Marek Luliński & Gniewomir Sarbicki

PYTHON C++ JAVASCRIPT

ZADANIA Z PROGRAMOWANIA

Helion

Opieka redakcyjna: Ewelina Burska

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pycjsz_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-4205-7

Spis treści

Wstęp

„**Programowanie komputerów** — proces projektowania, tworzenia, testowania i utrzymywania kodu źródłowego programów komputerowych lub urządzeń mikroprocesorowych. Kod źródłowy jest napisany w języku programowania, z użyciem określonych reguł. Może on być modyfikacją istniejącego programu lub czymś zupełnie nowym. Programowanie wymaga dużej wiedzy i doświadczenia w wielu różnych dziedzinach, jak projektowanie aplikacji, algorytmika, struktury danych, języki programowania i narzędzia programistyczne, kompilatory czy sposób działania podzespołów komputera. Między programistami trwają nieustanne debaty, czy programowanie komputerów jest sztuką, rzemiosłem, czy procesem inżynieryjnym. Bezpośrednią formą sztuki w tej dziedzinie jest demoscena".

— Wikipedia

Chcąc, aby komputer rozwiązał za nas zadanie albo chociaż pomógł w tej czynności, trzeba opracować algorytm, czyli przepis, który pokaże, jak w skończonej liczbie kroków i w skończonym czasie można osiągnąć poprawny wynik. Przekłada się to na wykonanie szczegółowej listy rozkazów zapisanej w języku symbolicznym tak, aby komputer mógł je zrealizować. Jeżeli program działa poprawnie dla różnych danych, to przekonanie, że działać będzie dla dowolnych danych, może być uzasadnione. Ważny jest też właściwy wybór języka programowania do konkretnego zadania. Stąd taka ich mnogość. Każdy język ma swoje specyficzne zastosowania i powstaje często tylko dla określonych celów. Nie udało się jeszcze stworzyć języka uniwersalnego, chociaż są takie, które do tego pretendują i mają wszechstronne zastosowania. Przedstawienie algorytmu w języku symbolicznym pozwala abstrahować od cech komputera i skupić się wyłącznie na rozwiązaniu zadania.

W książce tej każde zadanie będzie rozwiązane przy użyciu czterech języków: Pascal, C++, JavaScript, Python.

Taki wybór języków w tym opracowaniu nie jest przypadkowy. Pascal i C++ są obecnie najpopularniejszymi językami stosowanymi w edukacji informatycznej w szkołach, a język Python będzie niebawem wprowadzany do nauczania i dostępny dla ucznia na egzaminie maturalnym. Znaczenie przyporządkowane symbolom tych języków (semantyka) oraz składnia (syntaktyka), w szczególności języka Python, umożliwiają implementowanie algorytmów w sposób naturalny i przejrzysty. Język Pascal, mimo że uważany dziś za archaiczny, ma zastosowanie w programowaniu w środowiskach Delphi i Lazarus oraz jest umiarkowanie trudny.

Należy przy tym pamiętać, że komputer i jego oprogramowanie powinny być jedynie narzędziem, środkiem pomocniczym umożliwiającym rozwiązywanie postawionego problemu, a każde zadanie może stanowić drogę do rozwiązywania go według własnego pomysłu i w wybranym języku.

Przedstawione zadania będą miały charakter matematyczny i numeryczny, ale nie tylko — np. dział o przetwarzaniu tekstów. Trudniejsze kwestie matematyczne zostaną omówione. Programy pokazujące, jak komputer może rozwiązać postawione zadania, będą miały charakter strukturalny, często z wykorzystaniem rekurencji, i będą tworzone wyłącznie dla trybu tekstowego. Nie przewiduje się w niniejszym opracowaniu przykładów programowania w trybie graficznym, chociaż zadania z tym związane mogą być bardzo interesujące.

Książka przeznaczona jest dla ucznia i nauczyciela. Pomyślana została jako pomoc dydaktyczna dla nauczyciela — stając się być może inspiracją do modyfikowania i tworzenia własnych zadań — ale przede wszystkim dla ucznia, dając mu liczne przykłady zadań, jakie można rozwiązać za pomocą komputera. Niniejsze opracowanie powinno być traktowane przez nauczyciela jako pomoc w nauczaniu o algorytmach. Propozycja i dobra rada dla ucznia: zanim skorzystasz z gotowego rozwiązania i kodu, najpierw spróbuj zbudować własny program w wybranym przez siebie języku. Samodzielne rozwiązanie zadania jest potwierdzeniem zdobycia umiejętności algorytmicznego myślenia. Można więc traktować tę książkę nie jako zbiór zadań, ale jako zbiór programów do napisania.

Życzymy powodzenia w kreatywnym wykorzystywaniu książki.

Autorzy
Marek Luliński (Pascal, C++, JavaScript)
Gniewomir Sarbicki (Python)

Wprowadzenie

Zasady rozwiązywania zadań

W książce podawane są fragmenty kodów, przede wszystkim w postaci funkcji do wykorzystania we własnych programach.

W związku z tym należy wykonać następujące czynności:

1. Przygotować szablon kodu całego programu dla wybranego języka bądź dokumentu HTML.
2. W programie głównym wprowadzić dane z klawiatury, a w przypadku JavaScriptu — z pól tekstowych, deklarując odpowiednie typy danych.
3. Wykorzystać podany kod rozwiązania, a jeśli jest on w postaci funkcji, to wykonać odpowiednie jej wywołanie z parametrami aktualnymi.

Dane i wyniki

Zakładamy poprawność wszystkich wprowadzanych danych. Nie przewiduje się ich sprawdzania w podanych w książce rozwiązaniach. Błędne dane skutkować będą błędnym działaniem programu lub jego zawieszeniem się.

Dane będą często wprowadzane z klawiatury, a wyniki wyświetlane na ekranie.

Wszędzie, gdzie w rozwiązaniach zadań wystąpi wydruk wyników na ekranie, można go zastąpić zapisem do pliku tekstowego. Nie jest to jednak konieczne w żadnym z zadań w tej książce.

Wyniki w JavaScripcie będą wyprowadzane metodą `document.getElementById("id").innerHTML`, wówczas trzeba przygotować odpowiedni pojemnik (pojemniki) `<div>`, albo poleceniem `console.log()`, wówczas po otworzeniu dokumentu HTML należy otworzyć też konsolę przeglądarki. Wyniki oraz teksty komunikatów czytelnik może formatować sam według własnych potrzeb. Warto przejrzeć podane przykłady programów dla danego języka.

Pascal

Język programowania **Pascal** — nazwany tak na cześć francuskiego matematyka i filozofa Blaise Pascala — zaprojektował szwajcarski informatyk Niklaus Wirth. Wirth zaczął tworzyć język w roku 1968, ale pierwsza implementacja Pascala ukazała się w roku 1970. Wiele źródeł podaje jako rok jego powstania 1971. Jest to język wysokiego poziomu,

ale umożliwiający także dostęp do asemblera. W roku 1983 pojawił się Turbo Pascal firmy Borland, który cieszył się ogromną popularnością ze względu na prostotę obsługi, niezawodność oraz szybką kompilację i szybkie uruchomienie. Pascal jest językiem wewnętrznym znakomitych środowisk IDE — Delphi i Lazarus, umożliwiających szybkie tworzenie atrakcyjnych, wizualnie rozbudowanych aplikacji. Pascal stosowany był i nadal jest używany głównie do celów edukacyjnych, szczególnie do nauki programowania strukturalnego. Pascal działa na różnych platformach, jak Windows, macOS i Linux.

Struktura programu źródłowego

Plik źródłowy programu jest plikiem tekstowym z instrukcjami języka.

Plik z nadanym rozszerzeniem *pas* zawiera program napisany w języku Pascal.

W wyniku kompilacji i konsolidacji powstaje plik wykonywalny.

`program ......;` — nazwa programu
`uses .........;` — deklaracje modułów standardowych i modułów własnych
`type .........;` — deklaracje typów
`var ..........;` — deklaracje zmiennych
`begin` — początek programu
`..............;` — instrukcje
`end.` — koniec programu

Przykład programu

```
program Euklides;                    { nazwa programu }
uses Crt;                            { użycie modułu obsługi monitora i klawiatury }
var a, b, m, n: integer;             { deklaracja zmiennych typu całkowitego }
begin                                { początek programu }
   ClrScr;                           { „Clear Screen” — „czyszczenie” ekranu }
   write('   I liczba = ');          { wyświetlenie tekstu  na ekranie }
   readln(a);                        { wczytanie I liczby z klawiatury do zmiennej }
   m:=a;                             { podstawienie }
   write('  II liczba = ');          { wyświetlenie tekstu na ekranie }
   readln(b);                        { wczytanie II liczby z klawiatury do zmiennej }
   n := b;                           { podstawienie }
   while a <> b do                   { powtarzanie czynności, „pętla dopóki” }
      if a > b then a := a-b else b := b-a;    { instrukcja warunkowa „jeżeli” }
   writeln('  NWD(', m, ',', n, ') = ', a);    { wyświetlenie wyniku na ekranie }
   repeat until KeyPressed;          { „przytrzymanie” ekranu — oczekiwanie na klawisz }
end.                                 { koniec programu }
```

Przykład programu z funkcją

```
program Euklides_2;
uses Crt;
var m, n: integer;
{..... PODPROGRAM .....}
function euk(a, b: integer): integer;
begin
   while a <> b do
```

```
      if a > b then a := a - b else b := b - a;
   euk := a;
end;
{ ..... PROGRAM GŁÓWNY .....}
begin
   ClrScr;
   write('   I liczba = ');
   readln(m);
   write('  II liczba = ');
   readln(n);
   writeln('  NWD(', m, ',', n, ') = ', euk(m, n));
   repeat until KeyPressed;
end.
```

Przykład programu z procedurą

Włączenie kursora myszy w trybie tekstowym.

```
program p_mysz;
uses crt, dos;
var rejestr: registers;
{..... PODPROGRAM .....}
procedure mysz;
begin
   rejestr.AX := 0;
   intr($33, rejestr);
   rejestr.AX := 1;
   intr($33, rejestr);
end;
{ ..... PROGRAM GŁÓWNY .....}
begin
   clrscr;
   mysz;
   repeat until KeyPressed;
end.
```

Kompilatory, edytory programistyczne i środowiska języka Pascal

Windows

Turbo Pascal, Borland Pascal, Free Pascal, TMT Pascal Lite, DPascal, Dev-Pascal, Virtual Pascal, Irie Pascal, Hugo, Delphi, Lazarus

Linux

fpc (Free Pascal Compiler), Lazarus

Kompilacja: `fpc moje.pas`

Uruchomienie: `./moje`

Typy liczbowe

Liczby całkowite

Typ zmiennej	Zakres	Rozmiar pamięci [bajty]
`Byte`	0 – 255	1
`Word`	0 – 65535	2
`ShortInt`	–128 – 127	1
`Integer`	–32768 – 32767	2
`LongInt`	–2147483648 – 2147483647	4

Liczby rzeczywiste wymierne

Typ zmiennej	Zakres	Rozmiar pamięci [bajty]
`Real`	$2{,}9 \cdot 10^{-39} – 1{,}7 \cdot 10^{38}$	6
`Single`	$1{,}5 \cdot 10^{-45} – 3{,}4 \cdot 10^{38}$	4
`Double`	$5 \cdot 10^{-324} – 1{,}7 \cdot 10^{308}$	8
`Extended`	$3{,}4 \cdot 10^{-4932} – 1{,}1 \cdot 10^{4932}$	10
`Comp`	$–9{,}2 \cdot 10^{18} – 9{,}2 \cdot 10^{18}$	8

Funkcje matematyczne

`abs` — wartość bezwzględna
`sqrt` — pierwiastek kwadratowy
`sqr` — kwadrat liczby
`exp` — liczba *e* do potęgi
`frac` — część ułamkowa liczby
`int` — część całkowita liczby
`ln` — logarytm naturalny
`round` — zaokrąglenie matematyczne
`trunc` — obcięcie części ułamkowej liczby
`sin` — sinus
`cos` — cosinus
`arctan` — arcus tangens

Instrukcja warunkowa

Jedną z podstawowych instrukcji warunkowych jest instrukcja `if-else`.

Instrukcja może mieć następujące składnie:

```
if wyrażenie then instrukcja;
if wyrażenie then instrukcja_1 else instrukcja_2;
if wyrażenie then begin instrukcje_a; end else begin instrukcje_b; end;
```

W wyrażeniach logicznych stosujemy operatory porównań =, <>, <, >, <=, >= oraz operatory logiczne `and` i `or`.

Instrukcje iteracyjne

Iteracja (łac. *iteratio* — powtarzanie) — czynność powtarzania (najczęściej wielokrotnego) tej samej instrukcji (albo wielu instrukcji) w pętli.

Pętla `for` („dla”)

```
for  k := wartość_początkowa to wartość końcowa do instrukcja;
```

Zmienna `k` musi być typu całkowitego i jest zwiększana po każdym przebiegu pętli o 1. Przejście pętli odbywa się za pomocą zmiennej iteracyjnej.

Najpierw jest jej nadawana wartość początkowa, następnie sprawdzane jest, czy nie przekracza ona wartości końcowej, wykonywana jest podana instrukcja, na koniec następuje zwiększenie zmiennej iteracyjnej (tzw. licznika pętli) o 1 i czynności się powtarzają.

Pętla `repeat-until` („powtarzaj … aż do”)

```
repeat instrukcje; until warunek;
```

Stosujemy ją wówczas, gdy nie znamy dokładnej liczby powtórzeń, ale chcemy, aby podana instrukcja wykonała się chociaż jeden raz. Instrukcje są wykonywane tak długo, dopóki podany warunek jest fałszywy, a gdy się on spełni, następuje zakończenie powtarzania.

Pętla `while-do` („dopóki”)

```
while warunek do instrukcje;
```

Stosujemy ją wówczas, gdy nie znamy dokładnej liczby powtórzeń, ale możliwe, że podana instrukcja nie będzie wykonana ani razu. Instrukcje są wykonywane tak długo, dopóki warunek jest prawdziwy.

Funkcje

Funkcja to zblokowany ciąg instrukcji umożliwiający wielokrotne ich stosowanie z różnymi parametrami.

Funkcja to inaczej podprogram.

Funkcję deklarujemy w następujący sposób:

```
function nazwa_funkcji (parametry i ich typy): typ_funkcji;
```

Zamiast funkcji bez parametrów lepiej jest zastosować w Pascalu procedurę.

Procedurę deklarujemy w następujący sposób:

```
procedure nazwa_procedury (parametry i ich typy);
```

C++

Język **C++** został zaprojektowany przez Bjarne'a Stroustrupa jako następca C, rozbudowany głównie o obiektowość. W fazie tworzenia język nazywał się roboczo „C z klasami". W latach 90. zyskał dużą popularność zarówno w programowaniu systemowym, jak i użytkowym. C++ jest uznawany za jeden z najlepszych języków do programowania obiektowego. Programiści wybierają język C++, jeżeli wymagana jest najwyższa wydajność, bezpośredni dostęp do systemu i niezawodność. C++ wpłynął znacząco na późniejsze języki, jak Java czy PHP, które mają wiele wspólnych cech syntaktycznych.

Struktura programu źródłowego

Plik źródłowy programu jest plikiem tekstowym z poleceniami i instrukcjami języka.

Plik z nadanym rozszerzeniem *c* zawiera program napisany w języku C, a plik z rozszerzeniem *cpp* to program źródłowy C++.

W wyniku kompilacji powstaje kod pośredni w pliku *o* lub *obj*, a następnie po konsolidacji plik wykonywalny.

```
#include ......;   — dołączenie bibliotek
main()             — program główny
{                  — początek programu
.............  ;   — instrukcje
}                  — koniec programu
```

Przykład programu

```
#include <iostream>                  // dołączenie biblioteki wejścia/wyjścia — „input/output"
using namespace std;                 // użycie przestrzeni nazw std — „stream display"
main()                               // program główny
{                                    // początek programu
   int a, b, m, n;                   // deklaracje zmiennych typu całkowitego
   system("CLS");                    // Clear Screen — „czyszczenie" ekranu
   cout << "   I liczba = ";         // wyświetlenie tekstu na ekranie
   cin >> a;                         // wczytanie I liczby z klawiatury do zmiennej
   m = a;                            // podstawienie
   cout << "  II liczba = ";         // wyświetlenie tekstu na ekranie
   cin >> b;                         // wczytanie II liczby z klawiatury do zmiennej
   n = b;                            // podstawienie
   while(a != b)                     // powtarzanie czynności, „pętla dopóki"
      if (a > b) a =a - b; else b = b - a;   // instrukcja warunkowa „jeżeli"
   cout<<"  NWD(" << m << "," << n << ") = " << a << endl;    //' wyprowadzenie wyniku

   system("PAUSE");                  // „przytrzymanie" ekranu — oczekiwanie na klawisz
}                                    // koniec programu
```

Przydatne biblioteki

<iostream> — konieczna! operacje wejścia/wyjścia
<cstring> — obsługa tekstów
<cmath> — funkcje matematyczne
<fstream> — operacje plikowe
<sstream> — obsługa strumieni
<iomanip> — umożliwia m.in. ustalanie w liczbach liczby miejsc po przecinku
<cstdlib> — zawiera m.in. funkcje konwersji typów

Przykład programu z funkcją

```
#include <iostream>
using namespace std;
//..... PODPROGRAM .....
int euk(int a, int b)
{
   while(a != b)
   if (a > b) a = a - b; else b = b - a;
   return a;
}
//..... PROGRAM GŁÓWNY .....
main()
{
   int m, n;
   system("CLS");
   cout<<"   I liczba = ";
   cin>>m;
   cout<<"  II liczba = ";
   cin>>n;
   cout<<"  NWD(" << m << "," << n << ") = " << euk(m, n) << endl;
   system("PAUSE");
}
```

Kompilatory, edytory programistyczne i środowiska języka C++

Windows

Dev-C++, C++ Builder, Code::Blocks, Turbo C++, Dev-C++, Eclipse, Watcom C++, NetBeans, Intel C++ Compiler, Magic C++, Ulitmate++, Borland C++, Microsoft Visual C++, Microsoft Visual Studio

Linux

g++, Eclipse, Code::Blocks

Kompilacja: `g++ -o moje moje.cpp`

Uruchomienie: `./moje`

Typy liczbowe

Liczby całkowite

Typ zmiennej	Zakres	Rozmiar pamięci [bajty]
`char`	0 – 255	1
`wchar_t`	0 – 65535	2
`short int`	–32768 – 32767	2
`int`	–2147483648 – 2147483647	2 lub 4
`long int`	–2147483648 – 2147483647	4
`long long int`	$-9{,}2 \cdot 10^{18} - 9{,}2 \cdot 10^{18}$	8

Liczby rzeczywiste wymierne

Typ zmiennej	Zakres	Rozmiar pamięci [bajty]
`float`	$3{,}4 \cdot 10^{-38} - 3{,}4 \cdot 10^{38}$	4
`double`	$1{,}7 \cdot 10^{-308} - 1{,}7 \cdot 10^{308}$	8
`long double`	$3{,}4 \cdot 10^{-4932} - 1{,}1 \cdot 10^{4932}$	8, 10 lub 12

Funkcje matematyczne

`abs` — wartość bezwzględna
`fabs` — wartość bezwzględna liczby zmiennoprzecinkowej
`sqrt` — pierwiastek kwadratowy
`cbrt` — pierwiastek trzeciego stopnia
`pow` — potęgowanie
`exp` — liczba *e* do potęgi
`log` — logarytm naturalny
`log10` — logarytm o podstawie 10
`ceil` — zaokrąglenie w górę
`floor` — zaokrąglenie w dół
`round` — zaokrąglenie matematyczne
`trunc` — obcięcie części ułamkowej liczby
`sin` — sinus
`cos` — cosinus
`tan` — tangens
`asin` — arcus sinus
`acos` — arcus cosinus
`atan` — arcus tangens

Instrukcja warunkowa

Podstawową instrukcją warunkową jest instrukcja `if-else`.

Instrukcja może mieć następujące składnie:

```
if(wyrażenie) instrukcja;
if(wyrażenie) instrukcja_1; else instrukcja_2;
if(wyrażenie) { instrukcje_1 } else { instrukcje_2 };
```

W wyrażeniach logicznych stosujemy operatory porównań ==, !=, <, >, <=, >= oraz operatory logiczne && (*and*) i || (*or*).

Instrukcje iteracyjne

1. Pętla `for` („dla”)

   ```
   for(wartość_początkowa; warunek; krok) {instrukcje};
   ```

   Przejście pętli odbywa się za pomocą zmiennej iteracyjnej. Najpierw jest jej nadawana wartość początkowa, następnie sprawdzany jest warunek logiczny i jeśli jest on prawdziwy, to wykonywana jest podana instrukcja, na koniec następuje zmiana zmiennej iteracyjnej (tzw. licznika pętli) i ponownie sprawdzany jest warunek itd.

2. Pętla `do-while` („wykonuj … dopóki”)

   ```
   do{instrukcje} while (warunek);
   ```

   Stosujemy ją wówczas, gdy nie znamy dokładnej liczby powtórzeń, ale chcemy, aby podana instrukcja wykonała się chociaż jeden raz. Instrukcje są wykonywane tak długo, dopóki podany warunek jest prawdziwy.

3. Pętla `while` („dopóki”)

   ```
   while(warunek) {instrukcje}
   ```

   Stosujemy ją wówczas, gdy nie znamy dokładnej liczby powtórzeń, ale możliwe, że podana instrukcja nie będzie wykonana ani razu. Instrukcje są wykonywane tak długo, dopóki warunek jest prawdziwy.

   W przypadku zastosowania kilku funkcji w programie należy pamiętać o tym, że przy braku ich deklaracji (prototypów) i wywoływaniu jednej funkcji przez drugą bardzo ważna jest kolejność zapisu ich definicji.

   Dlatego zaleca się tworzenie nagłówków funkcji, wówczas kolejność ich zapisu będzie nieistotna.

   Wyjaśnia do poniższy przykład.

   ```
   #include <iostream>
   using namespace std;

   // funkcja
   int fun1()
   {
      fun2();    // błąd kompilacji! „fun2” niezadeklarowane
      return 0;
   }
   // funkcja
   int fun2()
   {
   ```

```
    return 0;
}
// funkcja główna
main()
{
    return 0;
}
```

Przykład programu z deklaracjami funkcji

```
#include <iostream>
using namespace std;

// nagłówki funkcji (na końcu średnik!)
int fun1();    // dzięki temu
int fun2();    // kompilacja bez błędów

// funkcja
int fun1()
{
   fun2();
   return 0;
}
// funkcja
int fun2()
{
   return 0;
}
// funkcja główna
main()
{
   return 0;
}
```

W języku C++ istnieje możliwość kontrolowania strumienia wejściowego przy użyciu metod `cin.good()` oraz `cin.fail()`.

Należy też pamiętać, że liczby zmiennoprzecinkowe w zależności od typu są zapamiętywane zawsze z pewnym przybliżeniem, co może prowadzić do błędów w obliczeniach.

