.03.2021	L7	Pomoc kardiologiczna	NFZ	420
6	772K	17.10.2021	L6	Pomoc kardiologiczna	NFZ	420
7	482K	25.06.2021	L8	Pomoc laryngologiczna	NFZ	230
8	671M	08.09.2021	L2	Pomoc kardiologiczna	NFZ	420
9	102K	18.02.2021	L7	Intensywna terapia	NFZ	790
10	529M	12.07.2021	L8	Pomoc laryngologiczna	Abonament medyczny	230
11	757K	12.10.2021	L2	Pomoc okulistyczna	NFZ	310
12	10K	09.01.2021	L7	Pomoc kardiologiczna	NFZ	420

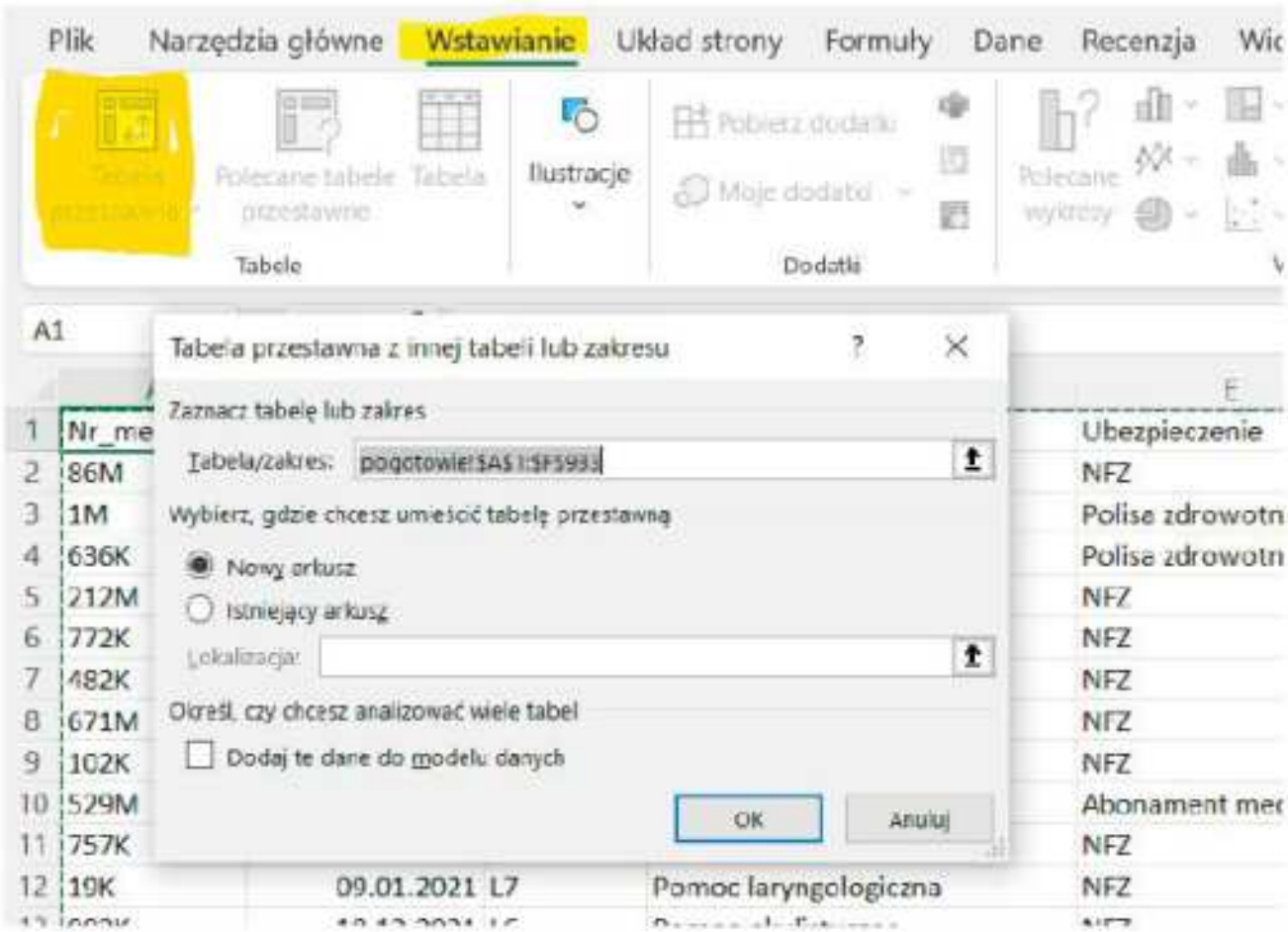
Dla każdego ubezpieczyciela chcemy policzyć, ile razy płacił za pomoc i jaki był sumaryczny koszt pomocy.

Do tych obliczeń wykorzystamy tabelę przestawną. Aby to zrobić:

1. Zaznaczmy dane, z których chcemy zrobić tabelę przestawną. Zamiast zaznaczać całą tabelę, możemy ustawić się w lewym górnym rogu naszej tabeli z danymi.

	A	B	C	D	E	F
1	Nr_medyczny	Data_przyjęcia	Kod Lekarza	Pomoc	Ubezpieczenie	Koszt leczenia
2	86M	12.02.2021	L3	Pomoc laryngologiczna	NFZ	230
3	1M	01.01.2021	L5	Intensywna terapia	Polisa zdrowotna	790
4	636K	28.08.2021	L6	Pomoc kardiologiczna	Polisa zdrowotna	420
5	212M	22.03.2021	L7	Pomoc kardiologiczna	NFZ	420
6	772K	17.10.2021	L6	Pomoc kardiologiczna	NFZ	420
7	482K	25.06.2021	L8	Pomoc laryngologiczna	NFZ	230
8	671M	08.09.2021	L2	Pomoc kardiologiczna	NFZ	420

2. Wybierzmy dane **Wstawianie/Tabela przestawna.**



Następnie wybierzmy, gdzie chcemy umieścić tabelę przestawną.

3. W obszarach tabeli przestawnej wstawmy w obszar **Wiersze** pole *Ubezpieczenie* – to jest pole, według którego grupujemy.

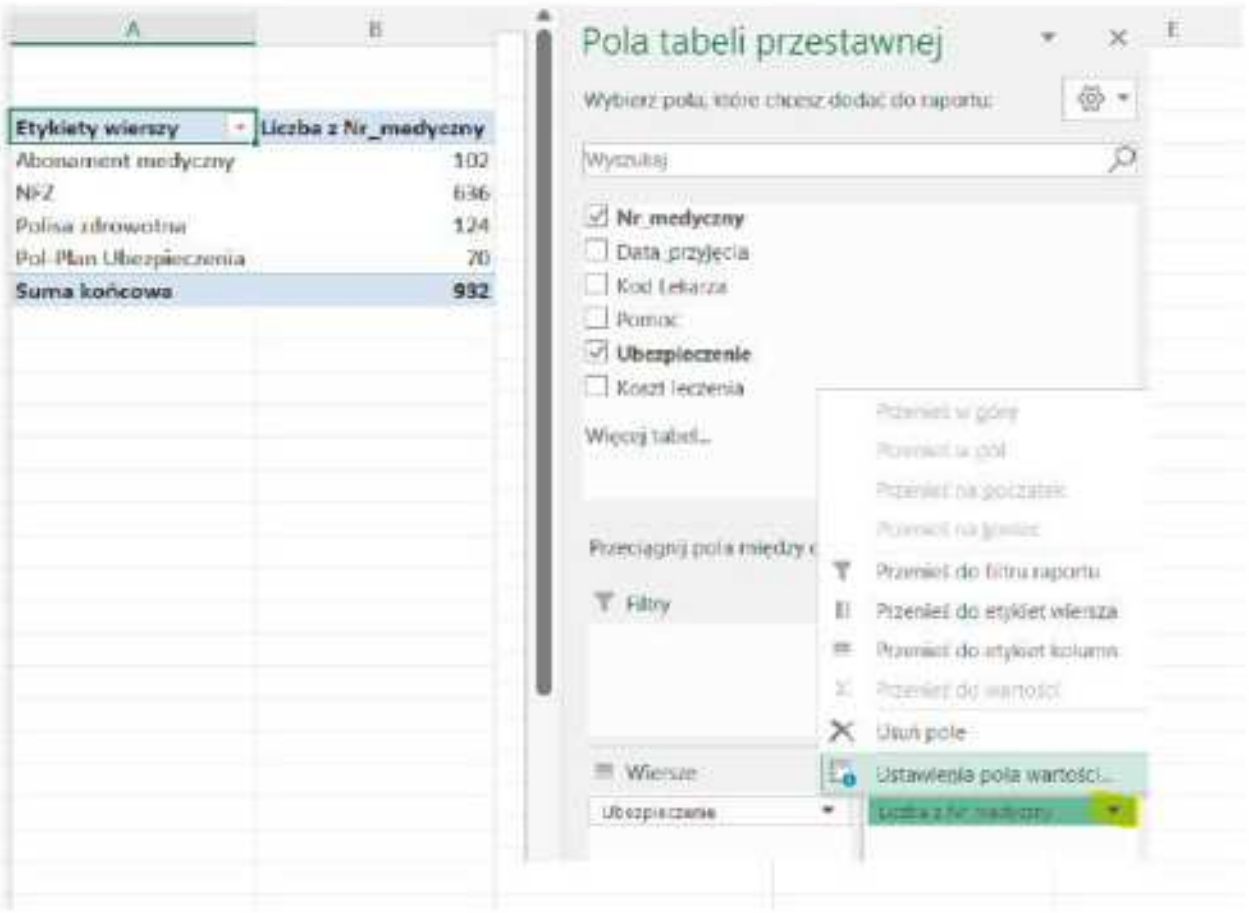
	A
1	
2	
3	**Etykiety wierszy**
4	Abonament medyczny
5	NFZ
6	Polisa zdrowotna
7	Pol-Plan Ubezpieczenia
8	**Suma końcowa**
9	
10	
11	

4. W obszar **Wartości** dodajemy *Nr_medyczny*, aby policzyć, ile wierszy znajduje się w każdej grupie.

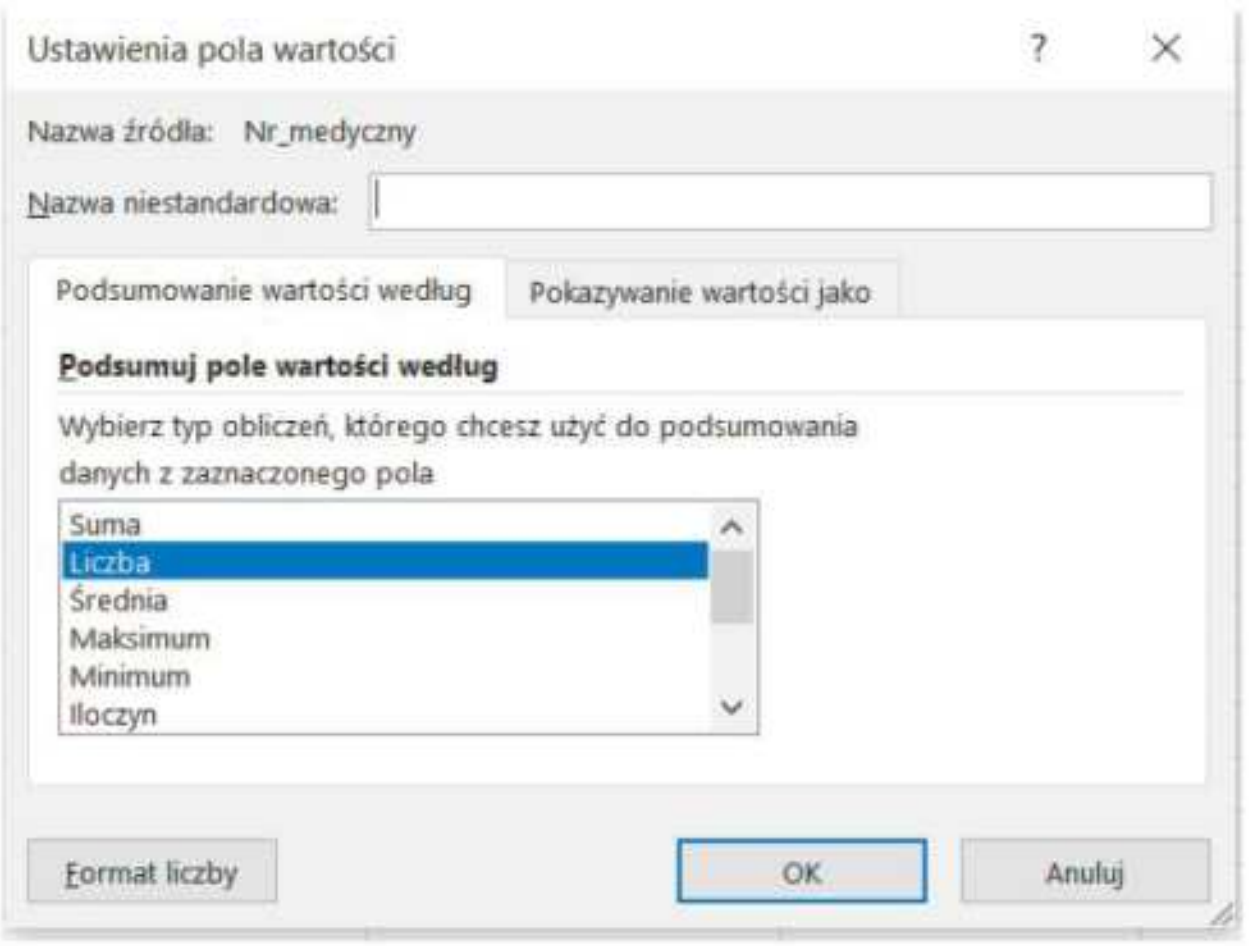
Uwaga!

Można to zrobić, przeciągając myszką wybrane pole w odpowiednie miejsce.

Do policzenia, za ile usług medycznych zapłacił każdy ubezpieczyciel, wybierzmy w polu *Nr_medycznego* **Ustawianie pola wartości**.



W nim zaznaczmy **Liczba** – pozwoli to nam policzyć, za ile usług zapłacił każdy ubezpieczyciel.

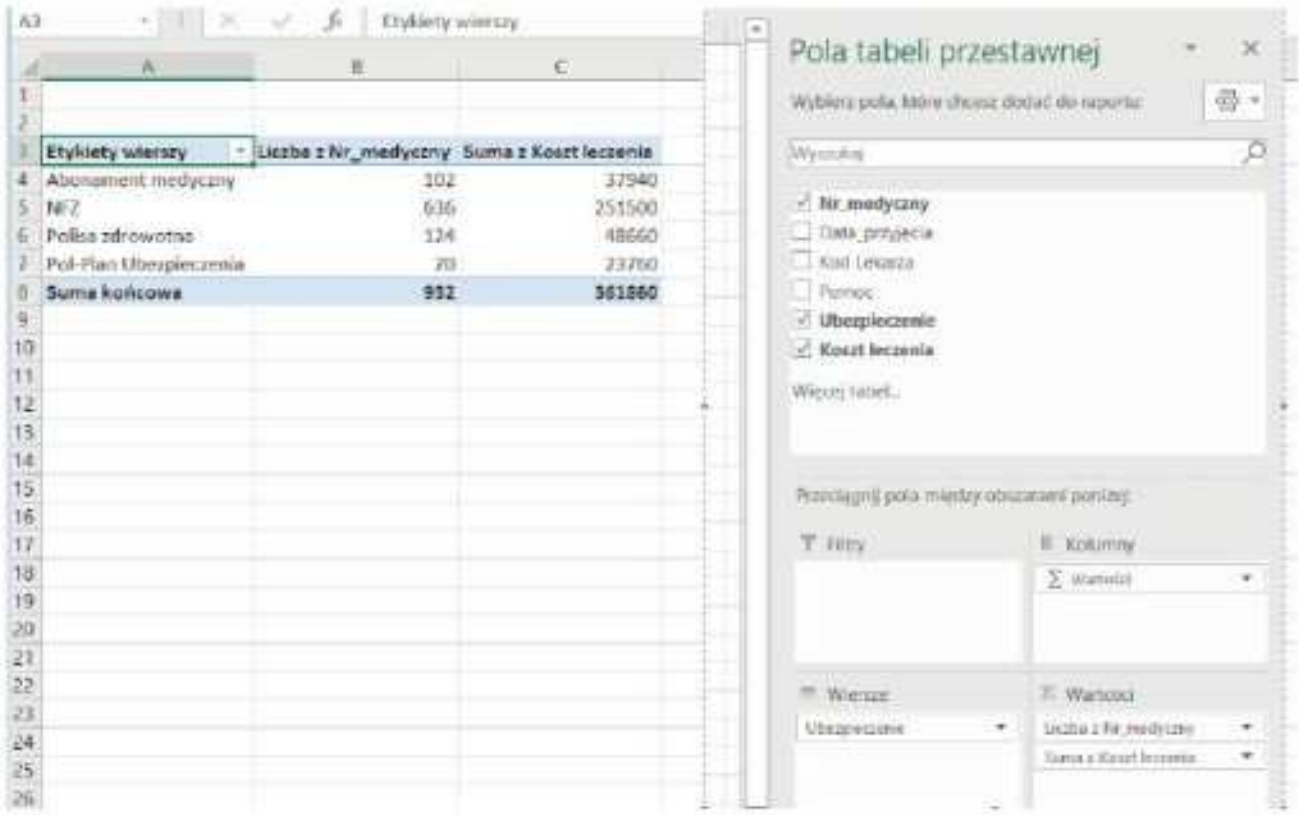


Wiemy już, za ile usług zapłacił każdy ubezpieczyciel medyczny. Teraz policzmy, jaki był sumaryczny koszt tych usług. Chcemy dla każdego ubezpieczyciela zsumować koszt każdej usługi, za którą zapłacił.

Aby to zrobić:

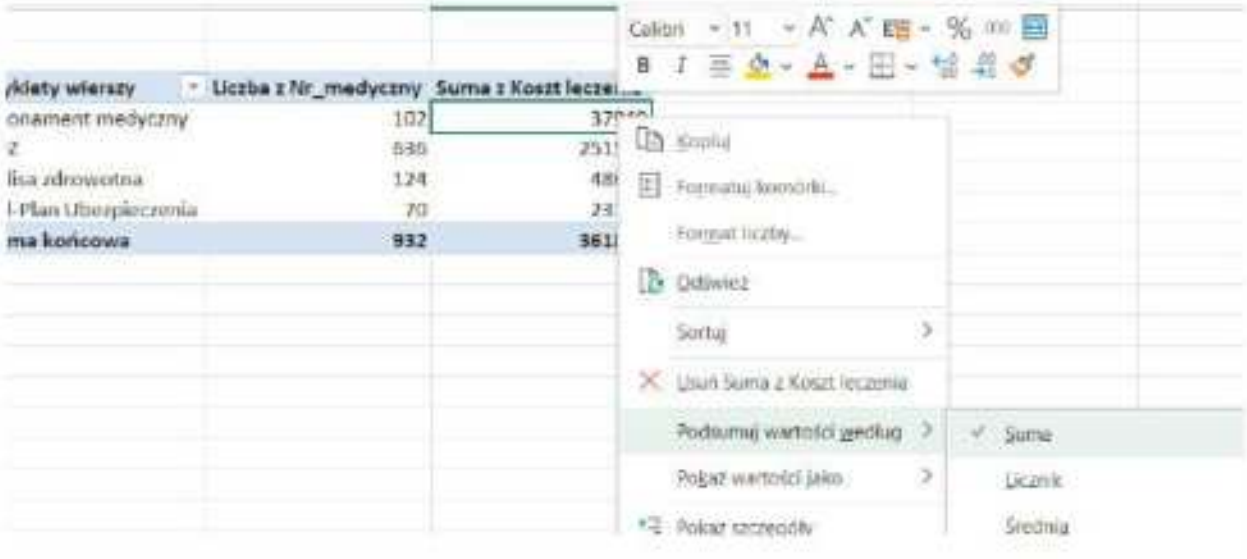
1. Dodajmy do obszaru **Wartości** pole *Koszt leczenia*.