W poniższym przykładzie może pojawić się komunikat, że liczby `y` i `z` nie są równe!

```
double x, y, z;

   x = 5.0;
   y = 4.0 * x - 25.8;
   z = -5.8;

if(z == y) cout << "z i y są równe";
else cout << "z oraz y NIE są równe!";
```

JavaScript

JavaScript to obiektowy język skryptowy stosowany głównie przy projektowaniu stron WWW i interpretowany w środowisku przeglądarki internetowej, ale można w nim tworzyć także samodzielne aplikacje. Microsoft udostępnia biblioteki umożliwiające tworzenie aplikacji JS jako część środowiska Windows Scripting Host. JavaScript został wynaleziony przez Brendana Eicha w roku 1995 i opracowany w firmie Netscape. W 1996 roku organizacja ECMA rozpoczęła pracę nad specyfikacją języka. ECMAScript (wersja 7) wydany w roku 2016 to najnowsza oficjalna wersja języka JavaScript.

Nie należy mylić JavaScriptu z językiem Java, bo są to zupełnie różne języki, zarówno w koncepcji, jak i konstrukcji, chociaż mają wiele wspólnych cech dotyczących składni.

```
<script type="text/javascript">
   kod skryptu
</script>
```

Przykład skryptu z zastosowaniem formularza

```
<html>
<head>
   <script type="text/javascript">

      function dodaj()
      {
      // ..... UCHWYTY .....
         var uch1 = document.getElementById("pole1");
         var uch2 = document.getElementById("pole2");
         var uch3 = document.getElementById("dif");

      // ..... POBIERANIE DANYCH Z PÓL TEKSTOWYCH .....
         var a = uch1.value;
         var b = uch2.value;

      // ..... KONWERSJA NA LICZBY CAŁKOWITE .....
         a = parseInt(a);
         b = parseInt(b);

      // ..... WYKONANIE DZIAŁAŃ NA LICZBACH .....

         c = a + b;
      // ..... PRZESYŁANIE WYNIKU DO FORMULARZA .....
         uch3.innerHTML = c;

         return 0;
      }

   </script>
</head>

<!--..... TWORZENIE FORMULARZA W SEKCJI BODY ..... -->
<body>
<form>
   I liczba  = <input type = "text" id = "pole1" size = "5">
   II liczba = <input type = "text" id = "pole2" size = "5">
```

```
   <input type = "button" value = "DODAJ" onclick = "dodaj()"> <br>
   <div id = "dif">  </div>
</form>
</body>
</html>
```

Wyniki i komunikaty można wyprowadzać w JS na kilka sposobów:

1. Za pomocą okienka `alert`

   (nie polecamy do wyprowadzania wyników, może być wykorzystane do komunikatów)

   ```
   alert(2 * 3 + 8);
   ```

2. Za pomocą polecenia `document.write()`

   ```
   document.write(2 * 3 + 8);
   ```

3. Za pomocą uchwytu do obiektu i metody `innerHTML`

   ```
   document.getElementById("xxx").innerHTML = 2 * 3 + 8;
   ```

   gdzie obiekt `"xxx"` musi by zdefiniowany w kodzie HTML np. jako paragraf `<p id="xxx"> </p>`

4. W konsoli przeglądarki, korzystając z polecenia `console.log()`

   ```
   console.log(2*3+8);
   ```

Konsolę w przeglądarkach otwiera się na różne sposoby. W przeglądarkach Firefox, Chrome i Opera można to zrobić za pomocą skrótu klawiszowego *Ctrl+Shift+I*.

Za pomocą kodu JS możemy sprawdzić, czy formularz HTML jest poprawnie wypełniony przez użytkownika (walidacja). Nie chcemy, aby brakowało danych albo były one niepełne. Możemy również wskazać, w jakim formacie powinny być wypełniane pola tekstowe. W podanym przykładzie dostęp do obiektu formularza odbył się bez stosowania uchwytu, poprzez identyfikator.

```
<html>
<head>
   <script type = "text/javascript">
      function sprawdz()
      {
         var f = document.forms["formularz"];
         if (f.t.value == "") alert("Brak danych!");
      }
   </script>
</head>
<body>

<form id = "formularz">
   Wpisz tekst: <input type = "text" id = "t">
   <input type = "button" value = "Wprowadź" onclick = "sprawdz()">
</form>

</body>
</html>
```

Komunikat nie musi być wyprowadzany w funkcji `sprawdz()`. Może ona zwracać wartości `true` albo `false` do wykorzystania w kolejnych czynnościach.

```
function sprawdz()
   {
      var f=document.forms["formularz"];
         if(f.t.value == "") return false;
         else return true;
   }
```

Instrukcje i funkcje w kodzie JS mogą być zapisywane w osobnych plikach o rozszerzeniu *js*. Skraca to kod HTML, ułatwia wyszukiwanie błędów i eliminuje konieczność powielania kodu, gdy jest on wykorzystywany na wielu podstronach.

Przykład wywołania skryptu z pliku .js

HTML

```
<html>
<head>
<meta http-equiv = "Content-Type" content = "text/html; charset = UTF-8">
</head>
<body>
   Witaj!
   <script language = "JavaScript" type = "text/javascript" src = "czas.js">
   </script>
</body>
</html>
```

JS

```
// Pobieranie czasu
var dane = new Date();
godz = dane.getHours();
min = dane.getMinutes();
sek = dane.getSeconds();

// Wyświetlenie w dokumencie HTML
document.write("<br>");
document.write(godz +":" + min + ":" + sek);
document.write("<br>");

// Wyświetlenie tekstu
if (godz >= 5 && godz<19) document.write("Dzień dobry.");
else
   if (godz >= 19 && godz<23) document.write("Dobry wieczór.");
   else
      document.write("Czemu nie śpisz?");
```

Funkcje matematyczne

`Math.abs` — wartość bezwzględna

`Math.sqrt` — pierwiastek kwadratowy

`Math.pow` — potęgowanie

`Math.exp` — liczba *e* do potęgi

`Math.log` — logarytm naturalny

`Math.max` — największa z podanych liczb
`Math.min` — najmniejsza z podanych liczb
`Math.random` — liczba losowa z przedziału [0; 1)
`Math.ceil` — zaokrąglenie w górę
`Math.floor` — zaokrąglenie w dół
`Math.round` — zaokrąglenie matematyczne
`Math.sin` — sinus
`Math.cos` — cosinus
`Math.tan` — tangens
`Math.asin` — arcus sinus
`Math.acos` — arcus cosinus
`Math.atan` — arcus tangens

Python

Python został opracowany w latach 90. przez Guido van Rossuma w Centrum Matematyki i Informatyki w Amsterdamie. Jest to język programowania wysokiego poziomu o wszechstronnym przeznaczeniu, oferujący duży wybór bibliotek standardowych. Charakteryzuje się czytelnością i przejrzystością kodu. W porównaniu z innymi językami porównywalny kod jest krótki i szybki w tworzeniu. Może być stosowany w każdym systemie operacyjnym i nadaje się znakomicie do typowych zastosowań w zakresie różnych obliczeń i przetwarzania danych. Pozostawia programiście wiele swobody. Python rozwijany jest jako projekt open source zarządzany przez Python Software Foundation.

Interpretery i edytory

Jython (Java), IronPython (.NET), ActivePython, PyPy

W systemach opartych na Linuksie interpreter Pythona jest standardowo zainstalowany. W systemach Microsoft Windows należy zainstalować odpowiednie oprogramowanie.

Ponieważ w pisaniu kodu istotną rolę spełniają wcięcia, które określają bloki programu, wygodnie jest użyć odpowiednich edytorów, np. IDE Geany lub PyCharm. Obydwa programy działają na platformach Linux i Windows.

Instrukcje złożone (bloki) w Pythonie muszą zawierać nagłówek zakończony dwukropkiem, a wchodzące w skład bloku instrukcje muszą być wcięte w stosunku do nagłówka. Wcięcia podczas tworzenia kodu źródłowego w jakimś edytorze można tworzyć przy użyciu tabulatora albo spacji, przy czym liczba tabulacji bądź spacji jest nieistotna i w każdym bloku może być inna. Ważne jest, aby w jednym bloku wszystkie instrukcje były jednakowo zagnieżdżone, czyli miały takie same wcięcia. Inaczej interpreter wskaże błąd.

Wprowadzenie dotyczy wersji Python 2.7, w tej wersji są prezentowane wszystkie rozwiązania w tej książce.

Typy liczbowe

Podstawowe typy liczbowe to `int` (liczba całkowita) i `float` (liczba wymierna). Można na nich wykonywać działania +, -, *, /.


Uwaga

```
>>> print 3 / 4
0
```

Jeżeli oba argumenty są całkowite, wykonywane jest dzielenie całkowitoliczbowe. Jeżeli chcemy wymusić wynik w postaci wymiernej, powinniśmy napisać `3.0 / 4`.

Operator ** odpowiada za działanie potęgowania.

Dla dwóch argumentów całkowitych operator `%` zwraca resztę z dzielenia.

`int(x)` to operacja rzutowania liczby na liczbę całkowitą (zaokrąglenie w dół).

`float(x)` to zamiana liczby na liczbę wymierną.

Wyrażenie `abs(x)` zwraca wartość bezwzględną liczby `x`.

Działania matematyczne

Jeżeli chcemy mieć dostęp do większej liczby funkcji, importujemy je z modułów. Na przykład moduł `math` dostarcza funkcji matematycznych:

- `acos` — arcus cosinus
- `acosh` — arcus cosinus hiperboliczny
- `asin` — arcus sinus
- `asinh` — arcus sinus hiperboliczny
- `atan` — arcus tangens
- `atan2` — arcus tangens ilorazu argumentów, odtwarza kąt z przedziału $(\pi, \pi]$
- `atanh` — arcus tangens hiperboliczny
- `ceil` — zaokrąglenie w górę, wynik typu `float`
- `copysign` — zwraca pierwszy argument ze znakiem drugiego argumentu
- `cos` — cosinus
- `cosh` — cosinus hiperboliczny
- `degrees` — konwersja z radianów do stopni
- `e` — wartość liczby *e*
- `erf` — dystrybuanta rozkładu Gaussa
- `erfc` — 1–erf
- `exp` — funkcja wykładnicza
- `expm1` — exp–1
- `fabs` — wartość bezwzględna dla typu `float`
- `factorial` — silnia
- `floor` — zaokrąglenie w dół, wynik typu `float`
- `fmod` — reszta z dzielenia dla typu `float`

`frexp` — zwraca parę mantysa, eksponent
`fsum` — sumowanie po typie sekwencyjnym lub iteracyjnym o elementach `float`
`gamma` — funkcja Γ (uogólnienie silni na liczby wymierne i zespolone)
`hypot` — przeciwprostokątna trójkąta prostokątnego o podanych bokach
`isinf` — sprawdzenie, czy wartość typu `float` jest nieskończona
`isnan` — sprawdzenie, czy wartość typu `float` nie jest liczbą
`ldexp` — wartość liczby o podanej mantysie i eksponencie
`llgamma` — logarytm naturalny z wartości bezwzględnej funkcji Γ
`log` — logarytm naturalny
`log10` — logarytm przy podstawie 10
`log1p` — logarytm naturalny z $1 + x$
`modf` — zwraca część całkowitą i ułamkową z liczby typu `float`
`pi` — wartość liczby π
`pow` — potęga wymierna liczby wymiernej
`radians` — konwersja ze stopni do radianów
`sin` — sinus
`sinh` — sinus hiperboliczny
`sqrt` — pierwiastek
`tan` — tangens
`tanh` — tangens hiperboliczny
`trunc` — zaokrąglenie do najniższej (w sensie wartości bezwzględnej) liczby całkowitej

Jeżeli chcemy skorzystać z tych funkcji, wykonujemy instrukcję importu:

```
from math import sin, atan, acos, exp, sqrt
```

Powyższa linijka importuje funkcje sinus, arcus tangens, arcus cosinus, funkcję wykładniczą i pierwiastek kwadratowy. Od tej chwili można ich używać w programie.

Listy i krotki

Lista to uporządkowany zbiór obiektów. Obiekty jednej listy mogą być różnych typów, mogą być również listami. Listę definiujemy, umieszczając jej elementy w nawiasach kwadratowych, oddzielone przecinkami.

Na przykład:

```
lista = [0.1, 2.0, 3, [1.1, 2]]
```

stworzy listę zawierającą dwie liczby wymierne, liczbę całkowitą i listę zawierającą liczbę wymierną i całkowitą.

Krotka jest listą, której nie można modyfikować. Zajmuje mniej pamięci i powinniśmy jej używać, jeżeli nie potrzebujemy jej modyfikować. Krotkę definiujemy, umieszczając jej elementy oddzielone przecinkami w nawiasach okrągłych:

```
krotka = (0.1, 2.0, 3, [1.1, 2])
```

Definicja krotki musi zawierać przecinek, nawet jeżeli krotka ma mieć tylko jeden element:

```
krotka = (0.1, )
```

Operacje na listach

- **Indeksowanie** — składnia `lista[i]` zwraca `i`-ty element listy. Indeksy numerujemy od 0 do liczby, która jest wartością długości listy pomniejszoną o 1. Jeżeli podamy indeks ze znakiem ujemnym, będzie on liczony „w drugą stronę": `lista[-1]` to ostatni element listy `lista`.
- **Pobieranie wycinków** — składnia `lista[i:j]` zwraca listę złożoną z elementów listy `lista` od `i`-tego do `j - 1`. Indeksy mogą przyjmować wartości ujemne, jak wyjaśniliśmy w punkcie wyżej. Brak indeksu początkowego (`lista[:j]`) oznacza: od początku do `j - 1`. Brak indeksu końcowego (`lista[i:]`) oznacza od `i` do końca. W pobieraniu wycinka możliwe jest podanie trzech liczb `lista[i:j,k]`. Oznacza to elementy od `i`-tego do `j - 1` co `k`. Wartość ujemna `k` oznacza kierunek przeciwny. W szczególności `lista[::-1]` zwraca listę `lista` z elementami ułożonymi odwrotnie.
- **Dodawanie list** — operacja `lista1 + lista2` zwraca listę, której elementami są najpierw elementy `lista1`, potem elementy `lista2`.
- **Dodawanie elementu** — instrukcja `lista.append(x)` dodaje obiekt `x` na koniec listy.
- **Pobieranie ostatniego elementu** — instrukcja `lista.pop()` zwraca ostatni element listy, usuwając go z niej.
- **Sumowanie elementów** — jeżeli `lista` zawiera tylko liczby, można je zsumować instrukcją `sum(lista)`.
- **Pobieranie elementów maksymalnego i minimalnego** — jeżeli `lista` zawiera tylko liczby, funkcje `max` i `min` zwracają jej elementy — odpowiednio maksymalny i minimalny.
- **Sprawdzanie długości listy** — wyrażenie `len(lista)` zwraca długość listy `lista`.
- **Instrukcja zip** — dostaje ona jako argumenty listy, a zwraca pojedynczą listę krotek pierwszych elementów, drugich elementów itd. Wynik będzie miał tyle elementów, ile miał najkrótszy argument.

```
>>> zip ([1.2, 1.3, 1.4], [5, 6, 7, 8] ,[0.25, 0.75])
[(1.2, 5, 0.25), (1.3, 6, 0.75)]
```

Łańcuchy znaków

Łańcuch znaków definiujemy, umieszczając tekst pomiędzy parą znaków " lub '. Jeżeli tekst ma wiele wierszy, umieszczamy go pomiędzy parą potrójnych znaków " (""") lub ' ('''). Na łańcuchach możemy wykonywać wiele operacji dostępnych dla list, m.in. indeksowanie, pobieranie wycinków, dodawanie, funkcję `len`. Dodatkowo na łańcuchach można wykonywać poniższe operacje:

- **Dzielenie po separatorze** — dla łańcucha `s` wyrażenie `s.split(x)` zwraca listę podłańcuchów łańcucha, rozdzielonych podłańcuchem separującym `x`.

Przykład:

```
>>> s = '192.168.2.51'
>>> s.split ('.')
['192', '168', '2', '51']
```

Jeżeli interesuje nas tylko podział do pierwszego separatora, to wyrażenie `s.partition(x)` zwraca listę trzech elementów: podłańcuch do pierwszego wystąpienia separatora `x`, separator, wszystko za pierwszym separatorem. Jeżeli separator nie jest podłańcuchem `s`, ostatnie dwa elementy listy będą puste.

- **Łączenie listy łańcuchów w jeden łańcuch poprzez łańcuch separujący** — operacja odwrotna do powyższej. Wyrażenie `x.join(l)` poskleja wszystkie łańcuchy z listy `l` w jeden łańcuch, w miejsce łączenia wstawiając łańcuch separujący `x`. W szczególności łańcuch `x` może być pusty.

  Przykład:

  ```
  >>> l = ['192', '168', '2', '51']
  >>> '.'.join(l)
  '192.168.2.51'
  >>> ''.join(l)
  '192168251'
  ```

- **Znajdowanie podłańcucha** — wyrażenie `s.find(x)` znajdzie miejsce pierwszego wystąpienia podłańcucha `x` w łańcuchu `s`. Wartość `-1` oznacza, że łańcuch nie zawiera podłańcucha.
- **Wartość liczbowa znaku** — funkcja `ord` zwraca wartość liczbową pojedynczego znaku ASCII. W drugą stronę funkcja `chr` zwraca znak o kodzie odpowiadającym podanej wartości liczbowej.

Wyrażenia logiczne

Wyrażenie logiczne może przyjmować dwie wartości: `True` (prawda) lub `False` (fałsz). Wyrażenia logiczne można tworzyć za pomocą operatorów porównania: =, <, >, <=, >=. Wyrażenia możemy łączyć za pomocą operatorów logicznych, z których podstawowe to `and` (i) oraz `or` (lub).

Mając daną listę `l` wartości logicznych, funkcja `all(l)` zwraca `True`, jeżeli wszystkie elementy listy mają wartość `True`, w przeciwnym wypadku zwraca `False` (kwantyfikator ogólny). Funkcja `exists(l)` zwraca `True`, jeżeli przynajmniej jeden element listy przyjmuje wartość `True`, w przeciwnym wypadku zwraca `False` (kwantyfikator szczegółowy).

Obiekty innych typów w sposób naturalny są rzutowane na wartości logiczne i mogą być używane tam, gdzie wymagana jest wartość logiczna:

- listy: pusta → `False`, niepusta → `True`;
- łańcuchy znaków: pusty → `False`, niepusty → `True`;
- liczby: 0 → `False`, różne od 0 → `True`.

Instrukcje bitowe

Dwuargumentowe instrukcje bitowe to instrukcje wykonywane na wszystkich parach bitów o tych samych pozycjach. Najważniejsze to `&` (i — koniunkcja), `|` (lub — alternatywa) oraz `^` (albo — alternatywa wyłączająca):

```
  11101011        11101011        11101011
& 00011011    |   00011011    ^   00011011
  00001011        11111011        11110000
```

Instrukcje blokowe

Instrukcje złożone (bloki) w języku Python muszą zawierać nagłówek zakończony dwukropkiem, a wchodzące w skład bloku instrukcje muszą być wcięte w stosunku do nagłówka. Wcięcia podczas tworzenia kodu źródłowego w edytorze można tworzyć za pomocą tabulatora albo spacji, przy czym liczba tabulacji bądź spacji jest nieistotna i w każdym bloku może być inna. Ważne jest, aby w jednym bloku wszystkie instrukcje były jednakowo zagnieżdżone, czyli miały takie same wcięcia. Inaczej interpreter wskaże błąd.

Pętla for

Nagłówek pętli `for` wygląda następująco:

```
for element in sekwencja:
```

Instrukcje bloku są powtarzane po kolei dla wszystkich elementów obiektu `sekwencja`. Obiekt `sekwencja` może być łańcuchem znaków lub listą (w ogólności typem iteracyjnym — „rozumiejącym", co znaczy „następny" — lub typem iteracyjnym — pozwalającym brać elementy po indeksach). Listę kolejnych liczb całkowitych zwraca instrukcja `range(początek, koniec, krok)`.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(2, 10)
[2, 3, 4, 5, 6, 7, 8, 9]
>>> range(2, 10, 3)
[2, 5, 8]
```

Jeżeli chcemy wykonać iterację po kolejnych liczbach całkowitych, nie musimy tworzyć ich listy. Instrukcja `xrange` wywołana z tymi samymi argumentami zwróci tzw. iterator, czyli obiekt, który proszony o kolejne liczby, zwraca je, a gdy wyczerpie zakres, informuje o tym, nie generując wszystkich wyników naraz i bez zajmowania pamięci:

```
>>> for i in xrange(5):
... print i
...
0
1
2
3
4
```

Pętla while

Nagłówek pętli `while` ma postać:

```
while warunek:
```

Blok instrukcji jest wywoływany tak długo, jak długo wyrażenie `warunek` przyjmuje wartość `True`.

Instrukcja warunkowa

Najprostsza instrukcja warunkowa ma postać:

```
if warunek:
```

gdzie `warunek` to dowolne wyrażenie zwracające wartość logiczną `True` albo `False`. Blok instrukcji występujący po nim jest wywołany, jeżeli `warunek` ma wartość `True`, w przeciwnym wypadku jest ignorowany. Możemy również od razu zdefiniować, co ma się dziać, gdy warunek nie jest spełniony za pomocą bloku `else` następującego od razu po bloku `if`.

```
if T > 15:
   print 'Ciepło'
else:
   print 'Zimno'
```

To samo działanie możemy uzyskać, pisząc po bloku `if` kolejny blok `if` z zaprzeczonym warunkiem. Używając bloku `else`, dokonujemy o jedno sprawdzanie warunku mniej i zwiększamy czytelność kodu. Instrukcję wielokrotnego wyboru (odpowiednik `switch: case` w C) można zrealizować za pomocą zagnieżdżonej instrukcji `if-else`.

```
if T > 20:
   print 'Ciepło'
else:
   if T > 10:
      print 'Umiarkowanie'
   else:
      print 'Zimno'
```

Zdecydowanie czytelniej jest użyć w takiej instrukcji `elif`.

```
if T > 20:
   print 'Ciepło'
elif T > 10:
   print 'Zimno'
else:
   print 'Umiarkowanie'
```

Po bloku `if` może następować dowolnie wiele bloków `elif`, po których może (ale nie musi) następować jeden blok `else`. Warunek w bloku `elif` jest sprawdzany tylko, jeżeli wszystkie powyższe bloki `elif` oraz wiodący blok `if` dały wartość `False`.

Trójargumentowa instrukcja warunkowa

Zamiast pisać kod jak w pierwszym z przykładów użycia bloku `if`, możemy wykorzystać tzw. krótką instrukcję warunkową:

```
print 'Ciepło' if T > 15 else 'Zimno'
```

Blok funkcji

Funkcję definiujemy za pomocą instrukcji `def`. Nagłówek bloku wygląda następująco:

```
def nazwa_funkcji argument_1, argument_2, ..., argument_n):
```

Po nim następuje wcięty blok instrukcji. Funkcja zwraca wartość za pomocą instrukcji `return`. Instrukcja `return` powoduje natychmiastowe wyjście z funkcji. Funkcja może również nie zwracać żadnej wartości (jak funkcja typu `void` w C lub procedura w Pascalu), gdy nie ma w niej instrukcji `return`. W takiej sytuacji, gdy próbujemy pobrać wartość zwracaną przez funkcję, dostajemy wartość `None`. Funkcja może też nie przyjmować żadnego argumentu. Można nadać argumentom wartości domyślne (te argumenty muszą być po prawej stronie w definicji funkcji). Prześledźmy działanie poniższej funkcji:

```
def f(a, b = 1, c = 2):
   print a, b, c
f('Ala ')
f('Ala', 'ma')
f('Ala', 'ma', 'kota')
```

Otrzymamy w wyniku:

```
Ala 1 2
Ala ma 2
Ala ma kota
```

Funkcję możemy też zdefiniować za pomocą instrukcji `lambda`, nie nadając jej nazwy (funkcja anonimowa).

```
f = lambda x : x ** 2 + 1
g = lambda x, y : x ** y    # funkcja dwuargumentowa
```

W programowaniu funkcyjnym używamy funkcji, których argumentem jest inna funkcja. Wtedy argument konstruujemy jako konstrukcję lambda, nie przypisując jej do żadnej nazwy (w sytuacji gdy funkcja będąca argumentem jest nam potrzebna tylko w tym miejscu).

Listy składane

Załóżmy, że mamy listę i chcemy ją przekształcić w drugą listę, podnosząc do kwadratu jej elementy, ale tylko nieparzyste. Wykorzystując wiedzę z poprzednich rozdziałów, napisalibyśmy kod:

```
druga = []      # (2)
for i in pierwsza:
   if i % 2:    # liczba 0 jest traktowana jako False, każda inna jako True
      druga.append(i ** 2)
```

Minimalnie szybsza, ale o wiele bardziej czytelna i zwarta w zapisie jest charakterystyczna dla Pythona składnia listy składanej.

```
druga = [i ** 2 for i in pierwsza if i % 2]
```

Jest bardzo intuicyjna i przypomina matematyczny zapis $D = \{x^2 \mid x \in P \wedge x \bmod 2 \neq 0\}$ (`druga` to zbiór kwadratów tych elementów listy `pierwsza`, których reszta po podzieleniu przez 2 nie jest 0).

Jeżeli za pomocą list składanych mamy wykonać to, co za pomocą dwóch (lub więcej) zagnieżdżonych pętli `for`, ich nagłówki podajemy w konstrukcji listy składanej w takiej samej kolejności. Jako przykład rozważmy fragment kodu, który generuje wszystkie liczby dwucyfrowe, których druga cyfra jest mniejsza niż pierwsza:

```
[10* i + j for i in xrange(1, 10) for j in xrange(i)]
```

Programowanie funkcyjne

Spośród różnych stylów programowania (proceduralne, deklaratywne, obiektowe, funkcyjne) tylko programowanie funkcyjne polega na przetwarzaniu zbioru danych wejściowych przez kolejne funkcje.

- `map`

  Funkcja wbudowana `map` wymaga dwóch argumentów: funkcji i listy. Daje w wyniku listę, której elementami są wartości funkcji dla kolejnych elementów listy wejściowej.

  Przykład:

  ```
  >>> map(lambda x : x ** 2, range(10))
  [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
  ```

  Powyższy kod generuje kwadraty liczb od 0 do 9.

- `filter`

  Funkcja wbudowana `filter` również wymaga dwóch argumentów: funkcji i listy, przy czym funkcja ma zwracać wartości logiczne. Daje w wyniku listę tych elementów listy wejściowej, dla których funkcja zwraca wartość `True`.

  Przykład:

  ```
  >>> filter(lambda x : x % 2 == 0 or x % 3 == 0, range(20))
  [0, 2, 3, 4, 6, 8, 9, 10, 12, 14, 15, 16, 18]
  ```

  Powyższy kod wygeneruje listę liczb z zakresu od 0 do 19, których reszta z dzielenia przez 2 lub przez 3 wynosi 0.

  Kod, który utworzyliśmy na początku podrozdziału „Listy składane", można przepisać za pomocą `map` i `filter`. Użycie `map` i `filter` zamiast liczb składanych jest szybsze tylko dla funkcji wbudowanych lub importowanych z modułów, dla definiowanych w ciele programu (w tym za pomocą wyrażenia `lambda`) jest wolniejsze. Nie jest to rozwiązanie tak popularne jak listy składane, ale czasem jest uznawane za bardziej czytelne.

- `reduce`

  Funkcja `reduce` przyjmuje standartowo dwa argumenty: funkcję dwuargumentową `f` i listę `l`. Pierwszym argumentem funkcji `f` jest wynik jej ostatniego działania. Jej drugim argumentem jest kolejny argument listy `l`. Funkcja `reduce` wywołuje funkcję `f` na pierwszym i drugim elemencie listy, następnie na wyniku i trzecim elemencie listy itd. aż drugim argumentem funkcji `f` będzie ostatni element listy. Zwykle zaczynamy tu od pierwszego elementu listy, ale możemy poprzedzić go inną wartością startową podawaną jako trzeci, opcjonalny argument.

  Przykład:

  ```
  # obliczenie silni:
  reduce(lambda x, y : x * y, xrange(1, n+1))
  ```

Kolejnym przykładem niech będzie znalezienie wartości liczby, mając daną listę jej cyfr (przykład uogólnia się na dowolny system pozycyjny). Załóżmy, że cyframi są 1, 0, 2, 4. Wartość 1024 otrzymamy, mnożąc pierwszą cyfrę przez 10 i dodając drugą, następnie mnożymy wynik przez 10 i dodajemy trzecią itd.

```
reduce(lambda x, y : 10 * x + y, [1, 0, 2, 4])
```

Możemy zmodyfikować pierwszy przykład tak, aby `reduce` zwracało listę kolejnych silni:

```
>>> reduce(lambda x, y : x + [x[-1] * y], xrange(1, 10+1), [1])
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Porównywanie wydajności rozwiązań

Funkcja `time`, którą importujemy z modułu `time`, zwraca z dużą dokładnością liczbę sekund, które upłynęły od początku roku 1980. Dzięki niej możemy sprawdzać, ile czasu zajmuje wykonanie programu. Jako przykład wykonamy eksperyment porównujący różne sposoby generowania dwukrotności liczb z zakresu od 1 do miliona — konstrukcję listy za pomocą pętli, listę składaną, odwzorowywanie listy instrukcją `map` z funkcją definiowaną przez `lambda` oraz z wbudowaną funkcją dwukrotności. Funkcję tę (metodę wbudowaną) zapisuje się jako `(2).__mul__` (składnia ta pozostanie ciekawostką, jej wyjaśnienie wykracza poza zakres książki). Program testujący wygląda następująco:

```
from time import time
t = time()
l = []
for i in xrange (1000000):
   l.append(2 * i)
print 'konstrukcja przez pętlę:  ', time() - t
t = time()
l = [2 * i for i in xrange(1000000)]
print 'lista składana            ', time() - t
t = time()
l = map(lambda i : 2 * i, xrange(1000000))
print 'map i konstrukcja lambda: ', time() - t
t = time()
l = map((2).__mul__, xrange(1000000))
print 'map i funkcja wbudowana:  ', time() - t
```

Otrzymujemy w wyniku (wartości mogą być różne na różnych komputerach, proporcje powinny być podobne):

```
konstrukcja przez pętlę:       0.147644042969
lista składana:                0.0881679058075
map i konstrukcja lambda:      0.136617898941
map i funkcja wbudowana:       0.0497651100159
```

Potwierdza się to, o czym wspominaliśmy wcześniej — lista składana działa szybciej niż tworzenie nowej listy poprzez pętlę `for`, a użycie instrukcji `map` jest szybsze, jeżeli użyje się funkcji wbudowanej, podanie funkcji poprzez `lambda` powoduje spadek wydajności w porównaniu z listą składaną.

Rozdział 1.
Zadania

Zadanie 1. Czy liczba jest parzysta?

Liczba jest parzysta, jeżeli dzieli się przez 2. Innym sposobem rozpoznania jest sprawdzenie ostatniej jej cyfry — musi to być 0, 2, 4, 6 albo 8.

Jeżeli podana liczba jest większa od 1, to możemy odejmować od niej liczbę 2 tak długo, aż uzyskamy 0 albo 1 i w ten sposób rozstrzygniemy o parzystości.

Sprawdź, czy podana liczba naturalna jest liczbą parzystą.

Zadanie 2. Czy liczba jest pierwsza?

Liczba jest pierwsza, jeśli posiada dokładnie 2 dzielniki.

Sprawdź, czy podana liczba naturalna jest liczbą pierwszą.

Zadanie 3. Czy liczba jest super-pierwsza?

Liczba jest super-pierwsza, jeśli jest liczbą pierwszą oraz suma jej cyfr jest też liczbą pierwszą.

Taką liczbą jest np. 101.

Sprawdź, czy podana liczba naturalna jest liczbą super-pierwszą.

Zadanie 4. Czy liczba jest B-super-pierwsza?

Liczba jest B-super-pierwsza, jeśli jest liczbą super-pierwszą oraz suma dziesiętna jej cyfr w rozwinięciu dwójkowym jest też liczbą pierwszą.

Taką liczbą jest np. 1291.

Sprawdź, czy podana liczba naturalna jest liczbą B-super-pierwszą.

Zadanie 5. Czy liczba jest doskonała?

Liczba doskonała — liczba naturalna, która jest sumą wszystkich swych dzielników właściwych (to znaczy od niej mniejszych).

Najmniejszą liczbą doskonałą jest 6, ponieważ $6 = 3 + 2 + 1$. Następna to 28 ($28 = 14 + 7 + 4 + 2 + 1$).

Sprawdź, czy podana liczba naturalna jest liczbą doskonałą.

Zadanie 6. Czy liczba jest podzielna?

Liczba podzielna — liczba naturalna, która jest większa od 0 i dzieli się przez sumę swoich cyfr.

Taką liczbą jest np. 21.

Sprawdź, czy podana liczba naturalna jest liczbą podzielną.

Zadanie 7. Czy liczba jest liczbą Mersenne'a?

Liczby Mersenne'a to liczby postaci $2^p - 1$, gdzie p jest liczbą pierwszą.

Sprawdź, czy podana liczba naturalna jest liczbą Mersenne'a.

Zadanie 8. Czy liczby są bliźniacze?

Liczby bliźniacze — takie dwie liczby pierwsze, których różnica wynosi 2.

Przykłady liczb bliźniaczych: 3 i 5, 5 i 7, 11 i 13, …

Sprawdź, czy podane dwie liczby naturalne są liczbami bliźniaczymi.

Zadanie 9. Czy liczby są zaprzyjaźnione?

Liczby zaprzyjaźnione to para różnych liczb naturalnych, takich że suma dzielników właściwych każdej z tych liczb równa się drugiej. Dzielniki właściwe są to wszystkie dzielniki danej liczby oprócz niej samej.

Przykłady liczb zaprzyjaźnionych: 220 i 284, 1184 i 1210, 2620 i 2924, 5020 i 5564.

Sprawdź, czy podane dwie liczby naturalne są liczbami zaprzyjaźnionymi.

Zadanie 10. Największy wspólny dzielnik

Algorytm Euklidesa — algorytm wyznacza największy wspólny dzielnik dwóch liczb całkowitych dodatnich.

Algorytm opiera się na dwóch faktach:

- największy wspólny dzielnik liczby i zera to ta liczba;
- największy wspólny dzielnik liczb a i b jest równy największemu wspólnemu dzielnikowi liczb (`b mod a`) i `a`.

Pierwsze wzmianki na temat tego algorytmu pojawiły się w dziele Euklidesa zatytułowanym *Elementy* około 300 lat przed naszą erą. Istnieją różne wersje tego algorytmu.

Znajdź największy wspólny dzielnik dwóch podanych liczb naturalnych.

Zadanie 11. Permutacje

Permutacją zbioru n-elementowego nazywamy każdy n-wyrazowy ciąg utworzony ze wszystkich elementów tego zbioru. Wszystkich permutacji zbioru n-elementowego jest $n!$ (silnia). Stosując silnię, otrzymuje się bardzo wielkie liczby, np. $10! = 1 \cdot 2 \cdot 3 \cdot 6 \cdot 7 \cdot 8 \cdot 9 \cdot 10$, a wydawałoby się, że to niedużo.

$1! = 1$

$2! = 1! \cdot 2 = 2$

$3! = 2! \cdot 3 = 6$

$4! = 3! \cdot 4 = 24$

$5! = 4! \cdot 5 = 120$

$6! = 5! \cdot 6 = 720$

$7! = 6! \cdot 7 = 5040$

$8! = 7! \cdot 8 = 40320$

$9! = 8! \cdot 9 = 362880$

$10! = 9! \cdot 10 = 3628800$ i tyle należałoby wydrukować wyników dla zbioru 10-elementowego.

Wyznacz wszystkie permutacje zbioru liczbowego dla podanego n.

Zadanie 12. Podzbiory

Możemy założyć, że zbiór zawiera kolejne liczby. W programie może to być zbiór kolejnych n liczb naturalnych $\{1, 2, 3, \ldots, n\}$. Nie będzie tu jednak ograniczenia wielkości, gdyż wszystkich podzbiorów zbioru n-elementowego jest raptem 2^n.

Wyznacz wszystkie podzbiory zbioru liczbowego dla podanego n.

Zadanie 13. Ciąg Fibonacciego

Ciąg Fibonacciego jest to ciąg liczb naturalnych, w którym każdy następny element powstaje przez dodanie do siebie dwóch elementów go poprzedzających. Ciąg rozpoczynają liczby 1 i 1.

$F_1 = 1$, $F_2 = 1$, $F_3 = 2$, $F_4 = 3$, $F_5 = 5$, $F_6 = 8$, $F_7 = 13$, $F_8 = 21$, $F_9 = 34$, $F_{10} = 55$, $F_{11} = 89$, …

Wyznacz n-ty element ciągu Fibonacciego, stosując wzór rekurencyjny i wzór jawny.

Definicja rekurencyjna: $$F_n \begin{cases} 1 & \text{dla} \quad n = 1,2 \\ F_{n-1} + F_{n-2} & \text{dla} \quad n \geq 3 \end{cases}$$

Wzór jawny: $$F_n = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right]$$ *(wzór Bineta)*

Zadanie 14. Sumy

Zbuduj funkcje obliczające podane sumy dla podanego n.

s1 $1 + 2 + 3 + 4 + 5 + \ldots + n$

s2 $1 + 2 - 3 + 4 - 5 + \ldots \pm n$

s3 $1 + 3 + 5 + 7 + \ldots + (2n - 1)$

s4 $1 \cdot 2 + 2 \cdot 3 + 3 \cdot 4 + \ldots n \cdot (n + 1)$

s5 $$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n}$$

s6 $$1 + \frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \frac{1}{5} + \ldots \pm \frac{1}{n}$$

s7 $$\frac{2}{\sqrt{3}} + \frac{4}{\sqrt{5}} + \frac{6}{\sqrt{7}} + \ldots + \frac{2n}{\sqrt{2n+1}}$$

s8 $1^2 + 2^2 + 3^2 + 4^2 + \ldots + n^2$

Zadanie 15. Podziały

Podziałem liczby naturalnej n nazywamy każde przedstawienie jej w postaci sumy liczb naturalnych.

Wyznacz wszystkie podziały podanej liczby naturalnej.

Przykład dla $n = 7$

---

6 + 1

5 + 1 + 1

4 + 1 + 1 + 1

3 + 1 + 1 + 1 + 1

2 + 1 + 1 + 1 + 1 + 1

1 + 1 + 1 + 1 + 1 + 1 + 1

2 + 2 + 1 + 1 + 1

3 + 2 + 1 + 1

4 + 2 + 1

2 + 2 + 2 + 1

3 + 3 + 1

5 + 2

3 + 2 + 2

4 + 3

Zadanie 16. Rozkłady

Rozkład liczby naturalnej na czynniki pierwsze to zapisanie jej w postaci iloczynu liczb pierwszych.

Wyznacz rozkład podanej liczby naturalnej.

Np. $600 = 2 \cdot 2 \cdot 2 \cdot 3 \cdot 5 \cdot 5$

Zadanie 17. Trójkąt Pascala

Jednym z najbardziej interesujących układów liczbowych jest trójkąt Pascala.

Jest to ciąg liczb, który może być ułożony w trójkąt, gdzie każdy wiersz przedstawia współczynniki liczbowe dwumianu $(a + b)^n$.

$(a + b)^0$ — współczynnik 1,

$(a + b)^1$ — współczynniki to 1 i 1,

$(a + b)^2$ — współczynniki to 1, 2 i 1,

$(a + b)^3$ — współczynniki to 1, 3, 3 i 1

$(a + b)^4$ — współczynniki to 1, 4, 6, 4 i 1 itd.

```
0                    1
1                  1   1
2                1   2   1
3              1   3   3   1
4            1   4   6   4   1
5          1   5   10  10   5   1
```

```
6          1   6   15  20  15   6   1
7        1   7   21  35  35   21  7   1
8      1   8   28  56  70  56   28  8   1
9   1   9  36   84  126 126  84  36  9   1
  . . . . . . . . . . . . . . . . . . . . .
```

Zbudujemy taki trójkąt w postaci:

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10  10   5   1
1   6   15  20  15   6   1
1   7   21  35  35   21  7   1
1   8   28  56  70  56   28  8   1
1   9  36   84  126 126  84  36  9   1
```

co nie umniejsza jego czytelności.

W trójkącie Pascala każdy *i*-ty wyraz w wierszu niebędący jedynką jest sumą wyrazów *i*-tego i (*i* – 1)-tego w wierszu powyżej.

Elementy trójkąta można wyznaczyć, posługując się symbolami Newtona:

$\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Są to współczynniki w rozwinięciu dwumianu Newtona $(a + b)^n$.

Zbuduj trójkąt Pascala dla podanej liczby wierszy.

Zadanie 18. Liczba π

Liczba π, inaczej ludolfina (od imienia holenderskiego matematyka Ludolfa van Ceulena), określa stosunek długości okręgu do długości jego średnicy.

π ≈ 3,14159 26535 89793 23846 26433 83279 50288 41971 69399 37510 58209 74944 59230 78164 06286 20899 86280 34825 34211 70679 82148 08651 32823 06647 09384 46095 50582 23172 53594 08128 48111 74502 84102 70193 85211 05559 64462 29489 54930 38196…

Oblicz przybliżoną wartość liczby π, stosując szereg Leibniza:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \ldots$$

dla kolejnych n składników sumy, gdzie $n \in \{1,2,3,\ldots\}$, z dokładnością do kilku miejsc po przecinku. Dla n = 1 mamy $\pi \approx 4 \cdot 1 = 4$, dla n = 2 otrzymujemy $\pi \approx 4 \cdot (1 - 1/3) \approx 2{,}667$, dla n = 3 $\pi \approx 4 \cdot (1 - 1/3 + 1/5) \approx 3{,}467$ itd.

Zadanie 19. Liczba *e*

Liczba *e* to podstawa logarytmu naturalnego wykorzystywana w wielu dziedzinach matematyki i fizyki. W przybliżeniu wynosi 2,718281828459. Zwana jest też liczbą Eulera lub liczbą Nepera.

Oblicz przybliżoną wartość liczby *e*, stosując szereg

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \ldots$$

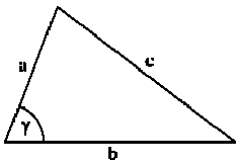
dla kolejnych n+1 składników sumy, gdzie n∈{0, 1, 2, …}, z dokładnością do kilku miejsc po przecinku. Dla n = 0 mamy e ≈ 1/0! =1, dla n = 1 otrzymujemy e ≈ 1/0!+1/1! = 2, dla n = 2 e ≈ 1/0!+1/1!+1/2! = 2,5, zaś dla n = 3 otrzymujemy e ≈ 1/0!+1/1!+1/2!+1/3! ≈ 2,667 itd.

Zadanie 20. Rozwiązywanie trójkąta

Dane są 3 liczby dodatnie. Sprawdź, czy mogą być one długościami boków trójkąta, a jeśli tak, to rozwiąż taki trójkąt. Rozwiązanie trójkąta polega na podaniu wszystkich informacji o nim:

- obwód,
- pole (ze wzoru Herona),
- kąty (z twierdzenia cosinusów),
- rodzaj trójkąta.

Miary kątów można obliczyć, stosując twierdzenie cosinusów.



$$c^2 = a^2 + b^2 - 2ab \cdot \cos\gamma$$

$$2ab \cdot \cos\gamma = a^2 + b^2 - c^2$$

$$\cos\gamma = \frac{a^2 + b^2 - c^2}{2ab}$$

$$\gamma = \arccos\left(\frac{a^2 + b^2 - c^2}{2ab}\right)$$

Wzór Herona: $\text{Pole} = \sqrt{p(p-a)(p-b)(p-c)}$, gdzie p to połowa obwodu trójkąta.

Należy pamiętać, że w językach programowania wszystkie funkcje trygonometryczne operują na miarach kątów wyrażonych w radianach.

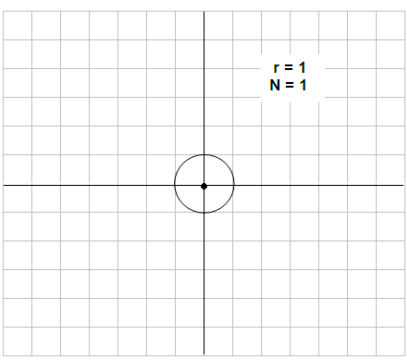
Rodzaj trójkąta rozpoznaj po długościach boków i po miarach kątów.

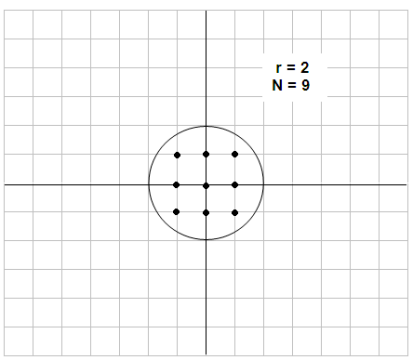
Zadanie 21. Punkty kratowe

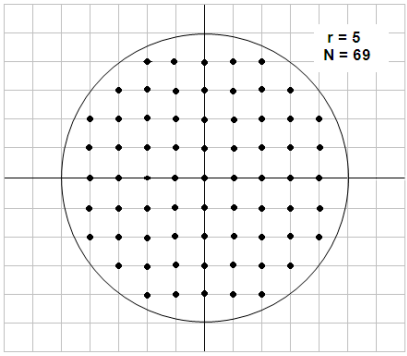
Punkt kratowy to punkt, którego współrzędne w układzie kartezjańskim są liczbami całkowitymi.

Dane jest koło o środku w początku układu współrzędnych i promieniu *r*.

Znajdź wszystkie punkty kratowe leżące **wewnątrz** takiego koła.





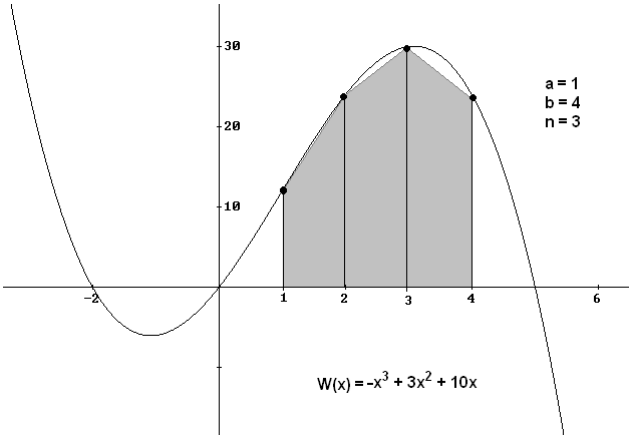


Zadanie 22. Pole figury

Dany jest stopień n oraz współczynniki a_0, a_1, a_2, …a_{n-2}, a_{n-1}, a_n wielomianu $W(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + ... + a_{n-2}x^2 + a_{n-1}x + a_n$.

Oblicz przybliżoną wartość pola figury ograniczonej osią X i wykresem funkcji $W(x)$ w podanym przedziale $\langle a; b\rangle$.

Do obliczenia pola zastosujemy **metodę trapezów**:



Przedział $\langle a; b\rangle$ dzielimy na $n + 1$ równo odległych punktów $x_0, x_1, x_2, \ldots, x_n$.

Odległość między punktami, czyli wysokość każdego trapezu $\Delta x = (b - a)/n$

$x_0 = a, x_1 = a + \Delta x, x_2 = a + 2 \cdot \Delta x, \ldots x_n = a + n \cdot \Delta x = b,$

Zatem dla $i = 0, 1, 2, \ldots, n$ $x_i = a + i \cdot (b - a)/n$

Dla każdego x_i obliczamy wartość funkcji $W(x_i) = W(a + i \cdot (b - a)/n)$

Pole pod wykresem funkcji przybliżane jest polami n trapezów.

trapez 1: $P_d(1)$ = podstawa_dolna = $W(a + 0 \cdot \Delta x)$

$P_g(1)$ = podstawa_górna = $W(a + 1 \cdot \Delta x)$

H = wysokość = Δx

$$\text{Pole} = P(1) = \frac{\left|P_d(1) + P_g(1)\right|}{2} \cdot H$$

trapez 2: $P_d(2)$ = podstawa_dolna = $W(a + 1 \cdot \Delta x)$
$P_g(2)$ = podstawa_górna = $W(a + 2 \cdot \Delta x)$
H = wysokość = Δx
$$\text{Pole} = P(2) = \frac{\left|P_d(2) + P_g(2)\right|}{2} \cdot H$$

trapez 3: $P_d(3)$ = podstawa_dolna = $W(a + 2 \cdot \Delta x)$
$P_g(3)$ = podstawa_górna = $W(a + 3 \cdot \Delta x)$
H = wysokość = Δx
$$\text{Pole} = P(3) = \frac{\left|P_d(3) + P_g(3)\right|}{2} \cdot H$$

trapez i: $P_d(i)$ = podstawa_dolna = $W(a + (i-1) \cdot \Delta x)$
$P_g(i)$ = podstawa_górna = $W(a + \Delta x)$
… H = wysokość = Δx
$$\text{Pole} = P(i) = \frac{\left|P_d(i) + P_g(i)\right|}{2} \cdot H$$

trapez n: $P_d(n)$ = podstawa_dolna = $W(a + (n-1) \cdot \Delta x)$
$P_g(n)$ = podstawa_górna = $W(a + n \cdot \Delta x)$
… H = wysokość = Δx
$$\text{Pole} = P(n) = \frac{\left|P_d(n) + P_g(n)\right|}{2} \cdot H$$

Przybliżona wartość pola figur jest sumą pól wszystkich otrzymanych w ten sposób trapezów:

$$\text{Pole} \approx P(1) + P(2) + \ldots + P(n)$$

Dla przykładu z rysunku otrzymamy wynik = 72.

Poprawność tego wyniku można sprawdzić, obliczając całkę:

$$\int_1^4 (-x^3 + 3x^2 + 10x)dx = \left(-\frac{x^4}{4} + x^3 + 5x^2 \right)_1^4 = \frac{297}{4} = 74{,}25$$

Zadanie 23. Rzut ukośny

Rzut ukośny to ruch ciała, któremu nadano prędkość o wektorze skierowanym pod pewnym kątem do poziomu. Zakładamy, że ruch ten odbywa się bez żadnych oporów, np. oporu powietrza.

Dana jest odległość (d) w metrach.

Oblicz, jaką trzeba nadać prędkość początkową, aby przedmiot osiągnął podaną odległość d w poziomie dla kątów 1°, 2°, 3°, … 88°, 89°.

Wzór na zasięg rzutu ukośnego: $z = \frac{v_0^2 \cdot \sin 2\alpha}{g}$

v_0 to prędkość początkowa, α — kąt rzutu, $g \approx 9{,}81$ m/s^2 — przyśpieszenie ziemskie.

Zadanie 24. Równanie liniowe

Równanie liniowe z jedną niewiadomą ma postać $ax + b = 0$.

Rozwiąż takie równanie.

Zadanie 25. Równanie kwadratowe

Równanie kwadratowe ma postać $ax^2 + bx + c = 0$.

Rozwiąż takie równanie.

Zadanie 26. Układ równań

Układ trzech równań liniowych z trzema niewiadomymi ma postać

$$\begin{cases} a_1 x + b_1 y + c_1 z = d_1 \\ a_2 x + b_2 y + c_2 z = d_2 \\ a_3 x + b_3 y + c_3 z = d_3 \end{cases}$$

Rozwiąż taki układ.

Zadanie 27. Liczba wyrazów w zdaniu

Dane jest zdanie, np. „Ala ma kota.”.

Oblicz liczbę wyrazów w podanym zdaniu.

Zadanie 28. Palindromy

Palindrom to wyrażenie, które czytane od końca (wspak) brzmi tak samo jak wyjściowe.

Np. „A to kanapa pana Kota”.

Sprawdź, czy podane zdanie jest palindromem.

Zadanie 29. Szyfr Cezara

Szyfrowanie tekstów metodą Cezara. Zaszyfruj i odszyfruj podany tekst, stosując metodę Cezara.

W szyfrowaniu metodą Cezara każdemu znakowi tekstu jawnego odpowiada znak przesunięty o określoną ilość znaków w alfabecie. Przyjmujemy, że jest to alfabet łaciński, który zawiera 26 liter: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z. W metodzie tej nie rozróżnia się liter wielkich i małych, tak więc tekst, który będziemy podawać, będzie zamieniany na litery wielkie. Przesuwając o 3 znaki: litera A staje się literą D, litera B staje się literą E, a litera Z literą C.

Jest to bardzo prymitywny szyfr, który można w prosty sposób złamać, więc nie gwarantuje żadnego bezpieczeństwa.

Zadanie 30. Szyfr XOR

Szyfrowanie tekstów metodą XOR. Zaszyfruj i odszyfruj podany tekst, stosując metodę XOR.

Operacja logiczna XOR:

0 XOR 0 = 0

0 XOR 1 = 1

1 XOR 0 = 1

1 XOR 1 = 0

Szyfrowanie polega na zastosowaniu operacji XOR na kodach ASCII znaków wejściowych i klucza (hasła) będącego jednym znakiem. Odszyfrowanie następuje w ten sam sposób, tym samym algorytmem, z zastosowaniem takiego samego klucza. Dokładniej jest to wyjaśnione w rozwiązaniach zadania.

Rozdział 2.
Rozwiązania

Typ `boolean` (`bool`) to typ logiczny. Zmienna `x` takiego typu może posiadać wartość `True` (`1`) albo `False` (`0`). Pytamy o nią wtedy w ten sposób:

`if x then …;`	Pascal
`if(x) …;`	C++, JS
`if x:`	Python

Funkcję o nazwie `f` typu logicznego możemy wywołać podobnie:

`if (`*parametry*`) then …;`	Pascal
`if (f(`*parametry*`)) …;`	C++, JS
`if f(`*parametry*`):`	Python

a zwracając jej wartość, napiszemy odpowiednio:

`f :=` *wyrażenie_logiczne*`;`	Pascal
`return` *wyrażenie_logiczne*`;`	C++, JS
`return` *wyrażenie_logiczne*	Python

```
f := wyrażenie_logiczne;
return wyrażenie_logiczne;
return wyrażenie_logiczne
```

Polecamy stosowanie typu logicznego w rozwiązaniach zadań, szczególnie jako wyników zwracanych przez funkcje.

Operacje, jakie można zastosować na wyrażeniach logicznych, to:

		Pascal	C++, JS	Python
1.	negacja	`not`	`!`	`not`
2.	koniunkcja	`and`	`&&`	`and`
3.	alternatywa	`or`	`||`	`or`
4.	alternatywa rozłączna	`xor`	`^`	`^`

Zadanie 1. Czy podana liczba naturalna jest parzysta?

Liczba jest parzysta, jeżeli dzieli się przez 2. Innym sposobem rozpoznania jest sprawdzenie ostatniej jej cyfry — musi to być 0, 2, 4, 6 albo 8. Jeżeli podana liczba jest większa od 1, to możemy odejmować od niej liczbę 2 tak długo, aż uzyskamy 0 albo 1 i w ten sposób rozstrzygniemy o parzystości, co jednak nie jest efektywne. Operator `mod` (Pascal) lub `%` (C++, JS, Python) wyznacza resztę z dzielenia dwóch liczb całkowitych.

pas

```
function z1(n: integer): boolean;
begin
   z1 := n mod 2 = 0;
end;
```

cpp

```
bool z1(int n)
{
   return n % 2 == 0;
}
```

js

```
function z1(n)
{
   return n % 2 == 0;
}
```

py

```
def z1(n):
   return n % 2 == 0
```

Możemy też sprawdzić, czy ostatnia cyfra w zapisie bitowym liczby to `0`.

```
def z1(n):
   return n & 1 == 0
```

Zadanie 2. Czy podana liczba naturalna jest pierwsza?

Liczba n jest pierwsza, jeśli posiada dokładnie dwa dzielniki, tzn. nie dzieli się przez liczby z zakresu od 2 do $n - 1$. Wystarczy sprawdzić, czy istnieją dzielniki podanej liczby n w przedziale $\left[2;\left\lfloor\sqrt{n}\right\rfloor\right]$, gdyż wyczerpuje to matematycznie wszystkie możliwości. Symbol $\lfloor\ \rfloor$ oznacza zaokrąglenie dolne.

pas

```
function z2(n: integer): boolean;
var i: integer;
begin
   if n = 1 then z2 := false else z2 := true;
   for i := 2 to round(sqrt(n)) do
      if n mod i = 0 then begin z2 := false; break; end;
end;
```

cpp

```
bool z2(int n)
{
   int i;
```

```
   if(n < 2) return false;
   for(i = 2; i * i <= n; i++) if(n % i == 0) return false;
   return true;
}
```

js

```
function z2(n)
{
   var i;
   if (n < 2) return false;
   for(i = 2; i * i <= n; i++) if(n % i == 0) return false;
   return true;
}
```

py

```
def z2(n):
   if x < 2:
      return False
   for i in xrange(2, int(n ** .5) + 1):
      if n % i == 0:
         return False
   return True
```

Zadanie 3. Czy podana liczba naturalna jest super-pierwsza?

Liczba jest super-pierwsza, jeśli jest liczbą pierwszą oraz suma jej cyfr też jest liczbą pierwszą.

Takimi liczbami są np. 11, 61, 101, 1013.

W celu rozwiązania zadania zbudujemy funkcję obliczającą sumę cyfr podanej liczby.

Funkcja `digits` wyznacza ostatnią cyfrę danej liczby, następnie ją od niej odejmuje i tak otrzymaną wartość dzieli przez 10, dopóki wartość ta jest większa od zera.

Np. dla $n = 827$ mielibyśmy ciąg zdarzeń: cyfra = 7, $n = 827 - 7 = 820$, $n = 820 : 10 = 82$, cyfra = 2, $n = 82 - 2 = 80$, $n = 80 : 10 = 8$, cyfra = 8, $n = 8 - 8 = 0$, $n = 0 : 10 = 0$ STOP

Wszystkie tak otrzymane cyfry są sumowane.

pas

```
{ suma cyfr }
function digits(n: integer): integer;
var suma, c: integer;
begin
   suma := 0;
   while n > 0 do
   begin
      c := n mod 10; suma := suma + c; n := n - c; n := n div 10;
   end;
digits := suma;
end;

{ sprawdzanie, czy liczba jest super-pierwsza }
function z3(n: integer): boolean;
begin
   if z2(n) and z2(digits(n))
```

```
   then z3 :=true else z3 := false;
end;
```

cpp

```
// suma cyfr
int digits(int n)
{
   int suma = 0, c;
   while(n > 0)
   {
      c = n % 10; suma = suma + c; n = n - c; n = n / 10;
   }
   return suma;
}

// sprawdzanie, czy liczba jest super-pierwsza
bool z3(int n)
{
   return z2(n) && z2(digits(n));
}
```

js

```
// suma cyfr
function digits(n)
{
   var suma = 0;
   var c;
   while(n > 0)
   {
      c = n % 10; suma =  suma +  c; n = n - c;  n = n / 10;
   }
   return suma;
}

// sprawdzanie, czy liczba jest super-pierwsza
function z3(n)
{
   return z2(n) && z2(digits(n));
}
```

py

```
# obliczanie sumy cyfr
def digits(s):
   l = 0
   while s:
      s, r = s / 10, s % 10
      l += r
   return l

# sprawdzanie, czy liczba jest super-pierwsza
def z3(s):
   return z2(s) and z2( digits (s))
```

Zadanie 4. Czy podana liczba naturalna jest B-super-pierwsza?

Liczba jest B-super-pierwsza, jeśli jest liczbą super-pierwszą oraz suma dziesiętna jej cyfr w rozwinięciu dwójkowym jest też liczbą pierwszą.

Takimi liczbami są np. 47, 61, 1033, 5059.

W celu rozwiązania tego zadania wykorzystamy funkcje z zadania 3. o liczbie super-pierwszej. Zbudujemy też nową funkcję `binary`, która obliczy sumę cyfr w rozwinięciu binarnym.

Funkcja `binary` wyznacza kolejne cyfry liczby w systemie dwójkowym (binarnym) od końca, uzyskując je jako reszty z dzielenia danej liczby przez 2. Cyfry te są sumowane. Resztą jest zawsze liczba 0 albo 1. Następnie liczba jest dzielona całkowicie przez 2 i czynności te są powtarzane, dopóki ciągle zmieniająca się jej wartość jest większa od 0.

Np. dla $n = 13$ mielibyśmy ciąg zdarzeń: reszta = 1, $n = 6$, reszta = 0, $n = 3$, reszta = 1, $n = 1$, reszta = 1; $n = 0$ STOP. Zatem mamy cyfry 1, 0, 1, 1 bo $13_{(10)} = 1101_{(2}$, a suma tych cyfr to 3 (kolejność wyznaczania cyfr jest nieważna). Natomiast liczba 13 nie jest B-super-pierwsza, bo nie jest super-pierwsza!

pas

```
function binary(n: integer): integer;
var s, r: integer;
begin
   s := 0;
   while n > 0 do begin r := n mod 2; s := s + r; n := n div 2; end;
   binary := s;
end;

{ sprawdzanie, czy liczba jest B-super-pierwsza }
function z4(n: integer): boolean;
begin
   if z2(n) and z2(digits(n)) and z2(binary(n))
   then z4 := true else z4 := false;
end;
```

cpp

```
int binary(int n)
{
   int s = 0, r;
   while(n > 0) { r = n % 2; s = s + r; n = n / 2;}
   return s;
}

// sprawdzanie, czy liczba jest B-super-pierwsza
bool z4(int n)
{
   return z2(n) && z2(digits(n)) && z2(binary(n));
}
```

js

Zastosowano funkcję `Math.floor(n / 2)`, gdyż w JS nie ma dzielenia całkowitego, a zwykły operator dzielenia `/` dawałby wynik z ułamkami. `floor` wyznacza największą liczbę całkowitą, która jest mniejsza od podanej liczby lub jej równa.

```
function binary(n)
{
   var s = 0;
   var r;
   while(n > 0)
   {
```

```
            r = n % 2;
            s = s + r;
            n = Math.floor(n / 2);
         }
         return s;
      }

      // sprawdzanie, czy liczba jest B-super-pierwsza
      function z4(n)
      {
         return z2(n) && z2(digits(n)) && z2(binary(n));
      }
```

py

```
def binary(s):
   l = 0
   while s:
      s, r = s / 2, s % 2
      l += r
   return l

def z3(s):
   return z2(s) and z2(digits (s)) and z2(binary (s))
```

Zadanie 5. Czy podana liczba naturalna jest doskonała?

Liczba doskonała — liczba naturalna, która jest sumą wszystkich swych dzielników właściwych (to znaczy od niej mniejszych).

Najmniejszą liczbą doskonałą jest 6, ponieważ 6 = 3 + 2 + 1. Następna to 28 (28 = 14 + 7 + 4 + 2 + 1), a kolejne to 496, 8128, 33550336, 8589869056, 137438691328.

Napiszemy funkcję `dividers`, która oblicza sumę wszystkich dzielników właściwych podanej liczby. Jeżeli już potrafimy znaleźć listę dzielników właściwych danej liczby, sprawdzenie, czy jest ona doskonała, jest bardzo proste.

pas

```
function dividers(n: integer): integer;
var k, s: integer;
begin
   s := 0;
   for k := 1 to n - 1 do  if n mod k = 0 then s := s + k;
   dividers := s;
end;

function z5(n: integer): boolean;
begin
   z5 := n = dividers(n);
end;
```

cpp

```
int dividers(int n)
{
   int k, s = 0;
   for(k = 1; k < n; k++) if (n % k == 0) s = s + k;
   return s;
```

```
}

bool z5(int n)
{
   return n == dividers(n);
}
```

js

```
function dividers(n)
{
   var k;
   var s = 0;
   for (k = 1; k < n; k++) if (n % k == 0) s = s + k;
   return s;
}

function z5(n)
{
   return n == dividers(n);
}
```

py

Do kolejnych zadań potrzebujemy funkcji zwracającej sumę wszystkich dzielników właściwych danej liczby. Dzielniki tworzą pary, które po wymnożeniu dają wyjściową liczbę: $n = pq$. Ustalmy, że $p \le q$. Zawsze $p \le \sqrt{n} \le q$, dlatego wystarczy przeszukać do $\lfloor \sqrt{n} \rfloor$ — liczby q uzyskamy z liczb p poprzez dzielenie. Musimy zabezpieczyć przypadek szczególny, gdy liczba jest kwadratem i jeden dzielnik się powtarza. Ostatnią wartość sprawdzamy oddzielnie. Jeżeli jej kwadratem jest n, to dodajemy ją do sumy raz, jeżeli mimo to `x % px == 0`, to dodajemy ostatnią parę dzielników. Na końcu dodajemy dzielnik oczywisty, czyli 1. Liczby n nie dodajemy, ponieważ nie jest dzielnikiem właściwym.

```
def divisors(x):
   l = 0
   px = int(x ** .5)
   for i in xrange(2, px):
      if x % i == 0:
         l += i + x / i
   if px * px == x:
      l += px
   elif x % px == 0:
      l += px + x / px
   return l + 1
```

Jeżeli już potrafimy znaleźć sumę dzielników właściwych danej liczby, to sprawdzenie, czy jest ona doskonała, jest bardzo proste:

```
def z5(x):
   return x == divisors(x)
```

Zadanie 6. Czy podana liczba naturalna jest podzielna?

Liczba podzielna to liczba naturalna, która jest większa od 0 i dzieli się przez sumę swoich cyfr.

Takimi liczbami są np. 21, 36, 126, 200.

Wykorzystamy funkcję `digits` z zadania 3. Napiszemy też funkcję, która sprawdzi, czy podana liczba dzieli się przez sumę swoich cyfr.

pas

```
function z6(n: integer): boolean;
begin
   z6 := n mod digits(n) = 0;
end;
```

cpp

```
bool z6(int n)
{
   return n % digits(n) == 0;
}
```

js

```
function z6(n)
{
   return n % digits(n) == 0;
}
```

py

```
def z6(x):
   return x % digits(x) == 0
```

Zadanie 7. Czy podana liczba naturalna jest liczbą Mersenne'a?

Liczby Mersenne'a to liczby postaci $2^p - 1$, gdzie p jest liczbą pierwszą.

Napiszemy funkcję, która będzie sprawdzać, czy podana liczba ma wymaganą postać. Utworzymy też funkcję `power` obliczającą potęgę liczby 2 o wykładniku naturalnym. Wstawiając zamiast podstawy 2 inną liczbę, otrzymamy łatwo funkcję obliczającą potęgę tej liczby.

Funkcja `z7` wykorzystuje funkcję `power`. Pętla wyznacza potęgi liczby 2 zmniejszone o 1, dopóki są one mniejsze od podanej liczby. Następnie sprawdzane jest, czy podana liczba jest odpowiedniej postaci i ostatnio przypisany jej wykładnik jest liczbą pierwszą (funkcja `z2` z zadania 2.). Od zmiennej `k` odjęta była na końcu liczba 1, gdyż pętla zwiększała wartość tej zmiennej zawsze po obliczeniu potęgi.

pas

```
{ obliczanie potęgi liczby 2 }
function power(w: integer): integer;
var k, pot: integer;
begin
   pot := 1;
   if w > 0 then for k := 1 to w do pot := pot * 2;
   power := pot;
end;

{ sprawdzanie, czy podana liczba jest liczbą Mersenne'a }
function z7(n: integer): boolean;
var x, k: integer;
begin
   k := 1;
   repeat
      x := power(k) - 1; inc(k);
```

```
   until x >= n;
   if (x = n) and (z2(k - 1)) then z7:= true else z7 := false;
end;
```

cpp

```
// obliczanie potęgi liczby 2 (wynik jest liczbą naturalną)
   int power(int w)
   {
      int k, pot = 1;
      if(w > 0) for(k = 1; k <= w; k++) pot = pot * 2;
      return pot;
   }

// sprawdzanie, czy podana liczba jest liczbą Mersenne'a
bool z7(int n)
{
   int x, k = 1;
   do
      {x = power(k) - 1; k++;}
   while(x < n);
   return x == n && z2(k - 1);
}
```

js

```
// obliczanie potęgi liczby 2 (wynik jest liczbą naturalną)
   function power(w)
   {
      var k;
      var pot = 1;
      if(w > 0) for(k = 1; k <= w; k++) pot = pot * 2;
      return pot;
   }

// sprawdzanie, czy podana liczba jest liczbą Mersenne'a
function z7(n)
{
   var x;
   var k = 1;
   do
      {x = power(k) - 1; k++;}
   while(x<n);
   return x == n && z2(k - 1);
}
```

py

Na początek wyznaczymy najmniejsze p, takie że 2^p jest większe od podanej liczby. Potem sprawdzamy, czy liczba jest równa $2^p - 1$.