2. W polu *Podsumuj wartości pola* wybierzmy **Suma**.



Uwaga!

Możemy wybrać **Podsumuj** wartości pola przez kliknięcie dwukrotne prawym klawiszem myszy – można mieć wybraną dowolną komórkę kolumny z *Koszt leczenia*. Zauważ, że możesz też wybrać opcję **Sortuj**, która pozwoli posortować dane w tabeli przestawnej zgodnie z podanym kryterium.



Rozszerzmy nasze zadanie:

Dla każdego ubezpieczyciela chcemy policzyć, ile razy płacił za każdy rodzaj pomocy oddzielnie i jaki był sumaryczny koszt każdego rodzaju pomocy.

Aby to zrobić, wystarczy w każdej grupie *Ubezpieczyciel* zrobić podgrupę *Pomoc*. Dodajmy do obszaru **Wiersze** pole *Pomoc*.

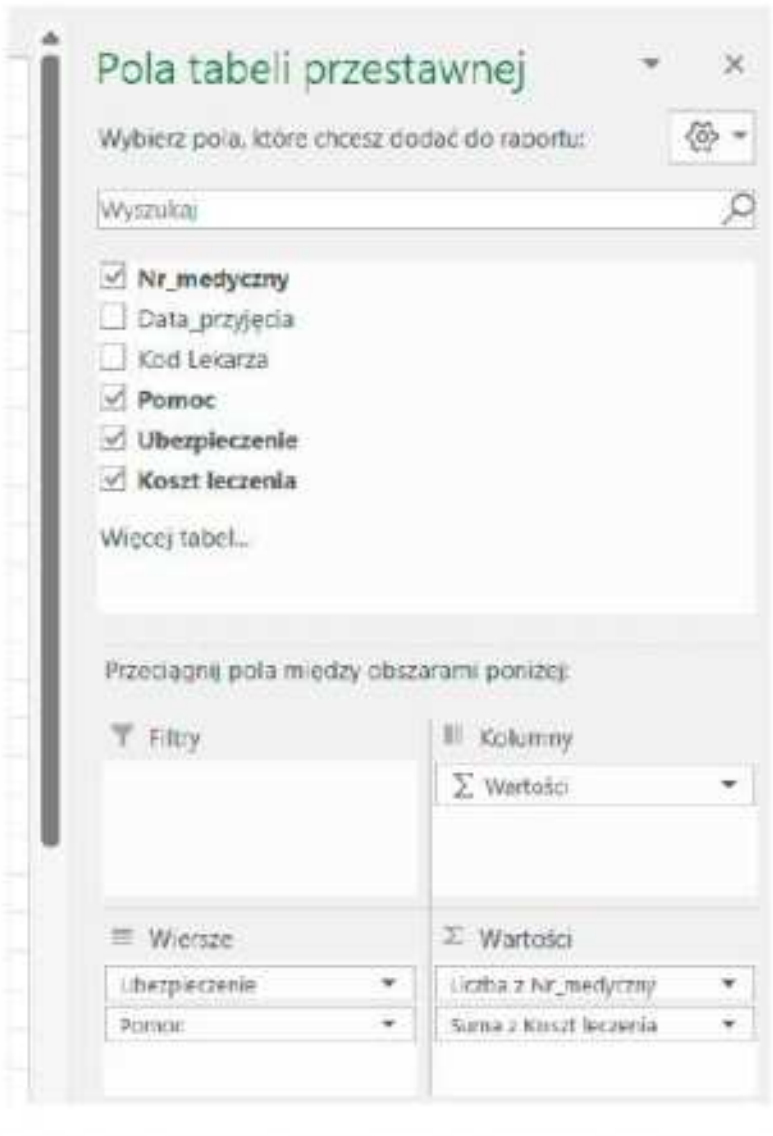
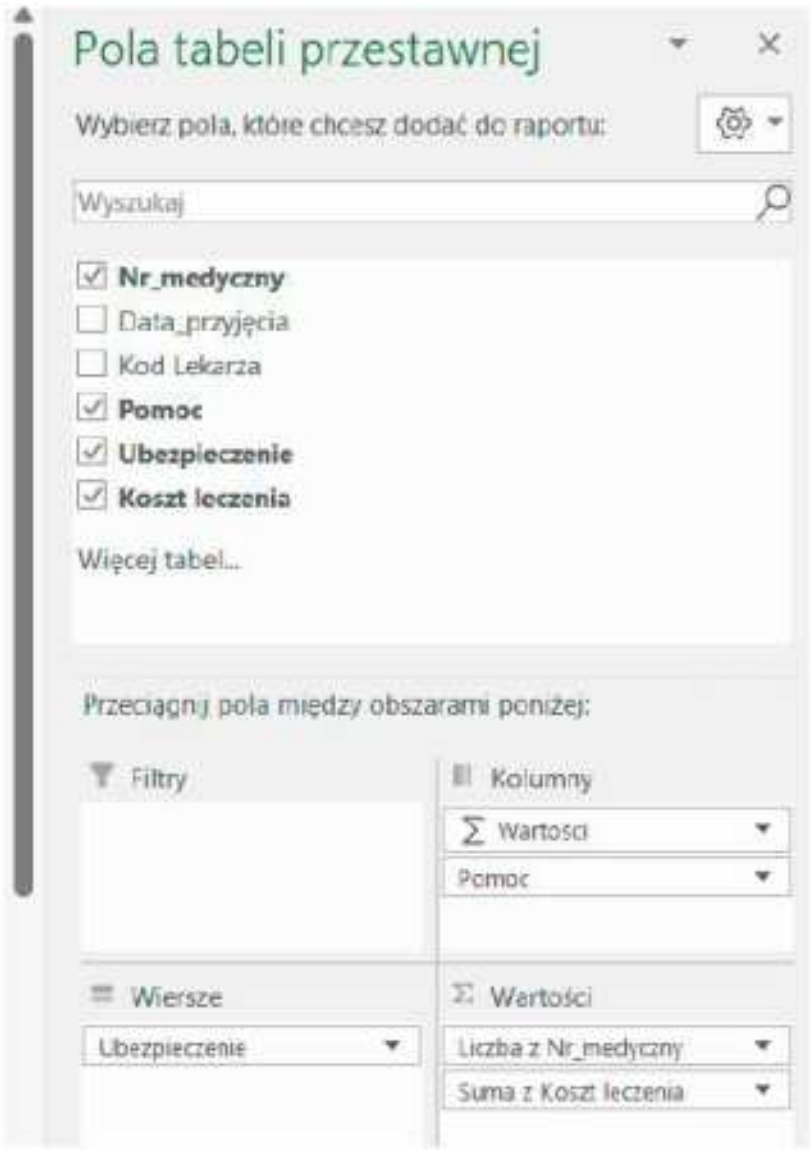


Tabela przestawna będzie wyglądała w następujący sposób:

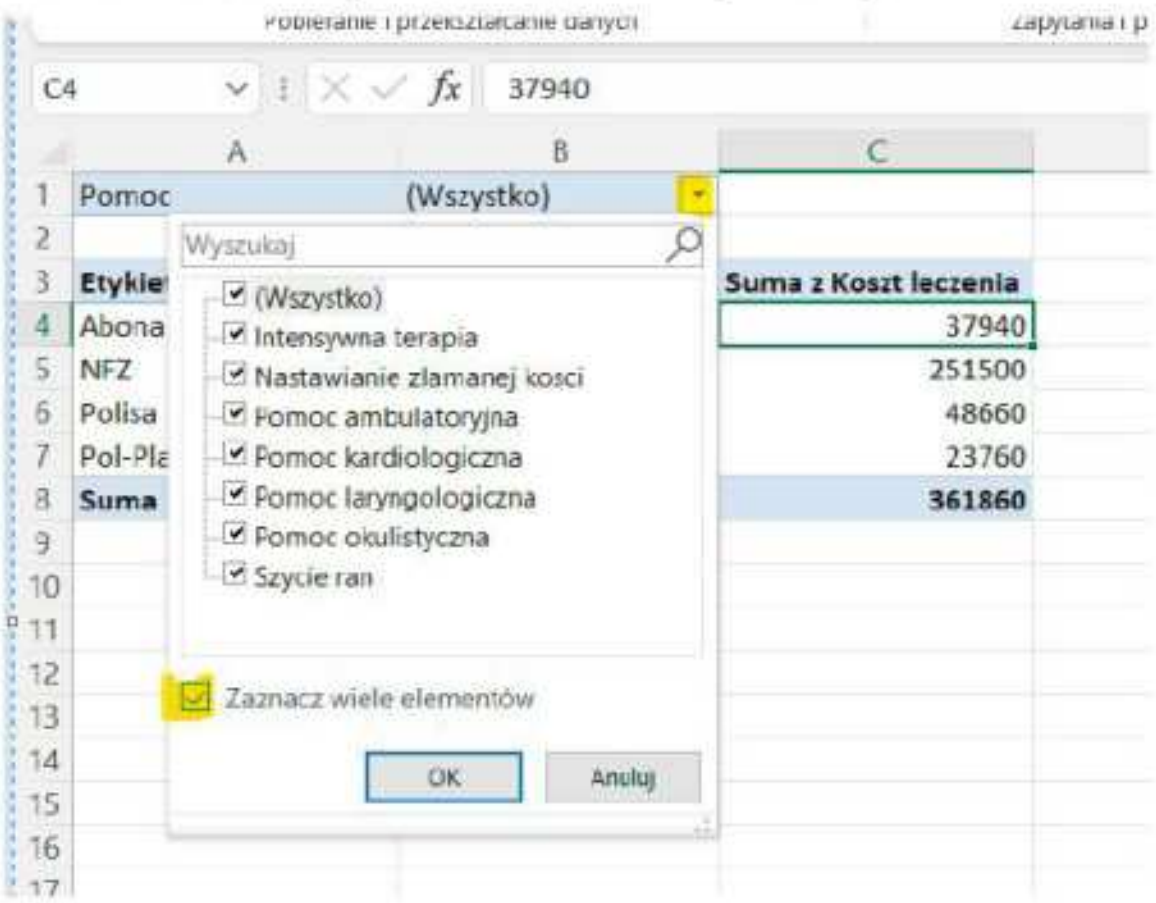
	A	B	C
1			
2			
3	**Etykiety wierszy**	**Liczba z Nr_medyczny**	**Suma z Koszt leczenia**
4	**Abonament medyczny**	**102**	**37940**
5	Intensywna terapia	9	7110
6	Pomoc ambulatoryjna	12	2280
7	Pomoc kardiologiczna	48	20160
8	Pomoc laryngologiczna	16	3680
9	Pomoc okulistyczna	13	4030
10	Szycie ran	4	680
11	**NFZ**	**636**	**251500**
12	Intensywna terapia	62	48980
13	Nastawianie złamanej kości	10	3700
14	Pomoc ambulatoryjna	52	9880
15	Pomoc kardiologiczna	342	143640
16	Pomoc laryngologiczna	75	17250

Zobaczmy, co się stanie, gdy przeniesiemy pole *Pomoc* do obszaru **Kolumny**.



Następnie przenieś pole *Pomoc* do obszaru **Filtr**.

Możemy teraz wyświetlać, ile razy każdy ubezpieczyciel zapłacił za wybraną pomoc i jaki był jej sumaryczny koszt. Wystarczy zaznaczyć: **Zaznacz wiele elementów**. Teraz wybierz, która pomoc cię interesuje.



ZADANIA DO WYKONANIA

1. Podaj kod pięciu lekarzy, którzy najczęściej udzielali pomocy pacjentom.
2. Utwórz zestawienie zawierające koszt pomocy w każdym miesiącu 2021 r., refundowany przez Narodowy Fundusz Zdrowia (NFZ).

PODSUMOWANIE LEKCJI

s. 104

- Tabela przestawna to jedno z zaawansowanych narzędzi arkusza kalkulacyjnego. Służy do grupowania i analizowania danych.

20. Wykresy, czyli jak atrakcyjnie przedstawiać dane

NA TEJ LEKCJI:

- dowiesz się, jakie są podstawowe typy wykresów i do czego ich używać;
- utworzysz wykres;
- zmienisz skale osi wykresu;
- dowiesz się, jak opisać wykres.

Typy wykresów

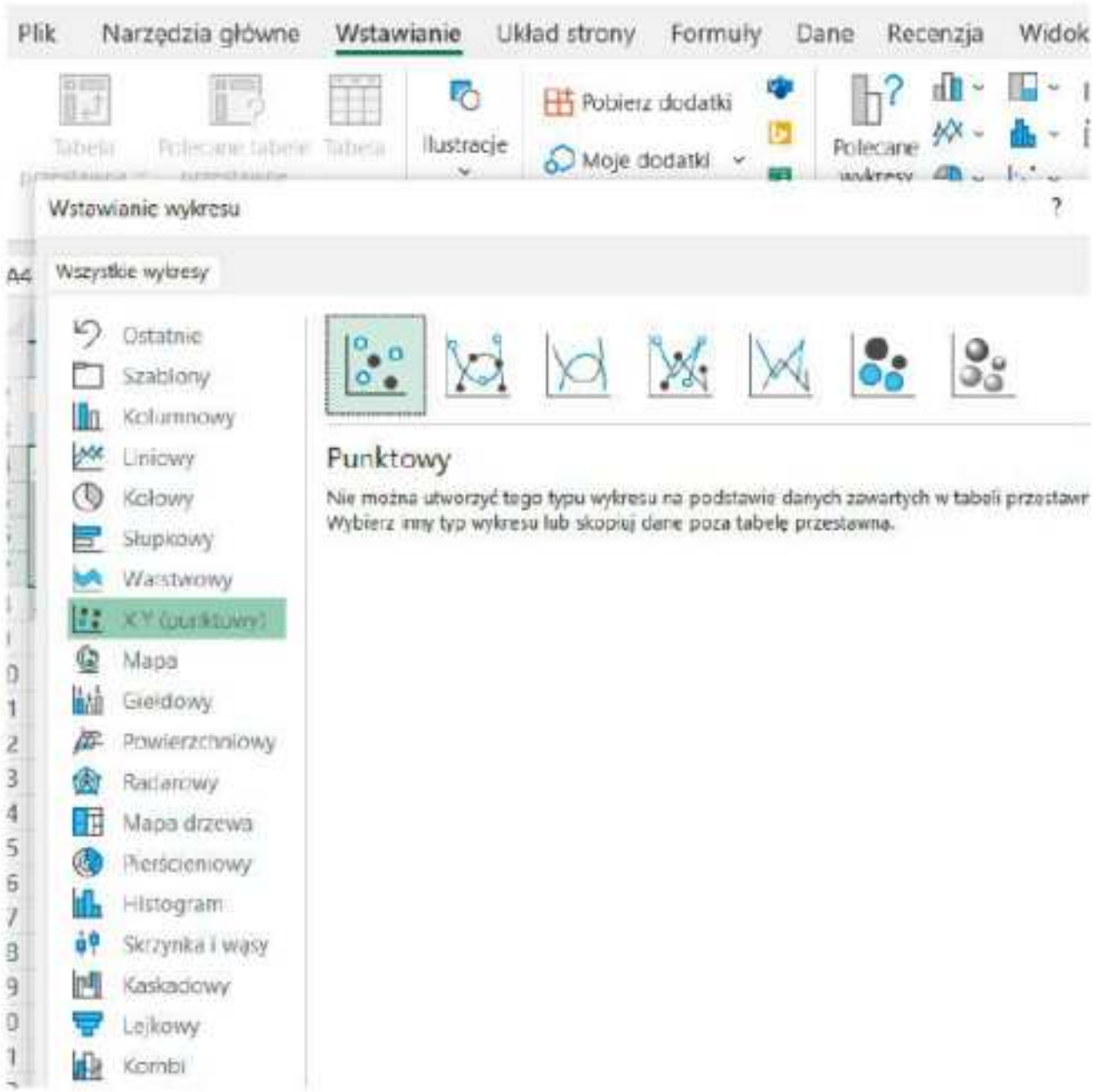
typy wykresów

Wykres pozwala nam w łatwy sposób zaprezentować dane zebrane w tabeli arkusza. Typ wybranego wykresu powinien być dostosowany do typu danych i treści, jaką ma przedstawiać.

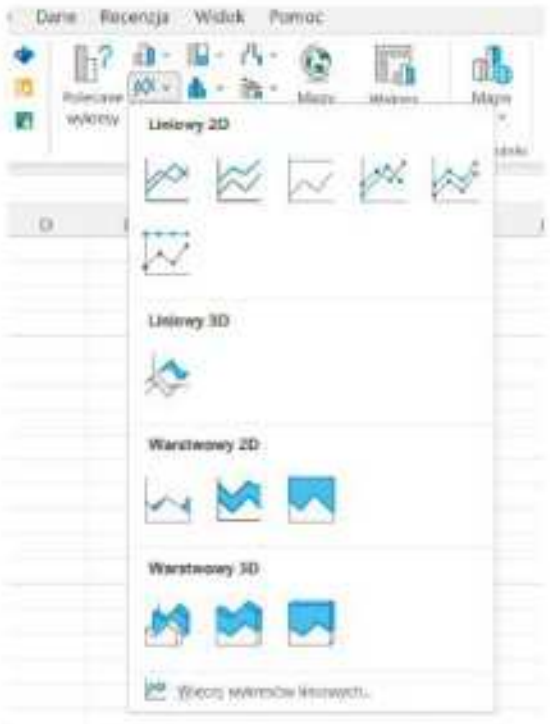
Dostępne w Excelu typy wykresów to:

- liniowy,
- punktowy,
- kołowy,
- słupkowy,
- kolumnowy,

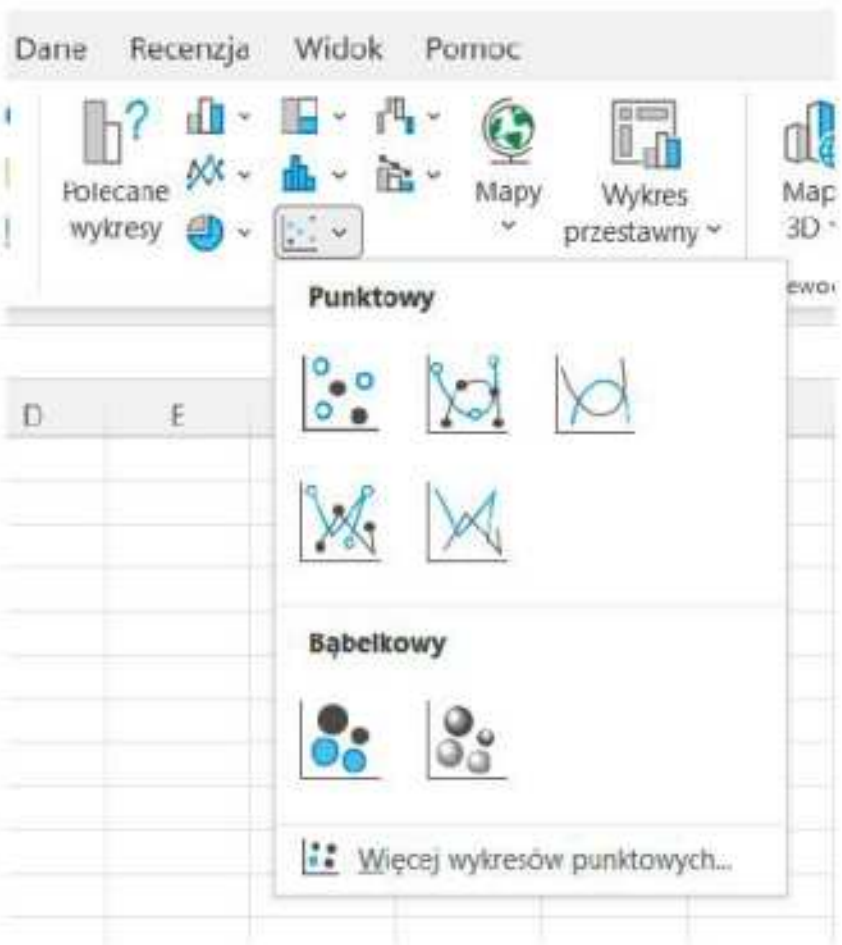
i inne.



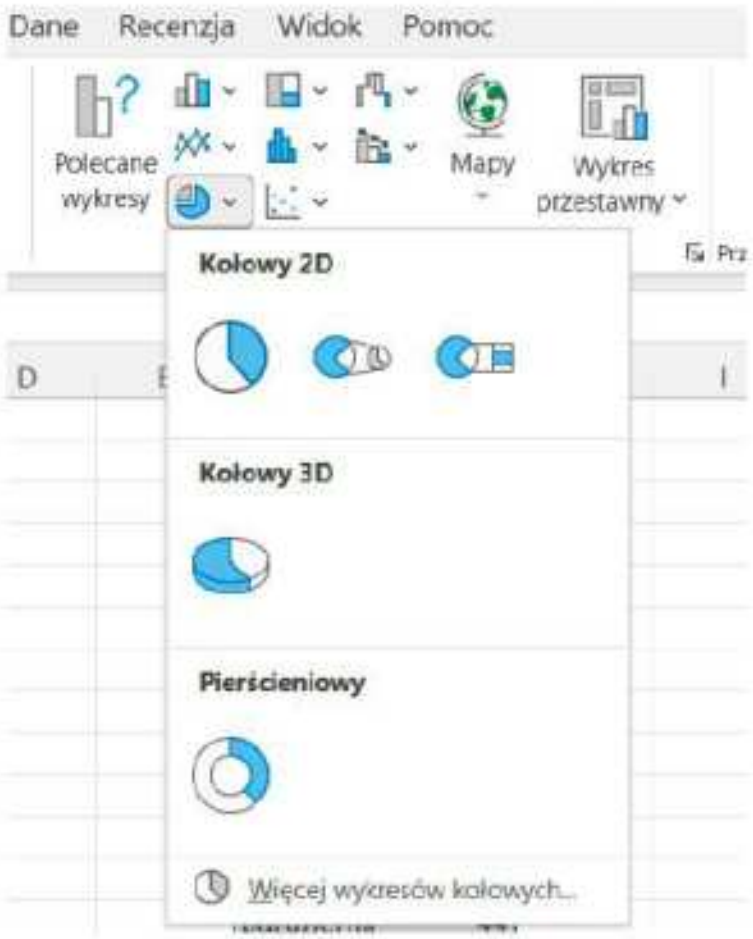
Wykres liniowy – prosty i często używany typ wykresu, na którym dane są prezentowane w postaci punktów połączonych liniami prostymi. Wykres liniowy często służy do wizualizowania zmian wartości w kolejnych przedziałach czasowych. Wykresy liniowe są przydatne do wyświetlania trendów w czasie i porównywania wielu serii danych.



Wykres punktowy – pozwala na wyświetlenie wartości dwóch zmiennych z zestawu danych przy użyciu współrzędnych kartezjańskich.



Wykres kołowy – pozwala na prezentowanie, jak ogólna całość dzieli się na różne części. Każda część koła jest konkretną kategorią w obrębie zestawu danych. Wykres kołowy wykorzystuje się do przedstawiania rozkładu procentowego. Na wykresie kołowym można pokazać tylko jedną serię danych.

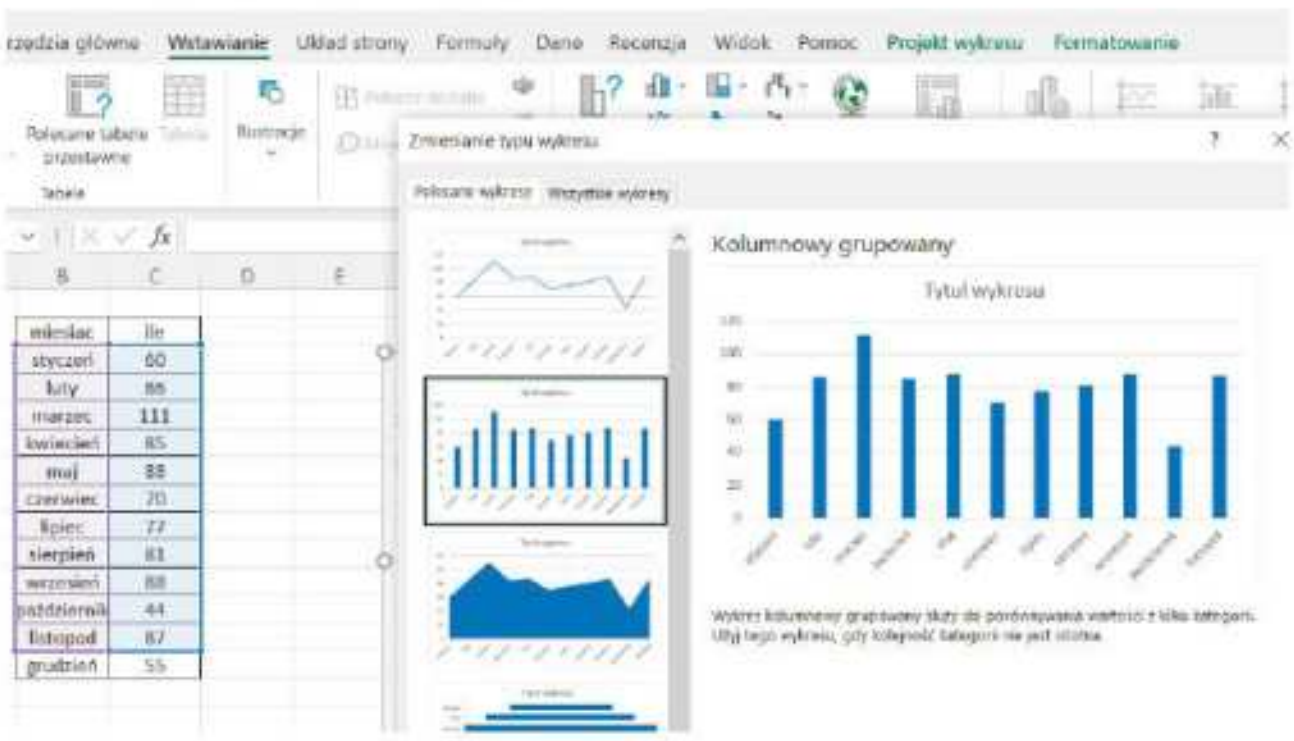


Wykres słupkowy/kolumnowy – służy do pokazania zależności między różnymi seriami niezależnych danych. Wykres przedstawia dane w postaci prostokątnych słupków o długości proporcjonalnej do wartości. W wykresie słupkowym słupki są rysowane poziomo, w wykresie kolumnowym – pionowo.