```
def z7(x):
   l = 1
   while x >= l:
      l *= 2
   return (x == l - 1)
```

Zauważmy, że taka liczba ma w zapisie binarnym same jedynki (podobnie jak liczba $10^p - 1$ ma w zapisie dziesiętnym dokładnie p dziewiątek). Generujemy kolejne cyfry podobnie jak w funkcji `binary`, ale nie potrzebujemy ich do siebie

dodawać, a jedynie sprawdzamy ich wartości. Jeżeli którakolwiek z nich będzie 0, przerywamy działanie funkcji i zwracamy `False`. Jeżeli wygenerujemy wszystkie cyfry i będą one jedynkami, zwracamy `True`.

```
def z7(x):
   while x:
      if x % 2 == 0:
         return False
      x = x / 2
   return True
```

Zamiast dzielenia całkowitoliczbowego i dzielenia modulo można użyć operacji binarnych.

```
def z7(x):
   while x:
      if x & 1 == 0:
         return False
      x = x >> 1
   return True
```

Drugie rozwiązanie jest szybsze od pierwszego, najszybsze jest trzecie rozwiązanie.

Zadanie 8. Czy dane dwie liczby naturalne są bliźniacze?

Liczby bliźniacze to takie dwie liczby pierwsze, których różnica wynosi 2.

Przykłady liczb bliźniaczych: 3 i 5, 5 i 7, 11 i 13.

Rozwiązanie zadania pozostawiamy czytelnikowi. Wystarczy sprawdzić, czy podane liczby są pierwsze (patrz zadanie 2.) i w przypadku obu pozytywnych odpowiedzi sprawdzić jeszcze, czy różnią się o 2.

Można też zrobić na odwrót. Najpierw sprawdzić, czy różnią się o 2, i ewentualnie od razu wyeliminować sprawdzanie, czy są pierwsze.

Zadanie 9. Czy dane dwie liczby naturalne są zaprzyjaźnione?

Liczby zaprzyjaźnione to para różnych liczb naturalnych, takich że suma dzielników właściwych każdej z tych liczb równa się drugiej. Dzielniki właściwe są to wszystkie dzielniki danej liczby oprócz niej samej.

Przykłady liczb zaprzyjaźnionych: 220 i 284, 1184 i 1210, 2620 i 2924, 5020 i 5564.

W rozwiązaniu zadania wykorzystamy funkcję `dividers` z zadania 5., która oblicza sumę dzielników właściwych danej liczby.

pas

```
function z9(a, b: integer): boolean;
begin
   if(dividers(a) = b) and (dividers(b) = a) then z9 := true
   else z9 := false;
end;
```

cpp

```
bool z9(int a, int b)
{
   return dividers(a) == b && dividers(b) == a;
}
```

js

```
function z9(a, b)
{
   return dividers(a) == b && dividers(b) == a;
}
```

py

Jeżeli dysponujemy funkcją zwracającą dla danej liczby listę jej dzielników właściwych (z rozwiązania zadania 5.), to sprawdzenie, czy liczby są zaprzyjaźnione, jest już łatwe, zgodnie z definicją liczb zaprzyjaźnionych.

```
def z9(x, y):
   return x == dividers(y) and y == dividers(x)
```

Zadanie 10. Największy wspólny dzielnik dwóch liczb. Algorytm Euklidesa.

Największy wspólny dzielnik znajdujemy rekurencyjnie. Wykorzystujemy tu dwa fakty:

1. Największy wspólny dzielnik liczby i zera to ta liczba. Jeżeli jeden z argumentów to zero, drugi jest poszukiwanym wynikiem.
2. Niech $a < b$. Największy wspólny dzielnik liczb a i b jest równy największemu wspólnemu dzielnikowi liczb (`b mod a`) i `a`.

Jeżeli mniejsza z liczb nie jest zerem, wykorzystujemy punkt drugi i wywołujemy funkcję na parze mniejszych liczb. Rekurencja skończy się w skończonej liczbie kroków, gdy mniejszy argument osiągnie wartość `0`.

pas

```
function z10(a, b: integer): integer;
var c: integer;
begin
   if a > b then z10 := z10(b, a)
   {na wypadek, gdyby ktoś podał jako pierwszą większą liczbę}
   else
      if a = 0 then z10 := b
      else z10 := z10(b mod a, a);
end;
```

cpp

```
int z10(int a, int b)
{
   if (a > b) return z10(b, a);
   // na wypadek, gdyby ktoś podał jako pierwszą większą liczbę
   else
      if (a == 0) return b;
      else return z10(b % a, a);
}
```

js

```
function z10(a, b)
{
   if (a > b) return z10(b, a);
   // na wypadek, gdyby ktoś podał jako pierwszą większą liczbę
   else
      if (a == 0) return b;
      else return z10(b % a, a);
}
```

py

```
def z10(a, b):
    if a > b:                          # na wypadek, gdyby ktoś podał
        a, b = (b, a)                  # większą liczbę jako pierwszy argument
    while a != 0:
        a, b = b % a, a
    return b

def z10(a, b):
    if a > b:                          # na wypadek, gdyby ktoś podał
        a, b = (b, a)                  # większą liczbę jako pierwszy argument
    return (b if a == 0 else z10(b % a, a))
```

Zadanie 11. Dany jest zbiór liczb naturalnych. Napisz program, który wyznaczy wszystkie permutacje tego zbioru.

W rozwiązaniu zadania możemy założyć, że zbiór zawiera kolejne liczby. W programie wystarczy zatem podać jedynie ich liczbę n; zbiór będzie miał wówczas postać $\{1, 2, 3, \ldots, n\}$. Dla innych nieuporządkowanych zbiorów liczbowych czy znakowych należałoby ich elementy tablicować i zastosować algorytm dla indeksów tej tablicy, co na to samo wychodzi.

Permutacją zbioru n-elementowego nazywamy każdy n-wyrazowy ciąg utworzony ze wszystkich elementów tego zbioru. Wszystkich permutacji zbioru n-elementowego jest $n!$ (silnia). Stosując silnię, otrzymuje się bardzo wielkie liczby. Np. $10! = 1 \cdot 2 \cdot 3 \cdot 6 \cdot 7 \cdot 8 \cdot 9 \cdot 10$, a wydawałoby się, że to niedużo.

$1! = 1$

$2! = 1! \cdot 2 = 2$

$3! = 2! \cdot 3 = 6$

$4! = 3! \cdot 4 = 24$

$5! = 4! \cdot 5 = 120$

$6! = 5! \cdot 6 = 720$

$7! = 6! \cdot 7 = 5040$

$8! = 7! \cdot 8 = 40320$

$9! = 8! \cdot 9 = 362880$

$10! = 9! \cdot 10 = 3628800$ i tyle należałoby wydrukować wyników dla zbioru 10-elementowego.

pas

Pokażemy pewien nietypowy sposób rozwiązania zadania dla niewielkiej liczby elementów $n < 10$, stosując zapis liczb w systemie dziesiętnym. Dla większej liczby można zastosować podany poniżej algorytm, używając np. systemu szesnastkowego dla $n < 16$ czy liter alfabetu łacińskiego dla $n < 24$.

Dla zbioru 16-elementowego mielibyśmy $16! = 20922789888000$ wyników, a dla zbioru 24-elementowego $24! = 620448484017332394393600000$ wyników.

Nie jest to metoda efektywna w porównaniu z algorytmem zastosowanym w C++ czy też z rozwiązaniem w języku Python, który daje programiście świetne narzędzia ułatwiające tworzenie kodu.

Niemniej zastosowana metoda jest interesująca.

Algorytm:

Dla $n = 5$ mamy zbiór {1, 2, 3, 4, 5}. Istnieje zatem 5! = 120 permutacji tego zbioru.

Stosując instrukcję powtarzania, przeglądamy wszystkie liczby dziesiętne pięciocyfrowe, zaczynając od liczby 12345, a kończąc na liczbie 54321, czyli 12345, 12346, 12347, 12348 itd. Jako wyniki przyjmujemy tylko te liczby, które zawierają jedynie cyfry 1, 2, 3, 4, 5 i są to cyfry różne. Otrzymujemy w ten sposób wszystkie permutacje.

12345 — przyjmujemy wydruk = (1, 2, 3, 4, 5)
12346 — odrzucamy
12347 — odrzucamy
…
24531 — przyjmujemy wydruk = (2, 4, 5, 3, 1)
24532 — odrzucamy
…
34555 — odrzucamy
…
54132 — przyjmujemy wydruk = (5, 4, 1, 3, 2)
…
54320 — odrzucamy
54321 — przyjmujemy wydruk = (5, 4, 3, 2, 1) itd.

```
{ potęga liczby 10 }
function pot(w: integer): longint;
var i: integer;
   p: longint;
begin
   p := 1;
   if w > 0 then for i := 1 to w do p := p * 10;
   pot := p;
end;

{ wyświetlanie  permutacji }
procedure z11(n: integer);
var x, y, z: longint;
   i, j, k, czy: integer;
   stary, nowy: string;
   zn: char;
begin
   { generowanie liczb x = 1234...n i y = n...321 }
   x := 0; y := 0;
   for k := 1 to n do x := x + k * pot(n - k);
   for k := n downto 1 do y := y + k * pot(k - 1);

   { pętla od x do y z wykorzystaniem zmiennych tekstowych }
   for z := x to y do
   begin
```

```
      str(z, stary);
      czy := 0; nowy := '';

      for i := 1 to n do
      begin
         zn := stary[i];
         for j := 1 to n do
         { sprawdzenie, czy cyfry się powtarzają }
         if(j <>i) and (zn = stary[j]) then czy := 1;
         { odejmujemy 48, gdyż kod (ord) cyfry zero to 48 }
         if(czy = 0) and (ord(zn) - 48 > 0) and (ord(zn) - 48 <= n) then
         ↪nowy := nowy + zn;
      end;
      if length(nowy) = n then writeln(nowy);
   end;
end;
```

cpp

Przedstawiamy algorytm wyznaczania permutacji przez wstawianie elementów do tablicy tak, aby się one nie powtarzały. Użyte tablice zostały zadeklarowane jako zmienne wskaźnikowe. Nazwa tablicy jest wskaźnikiem do pierwszego jej elementu.

Wywołanie funkcji: `z11(t, n, 0);` po uprzednim wyzerowaniu tablicy `t`.

```
int *t;
   t = new int[n];
   for(i = 0; i < n; i++) t[i] = 0;    // zerowanie tablicy
```

Parametr `n` to liczba elementów tablicy. Zmienna pomocnicza `p` jest zmienną globalną; po pierwszym wywołaniu funkcji będzie miała wartość `0`.

```
int p = -1;

// rekurencyjne wyznaczanie permutacji
void z11(int *w, int n, int k)
{
   int i;
   p++; w[k] = p;
   if(p == n)
   {
      if(w != 0) for(i = 0; i < n; i++) cout << w[i] << " ";
      cout << endl;
   }
   else for(i = 0; i < n; i++) if(w[i] == 0) z11(w, n, i);
   p = p - 1; w[k] = 0;
}
```

js

Przeczytaj wyjaśnienie dotyczące zastosowanego algorytmu dla języka Pascal.

```
// potęga liczby 10 (wynik jest liczbą naturalną)
function pot(w)
{
   var i, p = 1;
   if(w > 0) for(i = 1; i <= w; i++) p = p * 10;
   return p;
}

function z11(n)
```

```
{
   var x = 0,y = 0, z,
      i, j, k, czy,
      stary = "", nowy = "", zn = "";

   // generowanie liczb x=1234...n i y=n...321
   for(k = 1; k <= n; k++) x = x + k * pot(n - k);
   for(k = n; k >= 1; k--) y = y + k * pot(k - 1);

   // pętla od x do y z wykorzystaniem zmiennych tekstowych
   for(z = x; z <= y; z++)
   {
      stary = z.toString();
      czy = 0; nowy = "";
      for(i = 0; i < n; i++)
      {
         zn = "";
         zn = stary[i];

         // sprawdzenie, czy cyfry się powtarzają
         for(j = 0; j < n; j++) if(j != i && zn[0] == stary[j]) czy = 1;

         // odejmujemy 48, gdyż kod cyfry zero to 48, kod 1=49 itd.
         if(czy == 0 && zn.charCodeAt() - 48 > 0 && zn.charCodeAt(0) - 48 <= n)
         nowy = nowy + zn;
      }
         // drukowanie permutacji w konsoli przeglądarki
        if(nowy.length == n) console.log(nowy, "\n");
   }
}
```

Każda permutacja zbioru `A` jest sumą (konkatenacją) dwóch ciągów: ciągu jednoelementowego `{x}` oraz pewnej permutacji pozostałych wyrazów ciągu. Prowadzi to do funkcji rekurencyjnej:

```
function z11(A)
{
   if (A.length == 1) {return [A]}
   else
   {
      var wynik = [];
      for(var i = 0; i < A.length; i++)
      {
         var N = z11(A.filter(function(k){return(k != A[i])}));
         var n = N.length ;
         for(var j = 0;j < n; j++)
         wynik.push([A[i]].concat(N[j]));
      }
      return wynik ;
   }
}
```

py

```
def z11(A):
   if len(A) == 1:
      return [[A[0]]]
   else:
      wynik = []
      for i in A:
```

```
            for j in z11(filter(lambda x : x != i, A)):
                wynik.append([i] + j)
        return wynik
```

Możemy skrócić jej składnię, wykorzystując konstrukcję listy składanej i trójelementowej instrukcji warunkowej:

```
def z11(L):
    return [[A[0]]] if len(A) == 1 else [[i] + j for i in A for j in
        z11(filter(lambda x : x != i, A))]
```

Jeżeli zbiór jest jednoelementowy, to wynikiem jest lista zawierająca tylko ten zbiór. W przeciwnym wypadku stosujemy przepis rekurencyjny — generujemy listy postaci: lista jednoelementowa zawierająca `i` + element listy permutacji zbioru wszystkich elementów bez elementu `i`. `(filter(lambda x:x!=i,A)` to zbiór `A` pomniejszony o element `x`.

Zadanie 12. Dany jest zbiór liczb naturalnych. Napisz program, który wyznaczy wszystkie podzbiory tego zbioru.

Tak jak w poprzednim zadaniu możemy założyć, że zbiór zawiera kolejne liczby. W programie wystarczy zatem podać jedynie ich liczbę *n* i zbiór będzie miał wówczas postać {1, 2, 3, …, *n*}. Nie będzie tu jednak ograniczenia wielkości, gdyż wszystkich podzbiorów zbioru *n*-elementowego jest raptem 2^n.

Liczba elementów	Liczba podzbiorów	Podzbiory
1	**2**	**{1}, ∅ (zbiór pusty)**
2	**4**	**{1}, {2}, (1, 2}, ∅, zbiór {1, 2} = {2, 1}**
3	**8**	**{1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}, ∅**

pas

Algorytm:

W celu rozwiązania zadania zastosujemy system dwójkowy.

Dla *n* = 3 mamy zbiór {1, 2, 3}, który umieścimy w tablicy.

Wypisujemy wszystkie liczby binarne, począwszy od 0 aż do 111, czyli od 0 do $2^3 - 1 = 7$ (razem 8).

0 — określa zbiór pusty
1 — podzbiór jednoelementowy = {1} $1 = 2^0$, określa indeks tablicy, tu 0
10 — podzbiór jednoelementowy = {2} 2^1, czyli indeks tablicy to 1,
11 — podzbiór dwuelementowy = {1, 2} $2^0 + 2^1$, wybieramy indeksy 0 i 1 itd.
100 — podzbiór jednoelementowy = {3}
101 — podzbiór dwuelementowy = {1,3}
111 — podzbiór trójelementowy = {1, 2, 3}

```
{ deklaracja tablicy — zmienna globalna }
var tab: array[1..1000] of integer;

{ potęga liczby 2 }
function power(w: integer): integer;
```

```
var k, pot: integer;
begin
   pot := 1;
   if w > 0 then for k := 1 to w do pot := pot * 2;
   power :=p ot;
end;

{ wyznaczanie kolejnych cyfr liczby w zapisie dwójkowym }
function binTostr(n: integer): string;
var
   c: integer;
   z: char;
   l: string;
begin
   l := '';
   while n > 0 do
   begin
      c := n mod 2; z := char( c + 48); l := l + z; n := n div 2;

      { Dodajemy 48, gdyż 48 to kod cyfry 0, a 49 to kod cyfry 1, zaś
      funkcja char zamienia liczbę na znak o podanym kodzie.
      Instrukcja l := l + z powoduje dodanie (dopisanie, konkatenacja)
      znaku do zmiennej tekstowej (string), która na początku ma wartość
      „puste" l =" (między apostrofami nie ma żadnej spacji!). }
   end;
   binTostr := l;
end;

{wyświetlanie  podzbiorów }
procedure z12(n: integer);
var
   i, k, dl, ile: integer;
   s: string;
begin
   for i := 1 to n do tab[i] := i;
   writeLn('Podzbiory: ');
   ile := power(n) - 1;
   for i := 1 to ile do
      begin
      s := binTostr(i);
      dl := length(s);
      write('{');
      for k := 1 to dl - 1 do
      if s[k] = '1' then write(tab[k], ', ');
      if s[dl] = '1' then write(tab[dl]); {po to, żeby na końcu wyświetlanego tekstu
                                           nie było przecinka}
      writeLn('}');
   end;
end;
```

cpp

```
// deklaracja tablicy — zmienna globalna
int tab[1000];

// potęga liczby 2 (wynik jest liczbą naturalną!)
int power(int w)
{
   int k, pot = 1;
```

```
    if(w > 0) for(k = 1; k <= w; k++) pot = pot * 2;
    return pot;
}
// wyznaczanie kolejnych cyfr liczby w zapisie dwójkowym
string binTostr(int n)
{
    int c;
    char z;
    string l = "";
    while(n > 0)
    {
        c = n % 2; z = c + 48; l = l + z; n = n / 2;

        /*
            Dodajemy 48, gdyż 48 to kod cyfry 0, a 49 to kod cyfry 1,
         zaś podstawienie char <- int, u nas jest to z = c + 48,
         zamienia liczbę na znak o podanym kodzie.
         Instrukcja l = l + z powoduje dodanie (dopisanie, konkatenacja)
         znaku (char) do zmiennej tekstowej (string), która na początku
         ma wartość „puste” l = ””
        (między cudzysłowami nie ma żadnej spacji!).
          */
    }
    return l;
}

// wyświetlanie  podzbiorów
void zl2(int n)
{
    int i, k, dl, ile;
    string s;
    for(i = 0; i <= n; i++) tab[i] = i;
    cout << "Podzbiory:" << endl;
    ile = power(n) - 1;
    for(i = 1; i <= ile; i++)
    {
        s = binTostr(i);
        dl = s.length();
        cout << "{";
        for(k = 0; k < dl - 1; k++)
        if(s[k] == '1') cout << tab[k] + 1 << ", ";
        if(s[dl - 1] == '1') cout << tab[dl - 1] + 1;    // po to, żeby na końcu tekstu
                                                         // nie było przecinka
        cout << "}" << endl;
    }
}
```

js

```
var tab = new Array(1000);    // tablica globalna

// potęga liczby 2 (wynik jest liczbą naturalną!)
function power(w)
{
    var k, pot = 1;
    if(w > 0) for(k = 1; k<= w; k++) pot = pot * 2;
    return pot;
}
```

```
// wyznaczanie kolejnych cyfr liczby w zapisie dwójkowym
function binTostr(n)
{
   var c, l = "";
   while(n > 0)
   { c = n % 2; l = l + c.toString(); n = Math.floor(n / 2); }
      return l;
   }
}

   //wyświetlanie podzbiorów
function z12(n)
{
   var i, k, dl, ile, x, s = "";
   var uchwyt2 = document.getElementById("dif");

   for(i = 0; i <= n; i++) tab[i] = i;
   uchwyt2.innerHTML = "Podzbiory: <br>";
   ile = power(n) - 1;

   for(i = 1; i <= ile; i++)
   {
      s = binTostr(i);
      dl = s.length;
      uchwyt2.innerHTML = uchwyt2.innerHTML + "{";

      for(k = 0; k < dl - 1; k++)
         if(s[k] == '1')
         {
            x = tab[k] + 1;
            uchwyt2.innerHTML = uchwyt2.innerHTML + x.toString() + ", ";
         }

      if(s[dl - 1] == '1')    // <-- żeby na końcu nie było przecinka
      {
         x = tab[dl - 1] + 1;
         uchwyt2.innerHTML = uchwyt2.innerHTML + x.toString();
      }
      uchwyt2.innerHTML = uchwyt2.innerHTML + "} <br>";
   }
}
```

Załóżmy, że mamy listę `L` podzbiorów pewnego zbioru `A` i dodajemy do niego pewien element `x`. Wtedy `L` będzie dalej listą podzbiorów zbioru $A \cup \{x\}$, ale dojdą jeszcze wszystkie podzbiory powiększone o element `x`. Prowadzi to do funkcji rekurencyjnej:

```
function z12(A)
{
   if(A.length == 0) {return [[]]}
   else
   {
      var N = z12(A.slice(1));
      return N.map(function(k) { return[A[0]].concat(k) }).concat(N);
   }
}
```

py

```
def z12(A):
   if len(A) == 0:
      return [[]]
   else:
      L = z12(A[1:])
      return L + [[A[0]] + l for l in L]
```

Zbiór pusty ma dokładnie jeden podzbiór. Jeżeli nie jest on pusty, to tworzymy listę `L` podzbiorów zbioru `A` bez elementu `A[0]`. Lista podzbiorów zbioru `A` to `L` + lista elementów `L` powiększonych o `A[0]`.

Zadanie 13. Napisz program wyznaczający *n*-ty wyraz ciągu Fibonacciego.

Ciąg Fibonacciego jest to ciąg liczb naturalnych, w którym każdy następny element powstaje przez dodanie do siebie dwóch elementów go poprzedzających. Ciąg rozpoczynają liczby 1 i 1.

$F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, F_6 = 8, F_7 = 13, F_8 = 21, F_9 = 34, F_{10} = 55, F_{11} = 89, \ldots$

pas

Funkcja rekurencyjna:

```
function z13(n: integer): longint;
begin
   if (n = 1) or (n = 2) then z13 :=1 else z13 := z13(n - 1) + z13(n - 2);
end;
```

Dla $n = 45$ po 3 minutach pracy komputer (1,67 GHz, 3,23 GB RAM) wygenerował wynik = 1134903170.

Tak długi czas obliczeń wynika głównie z zastosowania tu rekurencji. Dla $n = 50$ program po pewnym czasie przerwał pracę z powodu przepełnienia z podaniem kodu błędu systemowego.

Rozwiązanie z wykorzystaniem wzoru jawnego:

```
{ potęga o podstawie rzeczywistej i wykładniku naturalnym }
function pown(x: double; w: integer): double;
var k: integer;
   pot: double;
begin
   pot := 1;
   if w > 0 then for k := 1 to w do  pot := pot * x;
   pown := pot;
end;

{ obliczanie wartości elementu ciągu ze wzoru }
function z13_2(n: integer): double;
var p, q, r, s: double;
begin
   p := sqrt(5); q := 1 + p; r := 1 - p;  s := 1 / p;
   z13_2 := s * (pown(q / 2, n) - pown(r / 2, n));
end;
```

Tym sposobem uzyskujemy wyniki dla $n > 100$, ale już przybliżone, i dla $n > 173$ mimo zastosowania konwersji na tekst `str(Fibo2(n):0:0,s);` już w postaci zmiennoprzecinkowej (notacja naukowa), np. dla $n = 180$ mamy wynik 1.85477076894721E+037.

W celu uzyskania większych i poprawnych wyników dla ciągu Fibonacciego napiszemy funkcję, która będzie wykonywała dodawanie dwóch liczb naturalnych złożonych nawet z setek cyfr sposobem pisemnym, takim jaki poznają uczniowie w szkole podstawowej, a więc 8 + 7 to 5 i 1 w pamięci.

Wynik dodawania liczb jednocyfrowych może być co najwyżej dwucyfrowy i nigdy nie będzie większy niż 18.

Algorytm:

1. Podajemy dwie liczby jako teksty.
2. Obliczamy długości tych tekstów.
3. Wyznaczamy maksimum z tych długości.
4. Jeżeli oba teksty mają taką samą długość, to przechodzimy do kroku 6.
5. Uzupełniamy krótszy z tekstów znakami '0' z przodu, tak aby teksty wyrównały się długościami. Na przykład: jeśli *a* = '12567', *b* = '78', to tekst *a* pozostaje bez zmiany, natomiast *b* = '00078'.
6. Przeglądamy teksty od końca i dodajemy ich kolejne znaki po konwersji na cyfry, pamiętając, że w przypadku otrzymania wyniku dwucyfrowego pierwsza jego cyfra, czyli 1, wędruje do pamięci.
7. Wynik umieszczamy w zmiennej tekstowej, dopisując do niej kolejne cyfry jako znaki.

Przykłady:

	0	***0***	***1***	***1***	***0***	***1***	***0***	← wyróżnione cyfry to pamięć
	1	5	0	4	8	0	4	
+	***0***	***0***	***0***	5	6	6	8	← wyróżnione cyfry to uzupełnienie
	1	5	1	0	4	7	2	

1	***1***	***0***	***1***	***1***	***0***	***1***	***0***	← wyróżniona cyfra 1 to nadmiar, należy to przewidzieć!
	8	5	0	6	8	0	4	
+	4	9	5	5	6	6	8	
1	3	4	6	2	4	7	2	

Zmienne użyte w programie:

`cyfra1`, `cyfra2`, `cyfra_pam` — cyfry obu liczb i cyfra „w pamięci”
`pam` — pamięć typu znakowego
`wynik` — zmienna tekstowa na wynik
`s` — zmienna tekstowa pomocnicza
`da`, `db`, `dmax` — długości tekstów
`k`, `j` — zmienne iteracyjne pętli
`l1`, `l2` — dwie liczby do dodania (w programie głównym)

```
function add(a, b: string): string;
var
   cyfra1, cyfra2, cyfra_pam, liczba, blad: integer;
   pam: char;
   wynik, wyn, s: string;
   da, db, dmax: longint;
   k, j: integer;
begin
   da := length(a);
   db := length(b);
   if da > db then dmax := da else dmax := db;
   if da <> db then
   begin
      s := '';
      if da > db then
      begin
         for j := 1 to da - db do s := s + '0';
         for j := 1 to db do s := s + b[j];
         b := s;
      end
      else
      begin
         for j := 1 to db - da do s := s + '0';
         for j := 1 to da do s := s + a[j];
         a := s;
      end;
   end;
   wynik := ''; pam := '0';
   for k := dmax downto 1 do
   begin
      val(a[k], cyfra1, blad);
      val(b[k], cyfra2, blad);
      val(pam, cyfra_pam, blad);
      liczba := cyfra1 + cyfra2 + cyfra_pam;
      str(liczba, s);
      if length(s) = 1 then
         begin
            wynik := wynik + s[1]; pam := '0';
         end
       else
         begin
            wynik := wynik + s[2]; pam := s[1];
         end;
   end;
   if pam = '1' then wynik := wynik + pam;
   wyn := '';
   for k := length(wynik) downto 1 do wyn := wyn + wynik[k];
   add := wyn;
end;

{-----------------------------------------}
function z13_3(n: integer): string;
var k: integer;
   f1, f2, f3: string;
begin
   f1 := '1'; f2 := '1';
   if n > 2 then
```

```
   for k := 3 to n do
   begin
      f3 := add(f1, f2);
      f1 := f2; f2 := f3;
   end;
   if n < 3 then z13_3 := '1' else z13_3 := f3;
end;
```

cpp

Funkcja rekurencyjna:

```
long long z13(int n)
   {
      if (n == 1 || n == 2) return 1; else return z13(n - 1) + z13(n - 2);
   }
```

Rozwiązanie przy użyciu wzoru jawnego:

```
double z13_2(int n)
{
   double w ;
   w = (1 / sqrt(5)) * pow(((1 + sqrt(5)) / 2), n) - (1 / sqrt(5)) * pow(((1 -
sqrt(5)) / 2), n);
   return w;
}
```

Algorytm z zastosowaniem dodawania liczb sposobem pisemnym (wyjaśnienie w rozwiązaniu w Pascalu):

```
string add(string a, string b)
{
   int cyfra1, cyfra2, cyfra_pam, liczba, blad;
   int c1, c2;
   char pam, zn;
   string wynik, wyn,s;
   int da, db, dmax, k, j;
   da = a.length();
   db = b.length();
   if da > db) dmax = da; else dmax = db;
   if(da != db)
   {
      s = "";
      if(da > db)
      {
         for(j = 0; j < da - db; j++) s = s + '0';
         for(j = 0; j < db; j++) s = s + b[j];
         b = s;
      }
      else
      {
         for(j = 0; j < db - da; j++) s = s + '0';
         for(j = 0; j < da; j++) s = s + a[j];
         a = s;
      }
   }
   wynik = ""; cyfra_pam = 0;
   for(k = dmax - 1; k >= 0; k--)
   {
      cyfra1 = a[k] - 48;
      cyfra2 = b[k] - 48;
```

```
            liczba = cyfra1 + cyfra2 + cyfra_pam;
            if(liczba < 10)
            {
               wynik = wynik + (char)(liczba + 48);
               cyfra_pam = 0;
            }
            else
            {
               c2 = liczba % 10; c1 = (liczba - c2) / 10;
               wynik = wynik + (char)(c2 + 48);
               cyfra_pam = 1;
            }
         }

         if(cyfra_pam == 1) wynik = wynik + '1';
         wyn = "";
         for(k = wynik.length() - 1; k >=0 ; k--) wyn = wyn + wynik[k];
         return wyn;
      }
      //----------------------------------------
      string z13_3(int n)
      {
         int k, j;
         string f1, f2, f3;

         f1 = "1"; f2 = "1";
         if(n > 2)
         for(k = 3; k <= n; k++)
         {
            f3 = add(f1, f2);
            f1 = f2;
            f2 = f3;
            }
         if(n < 3) return "1"; else return f3;
      }
```

js

Funkcja rekurencyjna:

```
function z13(n)
{
   if (n == 1 || n == 2) return 1; else return z13(n - 1) + z13(n - 2);
}
```

Rozwiązanie z użyciem wzoru jawnego:

```
// potęga o wykładniku naturalnym
function pown(x, w)
{
   var k;
   var pot = 1;
   if(w > 0) for(k = 1; k <=w ; k++) pot = pot * x;
   return pot;
}
// obliczanie wartości elementu ciągu ze wzoru
function z13_2(n)
{
   var p, q, r, s;
```

```
    p = Math.sqrt(5);
    q = 1 + p;
    r = 1 - p;
    s = 1 / p;
    return Math.floor(s * (pown(q / 2, n) - pown(r / 2, n)));
}
```

Przy wywołaniu funkcji `z13_2` należy zastosować zaokrąglanie wyniku do części całkowitych, gdyż dla wyższych wartości występuje niekorzystny nadmiar wynikający z przybliżeń liczb rzeczywistych.

Przykład:

```
document.getElementById("id_div-a").innerHTML=Math.floor(Fibo2(a));
```

py

n-ty wyraz ciągu Fibonacciego będziemy konstruowali na podstawie jego wyrazu $(n-1)$ i $(n-2)$.

```
def z13(n):
    if n < 2:
        return n
    a, b =(1, 0)
    for y in xrange(2, n+1):
        a , b = a + b, a
    return a
```

Możemy to rozwiązać również za pomocą wzoru jawnego:

```
def z13(n):
    return (((1 + 5 ** .5) / 2) ** n - ((1 - 5 ** .5) / 2) ** n) / 5 ** .5
```

Wzór jawny pozwala na obliczenie wyrazów ciągu do $n = 1474$, powyżej tej wartości otrzymujemy błąd przepełnienia. Spróbujmy nie potęgować liczb wymiernych, ale liczby postaci $\frac{a+b\sqrt{5}}{2}$.

Ich mnożeniem rządzi następująca reguła:

$$(a,b)\cdot(c,d)=\left(\frac{1}{2}a+\frac{\sqrt{5}}{2}b\right)\cdot\left(\frac{1}{2}c+\frac{\sqrt{5}}{2}d\right)=\frac{1}{2}\frac{ac+5bd}{2}+\frac{\sqrt{5}}{2}\frac{ad+bc}{2}=((ac+5bd)/2,(ad+bc)/2)$$

Dla liczby $(a,b)=\frac{a+b\sqrt{5}}{2}$ część a będziemy nazywać jej częścią całkowitą, b jej częścią pierwiastkową, a liczbę $(a,b)^{*}=(a,-b)=\frac{a-b\sqrt{5}}{2}$ jej liczbą sprzężoną. Zauważmy, że $x^{*}y^{*}=(xy)^{*}$, zatem $(x^{*})^{n}=(x^{n})^{*}$. Wzór Bineta wiąże *n*-ty wyraz ciągu Fibonacciego z częścią pierwiastkową liczby $(1, 1)^{n}$.

Implementujemy mnożenie liczb postaci $\frac{a+b\sqrt{5}}{2}$ reprezentowanych przez krotki (a,b):

```
def mn(k1, k2):
    return ((k1[0] * k2[0] + 5 * k1[1] * k2[1]) / 2, (k1[0] * k2[1] + k1[1] *
    ↪k2[0]) / 2)
```

Mając zdefiniowane mnożenie, musimy zaimplementować funkcję obliczającą potęgę naturalną liczby (1, 1). Wykonanie $n - 1$ razy mnożenia nie jest efektywne — można tych mnożeń wykonać zdecydowanie mniej. Dla przykładu policzmy 52. potęgę liczby 5, rozłóżmy przy tym wykładnik 52 na sumę potęg:

$$5^{52} = 5^{b00110100} = 5^{32+16+4} = 5^{32} \cdot 5^{16} \cdot 5^{4}$$

Wszystkie liczby w rozwinięciu dostaniemy przez kolejne podnoszenie do kwadratu liczby 5. Wynik inicjalizujemy za pomocą wartości 1. Przeglądamy bit po bicie od prawej strony wykładnik (używając przesunięcia bitowego) i jeżeli w wykładniku jest 1, mnożymy wynik przez aktualną wartość kwadratu. Następnie podnosimy istniejący kwadrat do kwadratu.

$$5 \to 5^2 \to 5^4 \to 5^8 \to 5^{16} \to 5^{32}$$

$$1 \to 1 \to 5^4 \to 5^4 \to 5^4 \cdot 5^{16} \to 5^4 \cdot 5^{16} \cdot 5^{32}$$

W powyższym przypadku wykonamy mnożenie 8 razy zamiast 51. Liczba n ma nie więcej niż $\lfloor \log_2 n \rfloor + 1$ jedynek w wykładniku. Napiszmy funkcję potęgi dla naszych liczb ze zbioru R:

```
def pp(w):
   kw =(1,1)    # pierwsza potęga podstawy
   wyn =(2,0)    # reprezentacja 1
   while w:
      if w & 1:
         wyn = mn(wyn, kw)
      kw = mn(kw, kw)
      w = w >> 1
   return wyn
```

Pozwala to już obliczyć n-ty wyraz ciągu Fibonacciego jako część pierwiastkową n-tej potęgi liczby (1, 1).

```
def z13(n):
   return pp(n)[1]
```

W pojedynczym przebiegu pętli w pierwszym rozwiązaniu wykonujemy jedno dodawanie — bardzo mało kosztowną operację. W rozwiązaniu trzecim w jednym przebiegu pętli wykonujemy cztery lub osiem mnożeń całkowitych, co jest o wiele bardziej kosztowne, jednak liczba przebiegów pętli rośnie wolniej (jak $\log_2(n)$) niż w pierwszym rozwiązaniu (jak n), dlatego od $n \approx 200$ trzecie rozwiązanie zaczyna wykazywać przewagę, którą powiększa wraz ze wzrostem n (dla $n = 1\,000\,000$ trzecie rozwiązanie działa ok. 57 razy szybciej niż pierwsze). Dla zakresu, w którym drugie rozwiązanie działa, trzecie rozwiązanie jest ponaddwukrotnie wolniejsze niż drugie, nie zapominajmy jednak, że drugie rozwiązanie daje tylko wynik przybliżony.

W Pythonie typ `int` odpowiada typowi `long` w C. Typ `long` jest w Pythonie nieograniczony — zmienna może zajmować dowolnie duży obszar pamięci (ograniczony ilością pamięci RAM komputera). W przypadku gdy wynik operacji na liczbach `int` przekracza zakres typu `int`, wynik jest automatycznie rzutowany na typ `long` (drukując wynik, widzimy literę `L` na końcu zapisu liczby). Komentarz ten dotyczy domyślnej w tej książce wersji Pythona 2.7. W Pythonie 3.0 oba te typy są zunifikowane w jeden typ `int`. W Pythonie nie występuje zatem problem przepełnienia, który jest rozwiązywany w innych językach w tym zadaniu.

Na przykład dla $n = 10\,000\,000$ wartość wyrazu ciągu Fibonacciego ma ponad 2 mln cyfr dziesiętnych i zajmuje w pamięci prawie 1 MB.

Zadanie 14. Zbuduj funkcje obliczające podane sumy dla podanego n.

s1 $1 + 2 + 3 + 4 + 5 + \ldots + n$

s2 $1 + 2 - 3 + 4 - 5 + \ldots \pm n$

s3 $1 + 3 + 5 + 7 + \ldots + (2n - 1)$

s4 $1 \cdot 2 + 2 \cdot 3 + 3 \cdot 4 + \ldots n \cdot (n + 1)$

s5 $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \ldots + \frac{1}{n}$

s6 $1 + \frac{1}{2} - \frac{1}{3} + \frac{1}{4} - \frac{1}{5} + \ldots \pm \frac{1}{n}$

s7 $\frac{2}{3} + \frac{4}{\sqrt{5}} + \frac{6}{\sqrt{7}} + \ldots + \frac{2n}{\sqrt{2n+1}}$

s8 $1^2 + 2^2 + 3^2 + 4^2 + \ldots + n^2$

					wyniki przybliżone			
n	**s1**	**s2**	**s3**	**s4**	**s5**	**s6**	**s7**	**s8**
1	1	1	1	2	1	1	1,1547	1
2	3	3	4	8	1,5	1,5	2,94355	5
5	15	–1	25	70	2,28333	1,21667	10,8931	55
10	55	7	100	440	2,92897	1,35437	30,2105	385
100	5050	52	10000	343400	5,18738	1,31183	943,285	338350
1000	500500	502	1000000	334334000	7,48547	1,30735	29814,7	333833500

pas

```
function s1(n: integer): longint;
var i: integer;
    s: longint;
begin
   s := 0; for i := 1 to n do s := s + i; s1 := s;
end;

function s2(n: integer): longint;
var i: integer;
    s: longint;
begin
   s := 1;
   for i := 2 to n do if i mod 2 = 0 then s := s + i else s := s - i;
   s2 := s;
end;

function s3(n: integer): longint;
var i: integer;
    s: longint;
begin
```

```
   s := 0; for i := 1 to n do s := s + 2 * i - 1; s3 := s;
end;

function s4(n: integer): longint;
var i: integer;
    s: longint;
begin
   s := 0; for i := 1 to n do s := s + i * (i + 1); s4 := s;
end;

function s5(n: integer): double;
var i: integer;
    s: double;
begin
   s:=0; for i := 1 to n do s := s + 1 / i; s5 := s;
end;

function s6(n: integer): double;
var i: integer;
    s: double;
begin
   s := 1;
   for i := 2 to n do if i mod 2 = 0 then s := s + 1 / i else s := s - 1 / i;
   s6 := s;
end;

function s7(n: integer): double;
var i: integer;
    s: double;
begin
   s := 0; for i := 1 to n do s := s + 2 * i / sqrt(2 * i + 1); s7 := s;
end;

function s8(n: integer): longint;
var i: integer;
    s: longint;
begin
   s := 0; for i := 1 to n do s := s + i * i; s8 := s;
end;
```

cpp

```
long int s1(int n)
{
   int i; long int s = 0; for(i = 1; i <= n; i++) s = s + i; return s;
}

long int s2(int n)
{
   int i; long int s = 1;
   for(i = 2; i <= n; i++) if(i % 2 == 0) s = s + i; else s = s - i;
   return s;
}

long int s3(int n)
{
   int i; long int s = 0; for(i = 1; i <= n; i++) s = s + 2 * i - 1; return s;
}
```

```
long int s4(int n)
{
   int i; long int s = 0;
   for(i = 1; i <= n; i++) s = s + i * (i + 1); return s;
}

double s5(int n)
{
   int i; double s = 0;
   for(i = 1; i <= n; i++) s = s + 1.0 / i; return s;
}

double s6(int n)
{
   int i;
   double s = 1;
   for(i = 2; i <= n; i++) if(i % 2 == 0) s = s + 1.0 / i; else s = s - 1.0 / i;
   return s;
}

double s7(int n)
{
   int i;
   double s = 0;
   for(i = 1; i <= n; i++) s = s + 2 * i / sqrt(2 * i + 1); return s;
}

long int s8(int n)
{
   int i;
   long int s = 0;
   for(i = 1; i <= n; i++) s = s + i * i; return s;
}
```

js

```
function s1(n)
{
   var i, s = 0;
   for(i = 1; i <= n; i++) s = s + i; return s;
}

function s2(n)
{
   var i, s = 1;
   for(i = 2; i <= n; i++) if(i % 2 == 0) s = s + i; else s = s - i;
   return s;
}

function s3(n)
{
   var i, s = 0;
   for(i = 1; i <= n; i++) s = s + 2 * i - 1; return s;
}

function s4(n)
{
   var i, s = 0;
```

```
   for(i = 1; i <= n; i++) s = s + i * (i + 1); return s;
}

function s5(n)
{
   var i, s = 0;
   for(i = 1; i <= n; i++) s = s + 1.0 / i; return s;
}

function s6(n)
{
   var i, s = 1;
   for(i = 2; i <= n; i++) if(i % 2 == 0) s = s + 1.0 / i; else s = s - 1.0/i;
   return s;
}

function s7(n)
{
   var i, s = 0;
   for(i = 1; i <= n; i++) s = s + 2 * i / Math.sqrt(2 * i + 1); return s;
}

function s8(n)
{
   var i, s = 0;
   for (i = 1; i <= n; i++) s = s + i * i; return s;
}
```

py

Ciągi uzyskujemy, odwzorowując zakres liczb naturalnych przez pewną funkcję. Naturalnym rozwiązaniem jest zatem instrukcja `map`. Sumę wyrazów ciągu dostajemy, działając na niego instrukcją `sum`:

```
def s1(n):
   return sum(xrange(1,n+1))
```

W przypadku ciągów naprzemiennych zamiast wykonywać potęgę `(-1) ** x` lepiej sprawdzać parzystość funkcji `(1 - 2 * (x % 2))`:

```
def s2(n):
   return sum(map(lambda x : (1 - 2 * (x % 2)) * x, xrange(1,n +1)))+2

def s3(n):
   return sum(map(lambda x : 2 * x - 1, xrange(1,n+1)))

def s4(n):
   return sum(map(lambda x : x * (x + 1) ,xrange(1,n+1)))
```

W przypadku ciągów malejących wartości należy sumować w drugą stronę, od najmniejszych. Wtedy małe wyrazy przed dodaniem dużego mają szansę się dodać i wnieść swój wkład, a jeżeli najpierw dodamy duże, to małe mogą być tak małe przy już policzonej sumie, że zostaną zignorowane. Przy tworzeniu ułamka pamiętajmy o wymuszeniu dzielenia liczb wymiernych, nie całkowitych.

```
def s5(n):
   return sum(map(lambda x : 1.0 / x, xrange(n, 0, -1)))    # 1.0 — ważne!

def s6(n):
```

```
    return sum(map(lambda x : (1 - 2 * (x  % 2)) * 1.0 / x, xrange(n, 0, -1)))
                                                          # 1.0 — ważne!

def s7(n):
    return sum(map(lambda x : 2 * x / (2 * x + 1) ** .5, xrange(1,n+1)))

def s8 (n):
    return sum(map(lambda x : x ** 2, xrange(1,n+1)))    # 1.0 — ważne!
```

Zadanie 15. Podziały

Podziałem liczby naturalnej *n* nazywamy każde przedstawienie jej w postaci sumy liczb naturalnych.

Wyznacz wszystkie podziały podanej liczby.

Przykład dla *n* = 7

```
-----------------------
6 + 1
5 + 1 + 1
4 + 1 + 1 + 1
3 + 1 + 1 + 1 + 1
2 + 1 + 1 + 1 + 1 + 1
1 + 1 + 1 + 1 + 1 + 1 + 1
2 + 2 + 1 + 1 + 1
3 + 2 + 1 + 1
4 + 2 + 1
2 + 2 + 2 + 1
3 + 3 + 1
5 + 2
3 + 2 + 2
4 + 3
```

Do rozwiązań w językach Pascal, C++ i JS należy przygotować **globalną** tablicę jednowymiarową typu całkowitego (`int`) o nazwie `tablica`, w której będą przechowywane i następnie wyświetlane kolejne rozkłady.

pas

Wywołanie funkcji: `z15(n, 0, 0, 0);`

```
.....................................................
procedure z15(n, m, q, i: integer);
var j: integer;
begin
   repeat
      dec(n); inc(m); tablica[i] := q;
      if (n >= m) and (m >= q) then
      begin
         write(n, '+', m);
         for j := i downto 1 do write('+', tablica[j]);
         writeLn;
         z15(n, 0, m, i + 1);
```

```
      end;
   until n < m;
end;
```

cpp

Słowo kluczowe `void` oznacza typ nieokreślony. Służy on m.in. do definiowania funkcji, które nie zwracają wyniku, a wykonują inne czynności, np. wyświetlanie. Dlatego wywołanie takiej funkcji polega na podaniu jedynie nazwy z parametrami aktualnymi.

Wywołanie funkcji: `z15(n, 0, 0, 0);`

```
.....................................................
void z15(int n, int m, int q, int i)
{
   do
   {
      n--; m++; tablica[i] = q;
      if((n >= m) && (m >= q))
      {
         cout << n << "+" << m;
         for(int j = i; j >= 1; j--) cout << "+" << tablica[j];
         cout << endl;
         z15(n, 0, m, i + 1);
      }
   }
   while(n >= m);
}
```

js

Wywołanie funkcji: `z15(n, 0, 0, 0);`

```
.....................................................
var tablica = new Array(1000);    // tablica globalna

function z15(n, m, q, i)
{
   var j;
   var s = "";
   do
   {
      n--; m++; tablica[i] = q;
      if((n >= m) && (m >= q))
      {
         s = n + "+" + m;
         for(j = i; j >= 1; j--) s = s + "+" + tablica[j];
         console.log(s + "\n");
         z15(n, 0, m, i+1);
      }
   }
   while(n >= m);
}
```

Wszystkie podziały liczby `n`, których pierwszym elementem jest liczba `k`, mają postać `[i]` + pewien podział liczby `n-i`. Wszystkie podziały uzyskamy, iterując to wyrażenie po wszystkich `i` oraz po wszystkich podziałach liczby `n-i`. Prowadzi to do funkcji rekurencyjnej:

```
function z15_2(n)
{
   if(n == 0) {return [[]]}
   else
   {
      var wynik = [];
      for(var i = 1; i <= n; i++)
      {
         var N = z15_2(n - i);
         for(var j = 0; j < N.length; j++)
         {
            wynik.push([i].concat(N[j]));
         }
      }
      return wynik;
   }
}
```

py

```
def z15(n):
   if n == 0:
      return [[]]                # Liczba zero ma jeden podział na sumę liczb
   else:                         # naturalnych dodatnich – podział pusty
      lista = []
      for i in xrange(1, n + 1):  # i będzie pierwszą liczbą podziału
         for j in z15(n - i):     # pozostałe elementy podziału tworzą j =
            lista.append([i] + j) # podział liczby n-i - wywołanie
                                  # rekurencyjne
      return lista
```

Możemy skrócić jej składnię, wykorzystując konstrukcję listy składanej i trójelementowej instrukcji warunkowej.

```
def z15(n):
   return [[]] if n == 0 else [[i] + j for i in xrange(1, n + 1) for j in
   ↪z15(n - i)]
```

W ten sposób dostaliśmy wszystkie uporządkowane podziały, tzn. dwa podziały różniące się kolejnością składników traktowane są jako różne. Jeżeli chcemy wygenerować listę nieuporządkowanych podziałów (czyli podziały różniące się kolejnością składników traktowane są jako jeden i ten sam podział), jest to równoważne zadaniu wygenerowania listy podziałów niemalejących (lub nierosnących). Potrzebujemy drugiego argumentu, który będzie przekazywał informację o maksymalnej wartości elementów podziału. Każdy niemalejący podział o składniku nie mniejszym od `m` jest postaci `[i]` (gdzie `i` $\geq$ `m`) + pewien podział z `z15(n - i, i)`. Żeby dostać wszystkie podziały, musimy wykonać iterację na tym wyrażeniu po wszystkich `i >= m` i po wszystkich elementach `z15(n - i, i)`. Podział jednoelementowy trzeba dodać ręcznie.

Prowadzi to do funkcji rekurencyjnej:

```
def z15(n, m = 1):
   if n == m:
      return [[n]]    # tylko jeden podział możliwy
   else:
      lista = []
      for i in range(m, n / 2 + 1):     # pierwszy element podziału >= m
         for j in z15(n - i, i):        # drugi element to podział liczby n – i
```

```
            lista.append([i] + j)    # z ograniczeniem i
        lista.append([n])
        return lista
```

Możemy skrócić jej składnię, wykorzystując konstrukcję listy składanej i trójelementowej instrukcji warunkowej.

```
def z15(n , m = 1):
   return [[n]] if n == m else [[i] + j for i in range(m, n / 2 + 1)
   for j in z15(n - i, i)] + [[n]]
```

Drugiemu argumentowi nadajemy wartość domyślną `1`, żeby nie trzeba było jej zawsze podawać przy wywoływaniu.

Zadanie 16. Rozkłady

Rozkład liczby naturalnej na czynniki pierwsze to zapisanie jej w postaci iloczynu liczb pierwszych.

Wyznacz rozkład podanej liczby, o ile jest ona większa od 1.

Np. $600 = 2 \cdot 2 \cdot 2 \cdot 3 \cdot 5 \cdot 5$

Algorytm:

Podana liczba będzie dzielona całkowicie przez dzielnik $d = 2$ tak długo, jak to będzie możliwe. Następnie zmienna `d` będzie zwiększana o 1 i czynności dzielenia powtórzą się. Dzielenie przez liczby złożone, np. 4, 6, itp., nie powiedzie się, gdyż już wcześniej „wyczerpały" to liczby pierwsze 2 i 3. Tak więc nie ma zupełnie potrzeby sprawdzania, czy kolejne dzielniki są liczbami pierwszymi.

pas

```
procedure z16(n: integer);
var d, schowek, czy: integer;
begin
   d := 2;
   schowek := n;    { przechowanie podanej liczby }
   write(n, '=');
   czy := 0;
   while(d < schowek) do
   begin
      while(n mod d = 0) do    { sprawdzanie podzielności }
      begin
         czy := 1;
         n := n div d;
         if n > 1 then
         write(d, '*') else write(d);    { żeby na końcu nie było znaku * }
      end;
      inc(d);    { zwiększanie dzielnika o 1 }
   end;
   if czy = 0 then write(schowek);    { wyświetlanie, jeśli jest to liczba pierwsza }
end;
```

cpp

```
void z16(int n)
{
   int d = 2, czy = 0, schowek;
   schowek = n;    // przechowanie danej liczby
   cout << n << "=";
```

```
    while(d < schowek)
    {
       while(n % d == 0)    // sprawdzanie podzielności
       {
          czy = 1;    // zaznaczenie, że wystąpiło dzielenie
          n = n / d;
          if(n > 1) cout <<d << "*"; else cout << d;  //żeby na końcu nie było znaku *
       }
       d++;    // zwiększanie dzielnika o 1
    }
    if(czy == 0) cout << schowek;    // wyświetlanie, jeśli liczba jest pierwszą
}
```

js

```
function z16(n)
{
   var d = 2, czy = 0, schowek;
   schowek = n;    // przechowanie danej liczby
   var uchwyt = document.getElementById("dif");
   uchwyt.innerHTML = n + "=";
   while(d < schowek)
   {
      while(n % d == 0)    // sprawdzanie podzielności
      {
         czy = 1;    // zaznaczenie, że wystąpiło dzielenie
         n = Math.floor(n / d);
         if(n > 1)
            uchwyt.innerHTML = uchwyt.innerHTML + d + "*";
         else
            uchwyt.innerHTML = uchwyt.innerHTML + d; //żeby na końcu nie było znaku *
      }
      d++;    // zwiększanie dzielnika o 1
   }
   if(czy == 0)
      uchwyt2.innerHTML = uchwyt2.innerHTML + schowek;
   // wyświetlenie, jeśli jest to liczba pierwsza
}
```

py

Chcemy wyznaczyć listę wszystkich dzielników pierwszych pewnej liczby n. Zauważmy, że jeżeli liczba p dzieli n, to lista dzielników liczby n jest postaci [p] + lista dzielników liczby n/p.

```
def z16(x):
   for i in xrange(2, int(x ** .5) + 1):
      if x % i == 0:
         return [i] + z16(x / i)
   return [x]
```

Zadanie 17. Trójkąt Pascala.

Jednym z najbardziej interesujących układów liczbowych jest trójkąt Pascala.

Jest to ciąg liczb, który może być ułożony w trójkąt, gdzie każdy wiersz przedstawia współczynniki liczbowe dwumianu $(a + b)^n$.

$(a + b)^0$ — współczynnik 1,

$(a + b)^1$ — współczynniki to 1 i 1,

$(a + b)^2$ — współczynniki to 1, 2 i 1,
$(a + b)^3$ — współczynniki to 1, 3, 3 i 1
$(a + b)^4$ — współczynniki to 1, 4, 6, 4 i 1 itd.

```
0                    1
1                  1   1
2                1   2   1
3              1   3   3   1
4            1   4   6   4   1
5          1   5   10  10   5   1
6        1   6   15  20  15   6   1
7      1   7   21  35  35   21  7   1
8    1   8   28  56  70  56   28  8   1
9  1   9  36   84  126 126  84  36  9   1

. . . . . .
```

Zbudujemy taki trójkąt, umieszczając liczby w tablicy dwuwymiarowej:

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
1   5   10  10   5   1
1   6   15  20  15   6   1
1   7   21  35  35   21  7   1
1   8   28  56  70  56   28  8   1
1   9  36   84  126 126  84  36  9   1
```

Tablica będzie indeksowana od zera, zatem:

wiersz nr 0 będzie miał postać: 1,
wiersz nr 1 będzie miał postać: 1, 1
wiersz nr 2 będzie miał postać: 1, 2, 1
wiersz nr 3 będzie miał postać: 1, 3, 3, 1
wiersz nr 4 będzie miał postać: 1, 4, 6, 4, 1
itd.

pas

```
procedure z17(n: integer);
var w, e: integer;
   tab: array [0..100, 0..100] of integer;
begin
   { --- wypełnianie tablicy --- }
   for w := 0 to n do
   begin
      tab[w, 0] := 1; tab[w, w] := 1;
      if w >= 2 then
      for e := 1 to w - 1 do tab[w, e] := tab[w - 1, e - 1] + tab[w - 1, e];
   end;

{ --- wyświetlenie  tablicy --- }
   writeLn(1);
   for w := 1 to n do
   begin
```

```
      write(tab[w, 0],'  ');
      if w >= 2 then for e := 1 to w - 1 do write(tab[w, e], '  ');
      if w > 0 then writeLn(tab[w, w], '  ');
   end;
end;
```

cpp

```
void z17(int n)
{
   int w, e;
   int tab[100][100];
   // --- wypełnianie tablicy
   for (w = 0; w <= n; w++)
   {
      tab[w][0] = 1; tab[w][w] = 1;
      if(w >= 2)
      for(e = 1; e <= w - 1; e++)
      tab[w][e] = tab[w - 1][e - 1] + tab[w - 1][e];
   }
   // --- wyświetlenie tablicy
   cout << 1 << endl;
   for(w = 1; w <= n; w++)
   {
      cout << tab[w][0] << " ";
      if(w >= 2)
         for(e = 1; e <= w - 1; e++) cout << tab[w][e] << "  ";
      if(w > 0) cout << tab[w][w] << endl;
   }
}
```

js

Umieścimy liczby w tablicy dwuwymiarowej, którą stworzymy poprzez zdefiniowanie tablicy tablic, gdyż w JavaScripcie tablice wielowymiarowe jako takie nie istnieją.

```
function z17(n)
{
   var uchwyt = document.getElementById("dif");

   // deklaracja tablicy
   var tab = new Array(100);
   for(i = 0; i <= 100; i++) tab[i] = new Array(100);

   // wypełnianie tablicy
   for(w = 0; w <= n; w++)
   {
      tab[w][0] = 1; tab[w][w] = 1;
      if(w >= 2)
         for(e = 1; e <= w - 1; e++)
      tab[w][e] = tab[w - 1][e - 1] + tab[w - 1][e];
   }

   // wyświetlenie  tablicy
   uchwyt.innerHTML = uchwyt.innerHTML + "1 <br>";
   for(w = 1; w <= n; w++)
   {
      for(e = 0; e <= n; e++)
         if(tab[w][e] != undefined)
```

```
                uchwyt.innerHTML=uchwyt.innerHTML+tab[w][e]+"__";
            uchwyt.innerHTML = uchwyt.innerHTML + " <br>";
        }
    }
```

py

W trójkącie Pascala *i*-ty wyraz w wierszu, z wyjątkiem wyrazów skrajnych, które są jedynkami, jest sumą wyrazów o numerach *i* oraz (*i* – 1) w wierszu powyżej.

Wiersz *i*-ty otrzymujemy, dodając dwie przesunięte o 1 kopie wiersza (*i* – 1)-szego, np.

```
      0 1 2 1
+     1 2 1 0
      --------
      1 3 3 1
```

Jeżeli `l` jest wyrazem (*i* – 1)-szym, to wyraz *i*-ty dostajemy na postawie powyższego przepisu za pomocą składni `map(sum,zip([0] + l[-1], l[-1] + [0]))`. Lista `zip([0] + l[-1], l[-1] + [0])` to lista pionowych par elementów do wysumowania (sprawdź!). Interesują nas sumy tych par, dlatego odwzorowujemy listę par przez funkcję `sum`. Generujemy tak listę kolejnych wierszy, zaczynając od wiersza zerowego `[1]`.

```
def z17(n):
    l = [[1]]
    for i in range(n):
        l.append(map(sum, zip([0] + l[-1], l[-1] + [0])))
return l
```

Konstruujemy rekurencyjnie kolejne wyrazy ciągu (w tym wypadku ciągu ciągów) na podstawie wyrazu wcześniejszego. W takiej sytuacji, podobnie jak w zadaniu 13., można to zrealizować za pomocą instrukcji `reduce`:

```
def z17(n):
    return reduce(lambda l , x : l + [map(sum, zip(l[-1] + [0],[0] + l[-1]))],
    ↪xrange(n),[[1]])
```

Ponieważ jest to czysta rekurencja, więc licznik iteracji `x` nie odgrywa żadnej roli w generowaniu nowych wyrazów ciągu.

Zadanie 18. Liczba π

Liczba π, inaczej ludolfina (od imienia holenderskiego matematyka Ludolfa van Ceulena), określa stosunek długości okręgu do długości jego średnicy.

π ≈ 3,14159 26535 89793 23846 26433 83279 50288 41971 69399 37510 58209 74944 59230 78164 06286 20899 86280 34825 34211 70679 82148 08651 32823 06647 09384 46095 50582 23172 53594 08128 48111 74502 84102 70193 85211 05559 64462 29489 54930 38196…

Oblicz przybliżoną wartość liczby π, stosując szereg Leibniza

$\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}+\cdots$ dla podanej liczby elementów szeregu. Dla wzoru

$\frac{\pi}{4}\approx 1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\frac{1}{11}$ liczba ta wynosi 6.

Oto kolejne przybliżenia liczby π otrzymane tą metodą:

$n = 1 \quad \Rightarrow \pi \approx 4$

$n = 2 \quad \Rightarrow \pi \approx 2{,}666666666666667$

$n = 3 \quad \Rightarrow \pi \approx 3{,}466666666666667$

$n = 4 \quad \Rightarrow \pi \approx 2{,}8952380952380956$

$n = 5 \quad \Rightarrow \pi \approx 3{,}3396825396825403$

.

.

.

$n = 100\,000 \quad \Rightarrow \pi \approx 3{,}1415826535897198$

.

.

.

$n = 1\,000\,000 \quad \Rightarrow \pi \approx 3.1415916535897743$

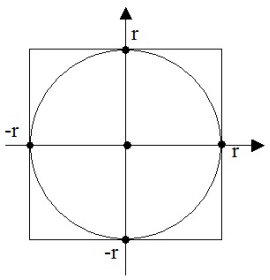
pas

```
function z18(n: longint): double;
var i: integer;
   suma: double;
begin
   suma := 0;
   for i := 1 to n do
      if i mod 2 = 0 then suma := suma - 1 / (i * 2 - 1)
      else suma := suma + 1 / (i * 2 - 1);
   z18 := suma * 4;
end;
```

cpp

```
double z18(long int n)
{
   int i;
   double suma = 0;
   for(i = 1; i <= n; i++)
      if(i % 2 == 0) suma = suma - 1.0 / (i * 2 - 1);
      else suma = suma + 1.0 / (i * 2 - 1);
   return suma * 4;
}
```

Innym sposobem wyznaczania przybliżonej wartości liczby π jest zastosowanie metody Monte Carlo.



Pole koła: $C = \pi \cdot r^2$

Pole kwadratu: $S = 4 \cdot r^2$

$$\frac{C}{S} = \frac{\pi \cdot r^2}{4 \cdot r^2} = \frac{\pi}{4} \Rightarrow \pi = \frac{4C}{S} = 4 \cdot \frac{\text{pole\_koła}}{\text{pole\_kwadratu}}$$

$$\pi = 4 \cdot \frac{\text{pole\_koła}}{\text{pole\_kwadratu}} \approx 4 \cdot \frac{N_{koła}}{N_{kwadratu}}$$

gdzie:

1. $N_{kwadratu}$ to liczba losowo wybranych punktów wewnątrz kwadratu,
2. $N_{koła}$ to liczba spośród wylosowanych już punktów kwadratu, które znajdą się wewnątrz koła.

Dla uproszczenia będziemy losować liczby rzeczywiste z przedziału ⟨–1; 1⟩.

Losowanie odbywać się będzie za pomocą generatora liczb pseudolosowych.

```
double monte_carlo(int ile)
{
   srand(time(NULL));    // uruchomienie generatora losowania
   long int i, n = 0;
   double pi, x, y;

   for(i = 1; i <= ile; i++)
   {
      // losowanie liczb rzeczywistych z zakresu <–1; 1>
      // stała RAND_MAX jest maksymalną wartością,
      // jaka może być zwrócona przez funkcję rand()

      x = ((double)rand()/(RAND_MAX)) * 2 - 1;
      y = ((double)rand()/(RAND_MAX)) * 2 - 1;

      // sprawdzenie, czy punkt (x; y) należy do koła
      // o promieniu = 1
      if(x * x + y * y <= 1) n++;
   }
   pi = 4.0 * n / ile;
   return pi;
}
```

js

```
function z18(n)
{
   var i;
   var suma = 0;
   for(i = 1; i <= n; i++)
      if(i % 2 == 0) suma = suma - 1.0 / (i * 2 - 1);
      else suma = suma + 1.0 / (i * 2 - 1);
   return suma * 4;
}
```

py

Obliczmy sumę szeregu jak w zadaniu 14.

```
def z18(n):
   return 4 * sum(map(lambda x : (1 - 2 * (x % 2)) * 1.0 / (2 * x + 1),
   ↪xrange(n - 1, -1, -1)))
```

Możemy też najpierw zsumować wyrazy dodatnie, potem ujemne i odjąć od siebie te dwie sumy.

```
def z18(n):
   return 4 * (sum(map(lambda x : 1.0 / (4 * x + 1), xrange(n / 2 - 1, -1,
   ↪-1)))
- sum(map(lambda x : 1.0 / (4 * x + 3), xrange(n / 2 - 1, -1, -1))))
```

Można też wykonać dodawanie w każdej parze i zsumować uzyskane liczby dodatnie.

```
def z18(n):
   return 8 * sum(map(lambda x : 1.0 / ((4 * x + 1) * (4 * x + 3)),
   ↪xrange(n / 2 - 1, -1, -1)))
```

Efektywności tych trzech rozwiązań przedstawiają się następująco:

```
from time import time
n = 5000000
t = time()
print 4 * sum(map(lambda x :(1 - 2 * (x % 2)) * 1.0 / (2 * x + 1),
↪xrange(n - 1, -1, -1))), 'oryginalne sformułowanie: ', time() - t
t = time()
print 4 * (sum(map(lambda x: 1.0 / (4 * x + 1), xrange(n / 2 - 1, -1, -1))) -
↪sum(map(lambda x: 1.0 / (4 * x + 3), xrange(n / 2 - 1, -1, -1)))), 'różnica
↪sum:              ', time() - t
t = time()
print 8 * sum(map(lambda x : 1.0 / ((4 * x + 1) * (4 * x + 3)), xrange(n /
↪2 - 1, -1, -1))), 'suma różnic:              ', time() - t
------------------------------------------------
3.14159245359
oryginalne sformułowanie:  1.56546807289
3.14159245359
różnica sum:               1.00716400146
3.14159245359
suma różnic:               0.662560224533
```

Zadanie 19. Liczba *e*

Liczba *e* to podstawa logarytmu naturalnego wykorzystywana w wielu dziedzinach matematyki i fizyki. W przybliżeniu wynosi 2,718281828459. Zwana jest też liczbą Eulera lub liczbą Nepera.

Oblicz przybliżoną wartość liczby *e*, stosując szereg $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots$ dla podanej liczby określającej występującą we wzorze silnię, licząc od zera.

Dla wzoru $e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \cdots$ liczba ta to 4.

Oto kolejne przybliżenia liczby *e* otrzymane tą metodą:

$n = 0 \quad \Rightarrow e \approx 1$

$n = 1 \quad \Rightarrow e \approx 2$

$n = 2 \quad \Rightarrow e \approx 2{,}5$

$n = 3 \quad \Rightarrow e \approx 2{,}6666666666666665$

$n = 4 \quad \Rightarrow e \approx 2{,}708333333333333$

.

.

.

$n = 10\,000 \quad \Rightarrow \mathrm{e} \approx 2{,}7182818284590455$

pas

```
function z19(n: longint): double;
var i: integer;
   s, m: double;
begin
   m := 1.0;
   s := 1.0;
   if n = 0 then z19 := s
   else
      for i := 1 to n do
      begin
         m := m / i;
         s := s + m;
      end;
   z19 := s;
end;
```

cpp

```
double z19(int n)
{
   int i;
   double s = 1.0, m = 1.0;
   if(n == 0) return s;
   else
      for(i = 1; i <= n; i++)
      {
         m = m / i;
         s = s + m;
      }
   return s;
}
```

js

```
function z19(n)
{
   var i;
   var m = 1.0, s = 1.0;
   if(n == 0) return s;
   else
      for(i = 1; i <= n; i++)
      {
         m = m / i;
         s = s + m;
      }
   return s;
}
```

py

Będziemy potrzebowali odwrotności silni. W jednej pętli możemy jednocześnie generować kolejną silnię i dodawać do sumy kolejny składnik.

```
def z19(n):
   s, m = (1.0, 1)
   for i in xrange(1, n):
      m = m * i
      s = s + 1.0 / m
   return s
```

Lepiej jednak w każdym przebiegu zamiast mnożenia i dzielenia wykonywać tylko dzielenie.

```
def z19(n):
   s, m = (1.0, 1.0)
   for i in xrange(1, n):
      m = m / i
      s = s + m
   return s
```

Możemy też wygenerować listę odwrotności silni za pomocą `reduce`, a następnie ją zsumować:

```
def z19(n):
   return sum(reduce(lambda x, y : x + [1.0 * x[-1] / y], xrange(1, n), [1]))
```

Lepiej jednak zawrzeć sumowanie od razu w instrukcji `reduce`.

```
def z19(n):
   return reduce(lambda x, y : (x[0] + x[1], x[1] / y), xrange(1, n), (0, 1.0))
```

Proporcja czasów wykonania powyższych rozwiązań dla $n = 100$ wynosi 24/19/65/24.