Tworzenie wykresu

Aby utworzyć wykres, musimy:

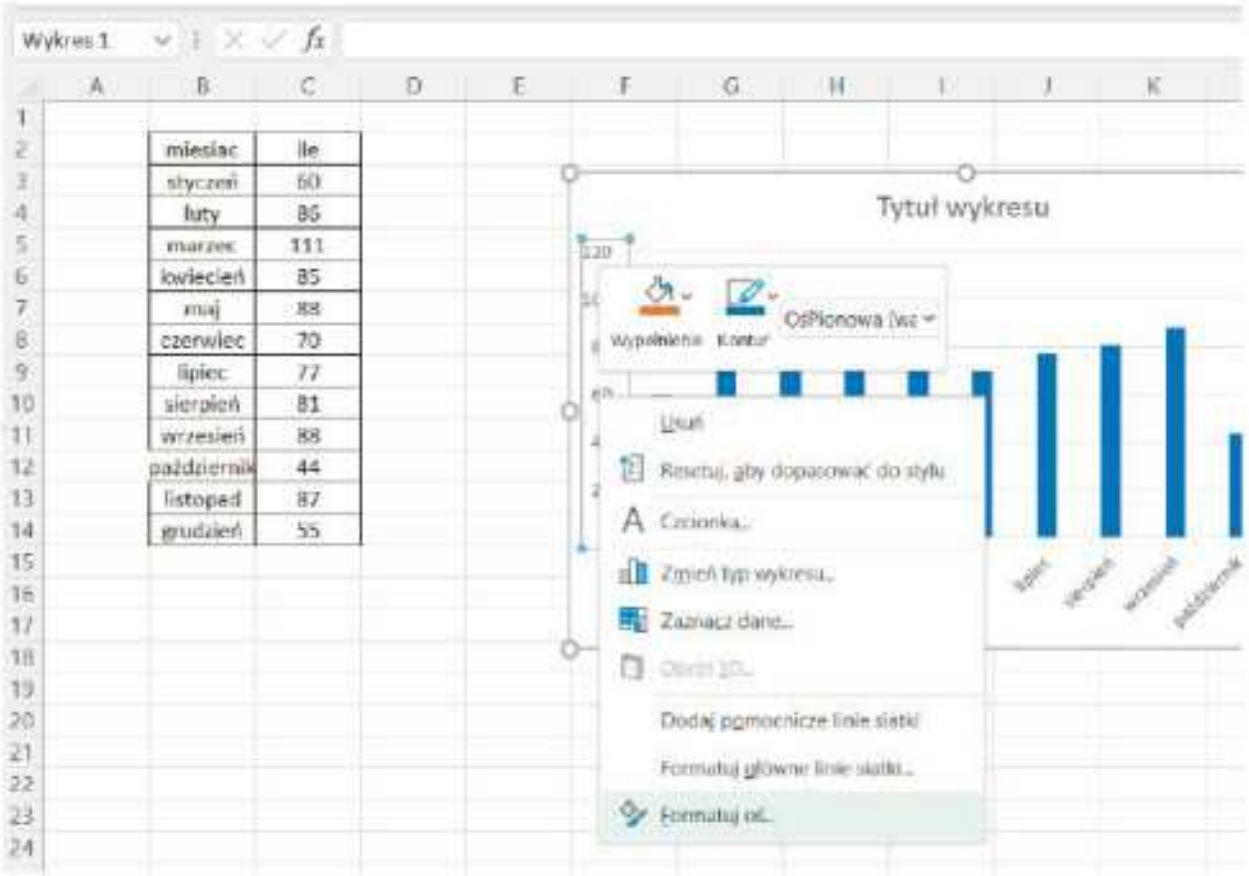
- zaznaczyć dane do wykresu,
- wybrać opcje Wstawianie/Polecane wykresy,
- zaznaczyć wybrany rodzaj wykresu.



Po wskazaniu typu wykresu – pod jego przykładem zobaczymy opis z informacjami, do jakiego rodzaju danych można stosować ten typ wykresu.

Jak zmienić skalę osi wykresu

Aby zmienić skalę na osi poziomej lub pionowej, wystarczy kliknąć prawym przyciskiem myszy wybraną oś i wybrać opcję **Formatuj oś**.



miesiąc	ile
styczeń	60
luty	86
marzec	111
kwiecień	85
maj	88
czerwiec	70
lipiec	77
sierpień	81
wrzesień	88
październik	44
listopad	87
grudzień	55

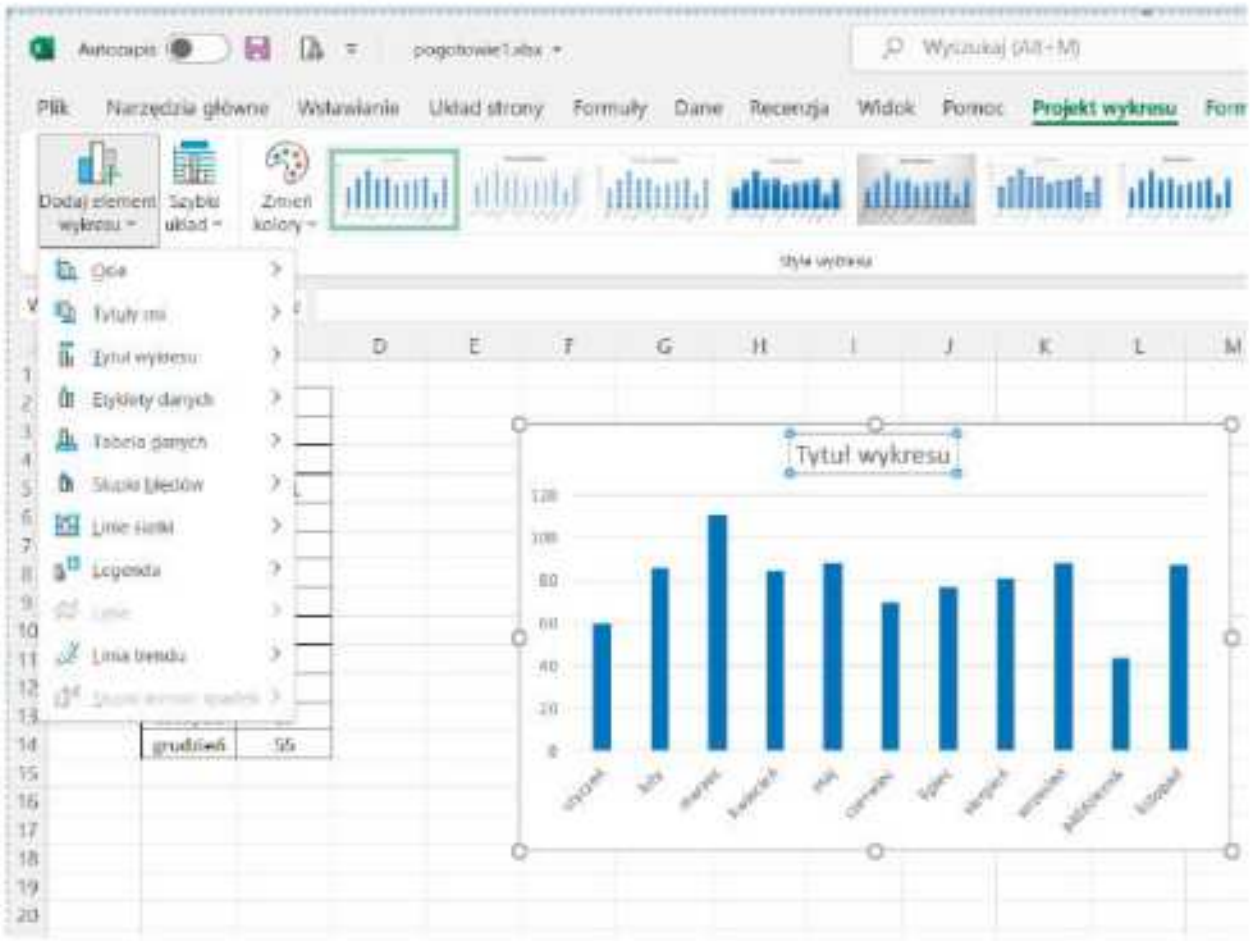
Następnie w **Opcjach osi** trzeba wybrać odpowiednie wartości.

Jak opisać wykres

Aby opisać wykres, należy dodać do wykresu:

- tytuł wykresu,
- etykiety danych,
- legendę.

Aby to zrobić, należy wybrać **Projekt wykresu/Dodaj element wykresu.**



ZADANIA DO WYKONANIA

1. Przedstaw na wykresie kolumnowym grupowym udział poszczególnych ubezpieczycieli w kosztach leczenia pacjentów w każdym miesiącu w 2022 r. Plik pogotowie.xlsx otrzymasz od nauczyciela.

PODSUMOWANIE LEKCJI

- Do reprezentowania danych często używamy wykresu. Najbardziej popularne wykresy to: liniowy, punktowy, kołowy i słupkowy/kolumnowy. s. 110

21. Pobieranie danych z pliku, czyli z notatnika do Excela

NA TEJ LEKCJI:

- dowiesz się, jak pobrać dane z pliku tekstowego.

Jak pobierać dane z pliku tekstowego?

Jeśli dane znajdują się w pliku tekstowym, należy je zaimportować do arkusza.

Import danych z pliku można wykonać na dwa sposoby:

- przez otwarcie danych z pliku tekstowego;
- przez zaimportowanie danych z pliku tekstowego.

import danych

W trakcie importu należy zwrócić uwagę:

- na to, jaki znak rozdziela dane w pliku;
- czy pierwszy wiersz zawiera opis kolumn.

Jeśli w pliku znajdują się dane typu data, należy przyjrzeć się, w jakim formacie są zapisane. Natomiast jeśli w pliku znajdują się dane zmiennoprzecinkowe, należy zobaczyć, co jest separatorem dziesiętnym.

W pliku aktywnosci.txt (który otrzymasz od nauczyciela) mamy informacje o aktywnościach sportowych pewnej grupy osób. Plik zawiera datę wykonania aktywności, godzinę jej rozpoczęcia, godzinę jej zakończenia, nazwę aktywności, numer PESEL osoby, która wykonała aktywność, i cenę za minutę tej aktywności. Dane są oddzielone znakiem tabulacji.

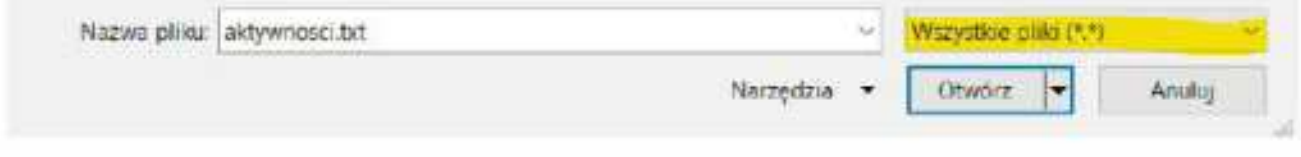
Zauważmy, że w tym pliku datę zapisano w formacie dd.mm.rrrrr. W pliku znajduje się numer PESEL, który będziemy importować jako tekst, żeby nie stracić zer na początku numeru. Separatorem dziesiętnym w liczbach zmiennoprzecinkowych jest kropka.

Importowanie przez otwarcie danych z pliku tekstowego

1. W skoroszycie wybieramy opcję **Plik/Otwórz/Przeglądaj** i wskazujemy plik, który chcemy otworzyć.

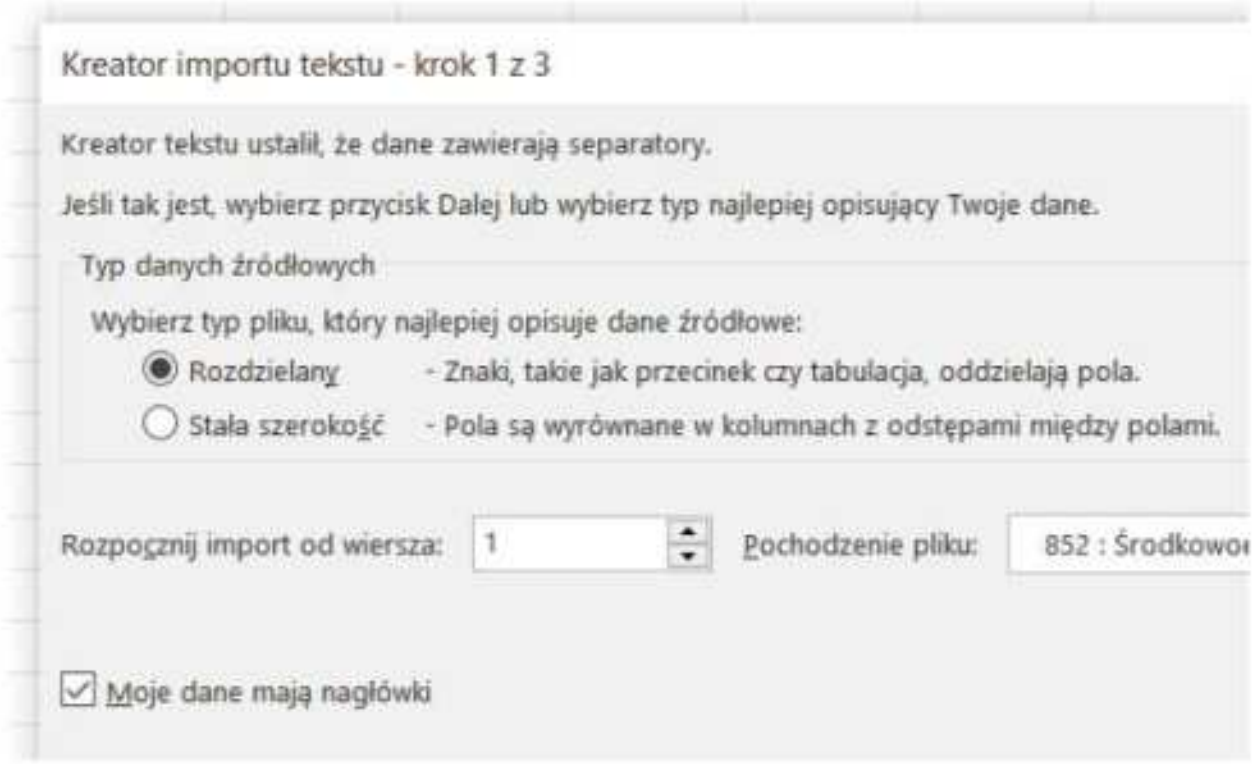
Uwaga!

Przy wybieraniu pliku należy zaznaczyć opcję **Wszystkie pliki**.

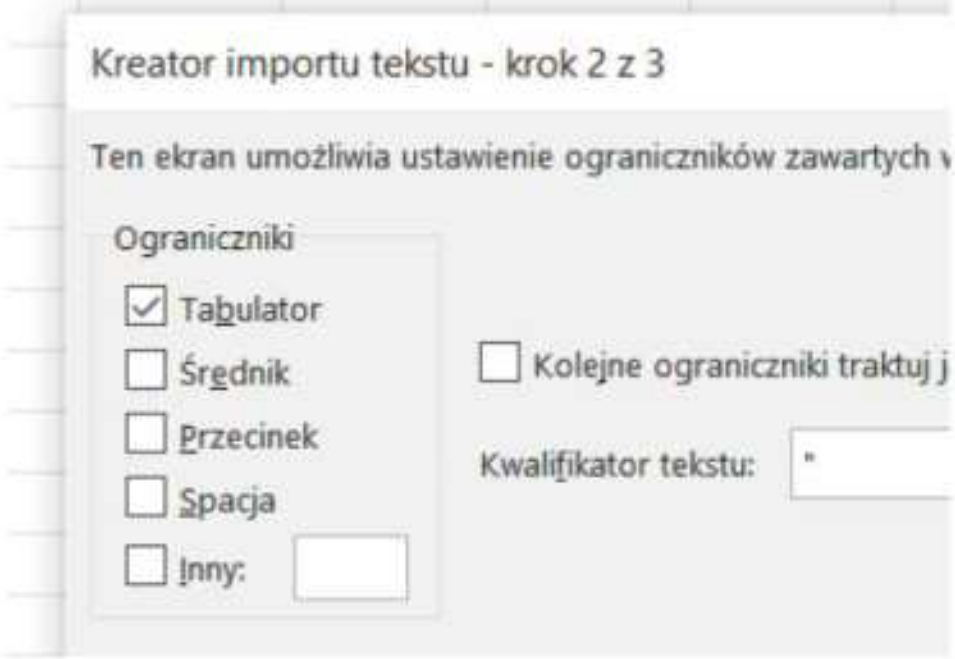


2. Po wybraniu pliku tekstowego pojawia się **Kreator importowania tekstu.**

- W pierwszym kroku zaznaczamy sposób rozdzielenia danych i podajemy informację, czy pierwszy wiersz zawiera nazwy kolumn.



- W drugim kroku podajemy, jaki znak oddziela kolumny.

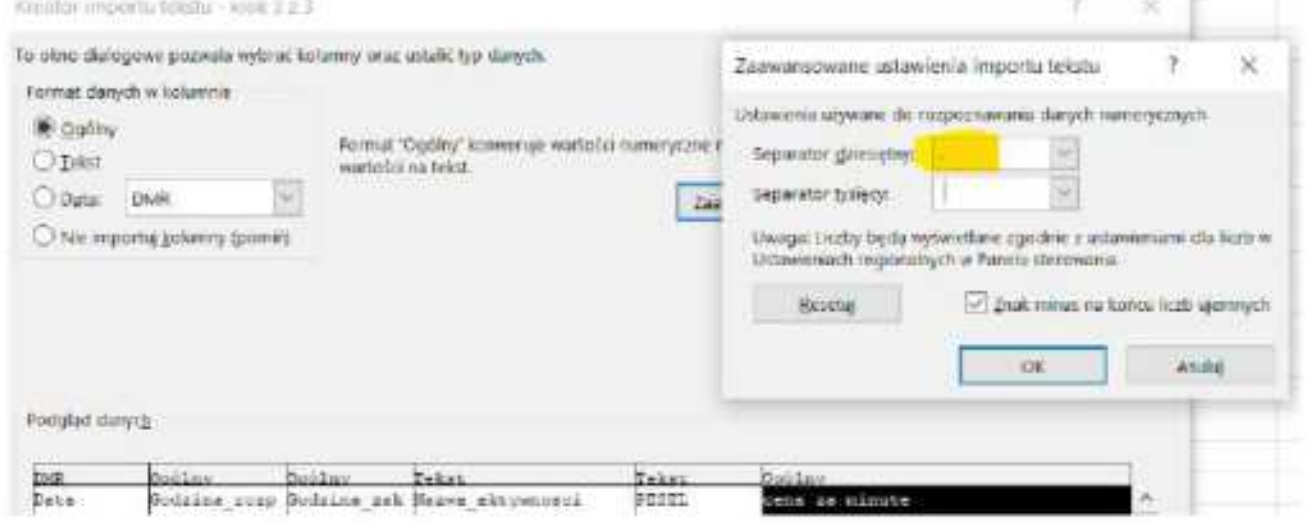


- W trzecim kroku dla każdej kolumny ustalamy typ danych. Domyślnie kreator wybiera typ **Ogólny**.

Dla danych typu **Data** zaznaczamy, w jakim formacie jest ona podana w pliku tekstowym.

Dla danych tekstowych zaznaczamy typ **Tekst**. Pamiętaj, że numer PESEL importujemy jako tekst.

Dla danych zmiennoprzecinkowych wybieramy, jaki znak jest separatorem dziesiętnym w pliku tekstowym.



Po wybraniu **Zakończ** dane zostaną zaimportowane do skoroszytu.

	A	B	C	D	E	F	G
1	Data	Godzina_rozp	Godzina_zak	Nazwa_aktywnosci	PESEL	cena za minute	
2	2018-07-01	17:00:00	17:45:00	zabawa	01301144688	5,44	
3	2018-07-01	20:00:00	21:15:00	zabawa	84061632305	5	
4	2018-07-01	16:00:00	16:45:00	modelowanie sylwetki	82092387166	5	
5	2018-07-01	20:00:00	21:00:00	dobra kondycja	92052540982	5,29	
6	2018-07-01	19:45:00	21:45:00	modelowanie sylwetki	88032884997	5,41	
7	2018-07-01	19:30:00	20:20:00	dobra kondycja	96021654296	5,93	
8	2018-07-01	19:00:00	20:30:00	modelowanie sylwetki	94110383957	5,03	
9	2018-07-01	20:00:00	21:15:00	modelowanie sylwetki	79052922062	5,12	
10	2018-07-01	19:00:00	20:30:00	zabawa	76021661410	5,73	
11	2018-07-01	18:45:00	19:30:00	zabawa	80091255745	5,95	

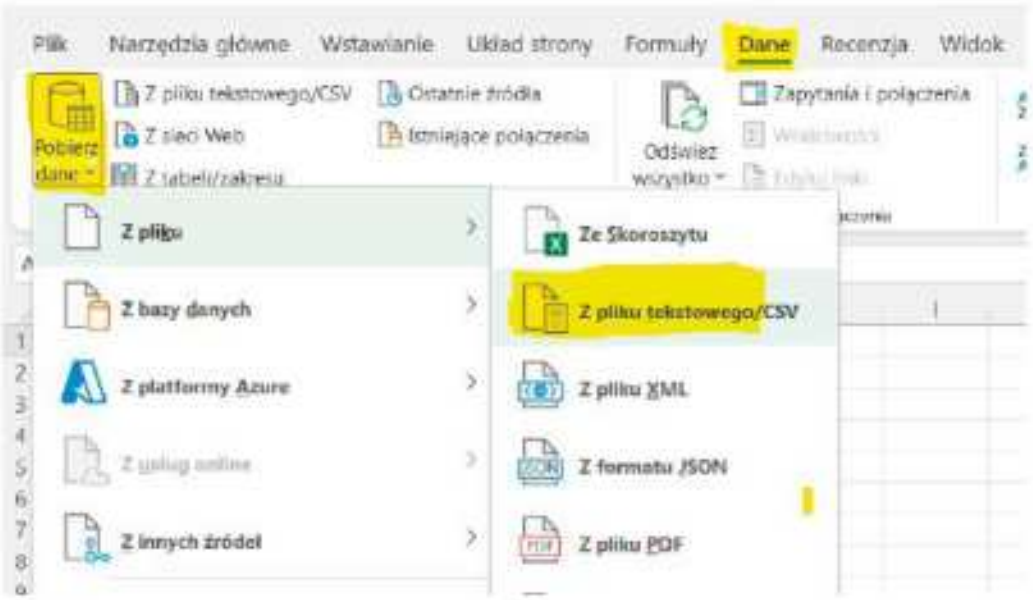
Uwaga!

Zauważ, że data w skoroszycie wyświetla się w formacie domyślnym dla tego skoroszytu (może być inny niż w pliku tekstowym). Dla liczb zmienno-przecinkowych także jest ustawiony domyślny separator dziesiętny.

3. Ostatnim punktem importowania danych przez otwarcie danych z pliku jest zapisanie danych jako skoroszyt programu Excel.

Pobieranie danych przez zaimportowanie z pliku tekstowego

Drugi sposób pobrania danych to wybranie opcji **Dane/Pobierz dane/Z pliku.**



ZADANIA DO WYKONANIA

1. Zaimportuj dane z pliku aktywnosci.txt (który otrzymasz od nauczyciela).

Użyj obu sposobów pobierania danych.

2. Zaimportuj do skoroszytu dane z pliku aktywnosci.txt (który otrzymasz od nauczyciela).
Na podstawie zdobytej do tej pory wiedzy wykonaj poniższe polecenia.

a) Dla każdej osoby wyznacz jej datę urodzenia oraz płeć.

b) Ustal, ile kobiet i ilu mężczyzn korzystało z aktywności.

c) Dla każdej aktywności podaj, ile osób z niej skorzystało.

d) Podaj łączną liczbę minut spędzonych przez mężczyzn na aktywności o nazwie relaks.

e) Dla każdego miesiąca podaj, ile pieniędzy użytkownicy wydali na aktywności. Dane przedstaw na wykresie słupkowym.

f) Podaj numery PESEL osób, które uprawiały więcej niż jedną aktywność tego samego dnia.

g) Podaj średni wiek kobiet i średni wiek mężczyzn, którzy korzystali z zajęć tanecznych (taniec). Wiek osoby oblicz, odejmując rok urodzenia od roku aktywności.

PODSUMOWANIE LEKCJI

s. 116

- Przy importowaniu danych przez otwarcie danych z pliku pamiętaj o zapisaniu pliku jako skoroszyt programu Excel. Przy pobieraniu danych z pliku zwróć uwagę na następujące kwestie:
  - jaki znak jest separatorem kolumn;
  - w jakim formacie zapisano daty;
  - co jest separatorem dziesiętnym w liczbach zmiennoprzecinkowych;
  - numer PESEL, który należy zaimportować jako tekst, żeby nie stracić zer na początku numeru.

PODSUMOWANIE PRZED SPRAWDZIANEM

13. Podstawowe typy danych, czyli co możemy wpisać w komórkę arkusza

- W komórkę możemy wpisać tekst, liczbę i formułę. s. 70
- Do komórki odwołujemy się przez adres względny, bezwzględny lub mieszany. s. 71

14. Trójkąt Sierpińskiego, czyli do czego przydaje się adresowanie względne i adresowanie bezwzględne oraz wypełnienie serią danych

- Do szybkiego wypełniania komórek podobnymi wartościami służy opcja **Narzędzia główne/Wypełnij/Seria danych**. s. 75
- Do generowania losowych wartości służą funkcje `LOS()`, `LOS.ZAKR()`. s. 80

15. Podstawowe funkcje tekstowe. Korzystanie z gotowych funkcji, czyli jak łatwo modyfikować dane tekstowe

- Podstawowe funkcje na tekście to: s. 83

Funkcja	Opis
DŁ	Zwraca liczbę znaków ciągu tekstowego.
LEWY	Zwraca określoną liczbę znaków z ciągu tekstowego z lewej strony.
PRAWY	Zwraca określoną liczbę znaków z ciągu tekstowego z prawej strony.
FRAGMENT.TEKSTU	Zwraca określoną liczbę znaków z ciągu tekstowego, zaczynając od zadanej pozycji.
ZNAJDŹ	Znajduje jedną wartość tekstową wewnątrz innej (z uwzględnieniem wielkich i małych liter).
ZŁĄCZ.TEKSTY	Łączy tekst z wielu zakresów i (lub) ciągów, ale nie zapewnia argumentów ignorowania pustych ani ogranicznika.
LITERY.WIELKIE	Konwertuje znaki tekstu na wielkie litery.
TEKST	Formatuje liczbę i konwertuje ją na tekst.

Wszystkie dostępne funkcje z określonych kategorii są widoczne w **Formuły/Wstaw Funkcję/Kategorie**.

16. Podstawowe funkcje dotyczące daty i czasu. Korzystanie z gotowych funkcji, czyli jak łatwo pracować z datą

s. 87

- Podstawowe funkcje na dacie i czasie to:

Funkcja	Opis
DATA	Zwraca liczbę kolejną dla wybranej daty.
ROK	Konwertuje liczbę kolejną na rok.
MIESIĄC	Konwertuje liczbę kolejną na miesiąc.
DZIEŃ	Konwertuje liczbę kolejną na dzień.
DNI.ROBOCZE	Zwraca liczbę pełnych dni roboczych między dwiema datami.
TERAZ	Zwraca liczbę kolejną bieżącej daty i godziny.
GODZINA	Konwertuje liczbę kolejną na godzinę.
MINUTA	Konwertuje liczbę kolejną na minutę.
SEKUNDA	Konwertuje liczbę kolejną na sekundę.

Wszystkie dostępne funkcje z określonej kategorii widoczne są w **Formuły/Wstaw Funkcję/Kategorie**. Formatowanie komórek pozwala nam przedstawić datę i czas w takiej postaci, jakiej potrzebujemy.

17. Jak szukać w Excelu, czyli gdzie to jest

s. 94

- Funkcje `WYSZUKAJ`, `WYSZUKAJ.PIONOWO`, `WYSZUKAJ.POZIOMO` umożliwiają szukanie wartości w kolumnie, wierszu lub tablicy.

s. 94

- W funkcji `WYSZUKAJ` obszar, w którym szukana jest wskazana wartość, musi być posortowany.

s. 97

- Jeżeli funkcja `WYSZUKAJ` nie może znaleźć wartości określonej przez [szukana wartość], to zwraca największą wartość w przeszukiwanym bloku [przeszukiwany wektor], która jest mniejsza od argumentu [szukana wartość] lub równa temu argumentowi. Jeśli szukana wartość jest mniejsza od najmniejszej, to funkcja `WYSZUKAJ` zwraca wartość błędu #N/D.

18. Co ukrywa numer PESEL, czyli do czego przydaje się adres mieszany

s. 98

- Numer PESEL traktujemy jak tekst, żeby nie utracić zer na początku numeru.

s. 98

- Do zamiany tekstu na wartość liczbową można skorzystać z formuły `=WARTOŚĆ(FRAGMENT.TEKSTU(A2;5;2))` lub przemnożyć nasz `=FRAGMENT.TEKSTU(A2;5;2)` razy 1.

19. Tabela przestawna, czyli wygodne grupowanie danych

- Tabela przestawna to jedno z zaawansowanych narzędzi arkusza kalkulacyjnego. Służy do grupowania i analizowania danych. s. 104

20. Wykresy, czyli jak atrakcyjnie przedstawiać dane

- Do reprezentowania danych często używamy wykresu. Najbardziej popularne wykresy to: liniowy, punktowy, kolowy i słupkowy/kolumnowy. s. 110

21. Pobieranie danych z pliku, czyli z notatnika do Excela

- Przy importowaniu danych przez otwarcie danych z pliku pamiętaj o zapisaniu pliku jako skoroszyt programu Excel. Przy pobieraniu danych z pliku zwróć uwagę na następujące kwestie: s. 116
  - jaki znak jest separatorem kolumn;
  - w jakim formacie zapisano daty;
  - co jest separatorem dziesiętnym w liczbach zmiennoprzecinkowych;
  - numer PESEL, który należy zaimportować jako tekst, żeby nie stracić zer na początku numeru.