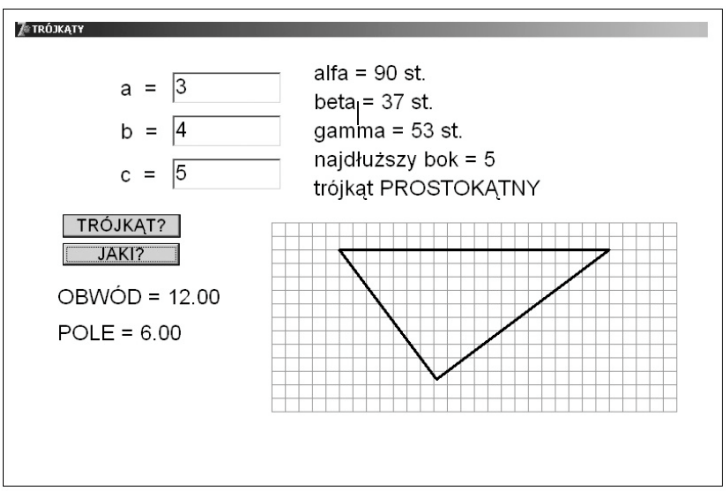
Zadanie 20. Rozwiązywanie trójkąta

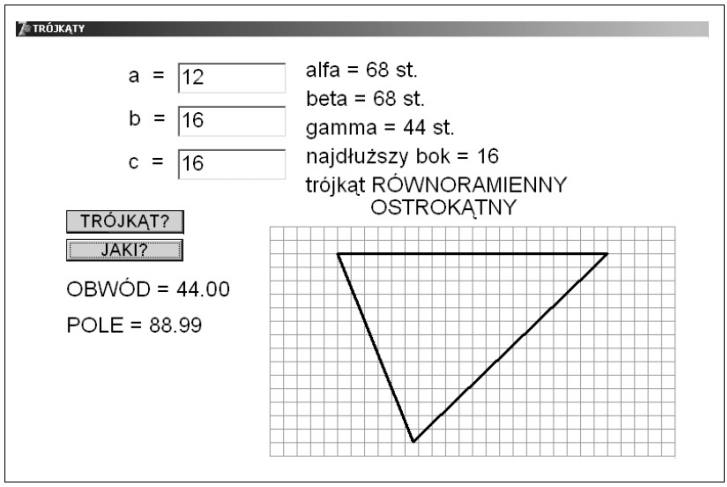
Dane są 3 liczby dodatnie. Sprawdź, czy mogą być one długościami boków trójkąta, a jeśli tak, to rozwiąż taki trójkąt.

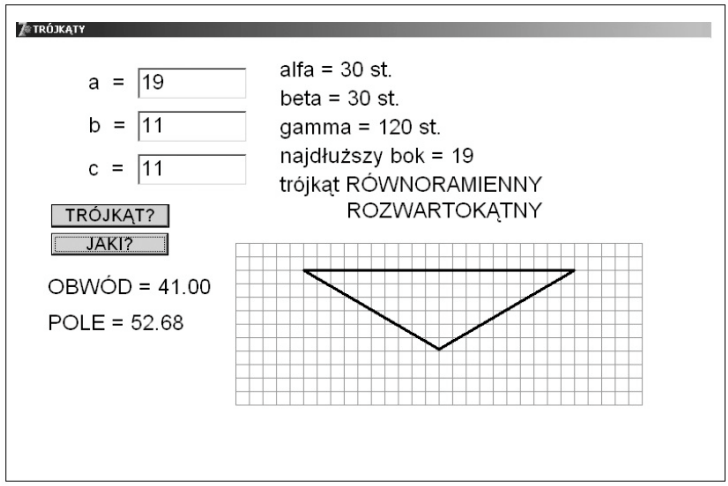
Rozwiązanie trójkąta polegać będzie na obliczeniu jego obwodu i pola oraz miar kątów.

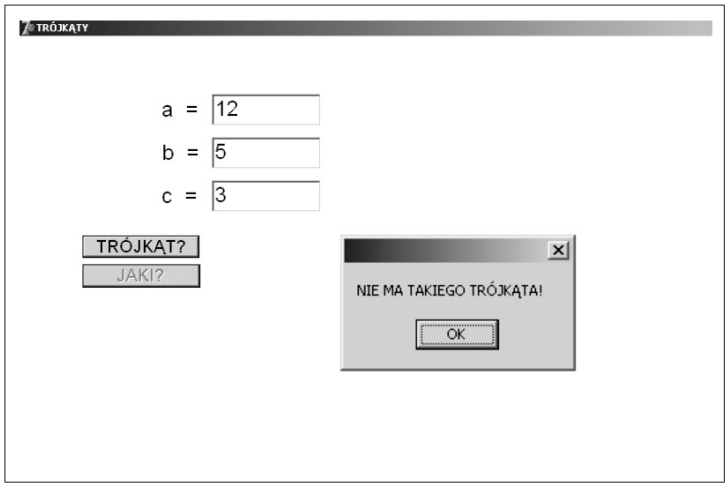
Można też określić rodzaj trójkąta.

W środowisku graficznym Delphi, Lazarusa czy też Visual Basica mogłoby to wyglądać tak:





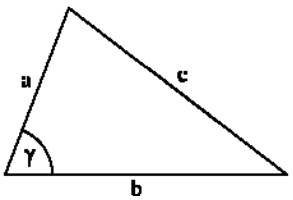




Można najpierw sprawdzić, czy podane liczby mogą być długościami boków jakiegoś trójkąta, czyli sprawdzić warunek: $a + b > c \wedge c + b > a \wedge c + a > b$. Pozostawiamy to czytelnikowi.

Pole obliczymy ze wzoru Herona: $\text{Pole} = \sqrt{p(p-a)(p-b)(p-c)}$, gdzie p to połowa obwodu trójkąta.

Miary kątów obliczymy, stosując twierdzenie cosinusów.



$$c^2 = a^2 + b^2 - 2ab \cdot \cos\gamma$$

$$2ab \cdot \cos\gamma = a^2 + b^2 - c^2$$

$$\cos\gamma = \frac{a^2 + b^2 - c^2}{2ab}$$

$$\gamma = \arccos\left(\frac{a^2 + b^2 - c^2}{2ab}\right)$$

Należy pamiętać, że w językach programowania wszystkie funkcje trygonometryczne operują na miarach kątów wyrażonych w radianach oraz że wyniki działań na liczbach rzeczywistych są przeważnie przybliżone.

pas

Ponieważ w języku Pascal nie ma funkcji arcus cosinus, należy ją utworzyć za pomocą istniejącej funkcji arcus tangens, korzystając ze wzoru: `ArcCos(x) = ArcTan(sqrt(1 - x * x) / x)`. Pamiętać też trzeba, że wartości funkcji cosinus dla kątów rozwartych są ujemne i należy zastosować wzór redukcyjny.

Sprawdzenie, czy podane trzy liczby mogą być długościami boków jakiegoś trójkąta, pozostawiamy czytelnikowi.

```
{  typ obiektowy
  ob — obwód, p — pole, k1, k2, k3 — kąty, s — rodzaj trójkąta (tekst) }
type wynik = object
   ob, p, k1, k2, k3: double;
   s:  string;
end;
{ ---------------------------------- }
function acos(x: double): double;
begin
   if x <> 0 then
   begin
      acos := ArcTan(sqrt(1 - sqr(x)) / x);
      if acos < 0 then acos := PI + acos;
   end else acos := PI / 2;
end;
{ ---------------------------------- }
function z20(a, b, c: double): wynik;
var
   i, j: integer;
   d, z: double;
   w: wynik;

begin
   if a > b then begin z := a; a := b; b := z; end;
```

```
    if b > c then begin z := b; b := c; c := z; end;
    if a > b then begin z := a; a := b; b := z; end;


    d := c * c - a * a - b * b;

    if d < 0 then w.s := ' ostrokątny';
    else
       if d > 0 then w.s := ' rozwartokątny' else w.s := ' prostokątny';

    if((a = b) and (b = c)) then w.s: = w.s + ', równoboczny, '
    else if ((c = b) or (b = a)) then w.s := w.s + ', równoramienny, ';

    w.ob := a + b + c;
    w.p := sqrt((w.ob) * (a + b - c) * (b + c - a) * (c + a - b)) / 4;
    w.k1 := 180 * acos((a * a + b * b - c * c) / (2 * a * b)) / PI;
    w.k2 := 180 * acos((b * b + c * c - a * a) / (2 * b * c)) / PI;
    w.k3 := 180 * acos((c * c + a * a - b * b) / (2 * c * a)) / PI;

    z20 := w;
end;
```

Przykłady wywołania:

```
writeLn('obwód = ', z20(a, b, c).ob);
writeLn('pole = ', z20(a, b, c).p);
writeLn('alfa = ', z20(a, b, c).k1);
writeLn('beta = ', z20(a, b, c).k2);
writeLn('gamma = ', z20(a, b, c).k3);
writeLn('trójkąt', z20(a, b, c).s);
```

cpp

```
// typ strukturalny
// ob — objętość, p — pole, k1, k2, k3 — kąty, s — rodzaj trójkąta

struct wynik
{ double ob, p, k1, k2, k3; string s; };

wynik z20(double a, double b, double c)
{
   wynik w = {0, 0, 0, 0, 0, ""};
   double d;
   double t[3]; t[0] = a; t[1] = b; t[2] = c;
   sort(t, t + 3); a = t[0]; b = t[1]; c = t[2];
   if(c <= a + b) d = c * c - a * a - b * b; else return w;

   if(d < 0) w.s = " ostrokątny";
   else
      if(d > 0) w.s = " rozwartokątny";
      else w.s = " prostokątny";

   if(a == b && b == c) w.s += ", równoboczny, ";
   else if(c == b || b == a) w.s += ", równoramienny, ";

   w.o b = a + b + c;
   w.p = sqrt((w.ob) * (a + b - c) * (b + c - a) * ( c + a -b)) / 4;
   w.k1 = 180 * acos(1.0 * (a * a + b * b - c * c) / (2 * a * b)) / M_PI;
```

```
    w.k2 = 180 * acos(1.0 * (b * b + c * c - a * a) / (2 * b * c)) / M_PI;
    w.k3 = 180 * acos(1.0 * (c * c + a * a - b * b) / (2 * c * a)) / M_PI;
    return w;
}
```

Przykłady wywołania:

```
cout << z20(a, b, c).ob << endl;;
cout << z20(a, b, c).p << endl;;
cout << z20(a, b, c).k1 << endl;;
cout << z20(a, b, c).k2 << endl;;
cout << z20(a, b, c).k3 << endl;;
cout << z20(a, b, c).s << endl;;
```

js

```
function z20(a, b, c)
{
   // typ obiektowy
   // ob — objętość, p — pole, k1, k2, k3 — kąty, s — rodzaj trójkąta
   var w =
   {
      ob: 0,
      p: 0,
      k1: 0,
      k2: 0,
      k3: 0,
      s:  ""
   };

   var t = new Array(3);
   var d;
   t[0] = a; t[1] = b; t[2] = c;
   t.sort(); a = t[0]; b = t[1]; c = t[2];

   if(c <= a + b) d = c * c - a * a - b * b; else return w;
   if(d < 0) w.s = " ostrokątny";
   else
      if (d > 0) w.s = " rozwartokątny";
      else w.s = " prostokątny";
   if(a == b && b == c) w.s += ", równoboczny, ";
   else if(c == b || b == a) w.s += ", równoramienny, ";

   w.ob = a + b + c;

   w.p = Math.sqrt((a + b + c)* (a + b - c) * (b + c - a) * (c + a - b)) / 4;
   w.k1 = 180 * Math.acos((a *a + b * b - c * c) / (2 * a * b)) / Math.PI;
   w.k2 = 180 * Math.acos((b * b +c * c - a * a) / (2 * b * c)) / Math.PI;
   w.k3 = 180 * Math.acos((c * c + a * a - b * b) / (2 * c * a)) / Math.PI;

   return w;
}
```

Przykłady wywołania (wyniki w sześciu pojemnikach `<div>`):

```
var uchwyt1 = document.getElementById("dif1")
var uchwyt2 = document.getElementById("dif2")
var uchwyt3 = document.getElementById("dif3")
var uchwyt4 = document.getElementById("dif4")
var uchwyt5 = document.getElementById("dif5")
```

```
var uchwyt6 = document.getElementById("dif6")

uchwyt1.innerHTML = z20(a, b, c).ob;
uchwyt2.innerHTML = z20(a, b, c).p;
uchwyt3.innerHTML = z20(a, b, c).k1;
uchwyt4.innerHTML = z20(a, b, c).k2;
uchwyt5.innerHTML = z20(a, b, c).k3;
uchwyt6.innerHTML = z20(a, b, c).s;
```

py

Funkcja będąca rozwiązaniem zadania przyjmuje jako argumenty trzy długości boków i zwraca krotkę zawierającą: obwód, pole, kąty, opis (opis jest łańcuchem zawierającym cechy trójkąta, takie jak: ostrokątny, rozwartokątny, prostokątny, równoramienny równoboczny). Jeżeli liczby nie spełniają warunku trójkąta, funkcja zwraca `None`.

```
def z20(a, b, c):
   a, b, c =  sorted([a, b, c])
   if c <= a + b:
      d = c ** 2 - a ** 2 - b ** 2
      if d < 0:
      r = 'ostrokątny '
   elif d > 0:
      r = 'rozwartokątny '
   else:
      r = 'prostokątny '
   if a == b:
      if b == c:
         r += ', równoboczny '
      else:
         r += ', równoramienny '
   elif b == c:
      r +=  ', równoramienny '
   return (a + b + c, ((a + b + c) * ( a + b - c) * (b + c - a) *
   ↪(c + a - b)) ** .5 / 4,
      180 * acos(1.0 * (a ** 2 + b ** 2 - c ** 2) / (2 * a * b)) / pi,
      180 * acos(1.0 * (b ** 2 + c ** 2 - a ** 2) / (2 * b * c)) / pi,
      180 * acos(1.0 * (c ** 2 + a ** 2 - b ** 2) / (2 * c * a)) / pi, r)
```

Zadanie 21. Punkty kratowe

Punkt kratowy to punkt, którego współrzędne w układzie kartezjańskim są liczbami całkowitymi.

Dane jest koło o środku w początku układu współrzędnych i promieniu r.

Oblicz liczbę N wszystkich punktów kratowych leżących wewnątrz tego koła.

$r = 1 \quad \Rightarrow N = 1$

$r = 2 \quad \Rightarrow N = 9$

$r = 2{,}1 \quad \Rightarrow N = 13$

$r = 2{,}5 \quad \Rightarrow N = 21$

$r = 5 \quad \Rightarrow N = 69$

$r = 100 \quad \Rightarrow N = 31\,397$

pas

Rozwiązanie zadania tym razem w programie głównym:

```
program z21;
uses Crt;
var r: double;
   n, x, y, rr: integer;
begin
   ClrScr;
   n := 0;
   write('Promień = '); readLn(r);

   { obcięcie części ułamkowej liczby r }
   rr := Trunc(r);

   for x := -rr to rr do
      for y := -rr to rr do
         if(x * x + y * y < r * r) then inc(n);
   writeLn('N = ', n);

   readLn;
end.
```

cpp

Rozwiązanie zadania tym razem w programie głównym:

```
main()
{
   double r;
   int n = 0, x, y, rr;
   system("CLS");
   cout << "Promień = "; cin >> r;

   // obcięcie części ułamkowej liczby r
   rr = (int)floor(r);

   for(x = -rr; x <= rr; x++)
      for(y = -rr; y <= rr; y++)
         if(x * x + y * y < r * r) n++;
   cout << "N = " << n << endl;
   system("PAUSE");
   return 0;
}
```

js

```
function z21(r)
{
   var n = 0, x, y, rr;
   rr = parseInt(Math.floor(r));
   for(x = -rr; x <=r r; x++)
      for(y = -rr; y <= rr; y++)
         if(x * x + y * y < r * r) n++;
   console.log("N = ", n);
}
```

py

Musimy z kwadratu opisanego na kole odrzucić punkty spoza koła. Różnica tych dwóch zbiorów składa się z czterech narożników zawierających po tyle

samo punktów. Wystarczy więc policzyć punkty do odrzucenia w tylko jednej ćwiartce koła. Sprowadziliśmy zatem problem do policzenia w kwadracie punktów o dodatnich współrzędnych, które leżą poza kołem.

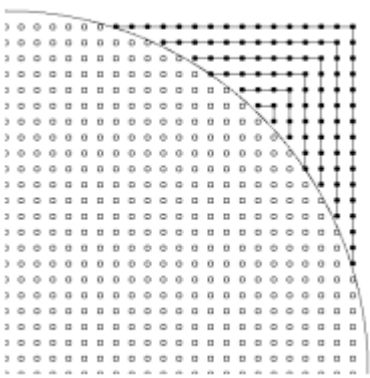
Najprostszą propozycją będzie:

```
from math import ceil

def z21(r):
    l = 0
    ir = int(ceil(r)) - 1       # int(ceil(r)) –1 — największa liczba
    for i in xrange(ir + 1):    # całkowita mniejsza niż r
        for j in xrange(ir + 1):
            if i * i + j * j >= r * *2:
                l += 1
    return (2 * ir + 1) ** 2 - 4 * l
```

Możemy ją poprawić, zauważając, że (z twierdzenia Pitagorasa) najmniejszą wartością współrzędnej *x* lub *y* punktu, który będzie odrzucony, jest `int((r ** 2 - ri ** 2) ** .5) + 1`:

```
from math import ceil

def z21(r):
    l = 0
    ir = int(ceil(r)) - 1
    m = int((r ** 2 - ri ** 2) ** .5)
    for i in xrange(m + 1, ir + 1):
        for j in xrange(m + 1, ir + 1):
            if i * i + j * j >= r ** 2:
                l += 1
    return (2 * ir + 1) ** 2 - 4 * l
```

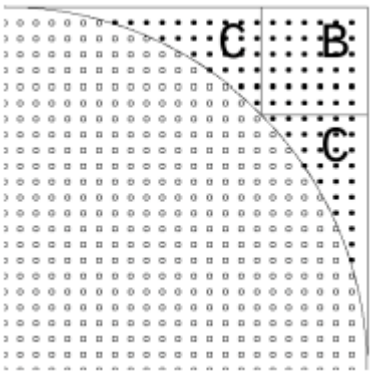
Ciągle jednak nie jest to optymalny algorytm. Zauważmy, że zbiór odrzucanych punktów jest symetryczny względem prostej $y = x$ i składa się z symetrycznych, zagiętych pod kątem prostych rzędów punktów (jak na rysunku).



Rozpoczynając od minimalnego *x*, sprawdzamy, czy w tym *x* zaczyna się rząd do usunięcia; jeżeli nie, to zwiększamy *x*. Jeżeli tak, to odejmujemy z obliczonego pola długość rzędu i zmniejszamy wartość $i(r)$ o 1. Rozpoczynamy poszukiwanie kolejnego pasa dla *x* powiększonego o 1. Poszukiwania zakończą się, gdy $x > i(r)$.

```
from math import ceil

def z21(r):
    ir = int(ceil(r)) - 1              # bok kwadratu opisanego
    l = (2 * ir + 1) ** 2              # liczba punktów w kwadracie opisanym
    x = int((r ** 2 - ir ** 2) ** .5) + 1    # początkowa wartość współrzędnej x
    while x <= ir:                     # pętla po x
        if ir ** 2 >= r ** 2 - x ** 2:         # jeżeli punkt leży poza wnętrzem koła
            l -= 8 * (ir - x) + 4      # odcinamy cztery pasy narożne
            ir -= 1                    # zmniejszamy górną granicę zakresu x
        x += 1                         # powiększamy x
    return l                           # po odcięciu wszystkich pasów narożnych zwracamy l
```

Zauważmy, że jedno odcięcie to jedno sprawdzenie mniej — suma liczby sprawdzeń i mnożeń nie przekracza *r*. Czas działania programu rośnie liniowo z *r*. W wersji naiwnej liczba sprawdzeń rośnie kwadratowo z *r*. Możemy również obliczyć długości odcinanych pasów. W kolejnym programie na początku od kwadratu opisanego odcinamy cztery maksymalne kwadraty w rogach (obszar *B* na rysunku). Pozostaje do usunięcia osiem obszarów pomiędzy kołem i usuniętym kwadratem narożnym (obszar *C* na rysunku).



Rzędy w obszarze *C* mają współrzędne *x* od $i(r)(1-\sqrt{2})$ (oznaczaną od teraz jako *isr*) do *ir*. Dla każdego rzędu w obszarze *C* jesteśmy w stanie obliczyć liczbę punktów do usunięcia. Wykonując iterację po rzędach, usuniemy pozostałe niepotrzebne punkty.

```
from math import ceil

def z21(r):
    ir = int(ceil(r)) - 1              # największa liczba całkowita mniejsza od r
    isr = int(ceil(r / 2 ** .5)) - 1   # największa liczba całkowita mniejsza od r / 2 ^ .5
    l = (2 * ir + 1) ** 2 - 4 * (ir - isr ) ** 2     # pole kwadratu opisanego bez 4
                                                     # kwadratów na rogach
    for y in xrange(isr + 1, ir + 1):
        x = int(ceil((r ** 2 - y ** 2) ** .5) - 1)   # długość rzędu do usunięcia
        l -= 8 * (isr - x)
    return l
```

Obie metody mają porównywalną wydajność. W pierwszej metodzie suma liczby porównań i odejmowań jest równa $i(r)=\sqrt{r^2-i(r)^2}+1$. W drugiej

obliczamy pierwiastek tylko $i(r)(1-\sqrt{2})$ razy, ale obliczanie pierwiastka jest bardziej kosztowną operacją niż mnożenie.

Zadanie 22. Pole figury

Dany jest stopień *n* oraz współczynniki $a_0, a_1, a_2, \ldots a_n - 2, a_n - 1, a_n$ wielomianu $W(x) = a_0x^n + a_1x^{n-1} + a_2x^{n-2} + ... + a_{n-2}x^2 + a_{n-1}x + a_n$.

Oblicz przybliżoną wartość pola figury ograniczonej osią X i wykresem funkcji $W(x)$ w podanym przedziale $\langle a; b\rangle$. Ograniczymy się do wielomianów stopnia co najwyżej 1000.

Dla rozwiązań w Pascalu i C++ należy przygotować plik tekstowy o nazwie *dane.txt*, w którym należy zapisać współczynniki wielomianu, każdy w osobnej linii. Wyjaśnia to poniższy przykład.

Program sam oblicza stopień wielomianu na podstawie liczby podanych współczynników.

Przykład zawartości pliku dla wielomianu $W(x) = -x^3 + 3x^2 + 10x$

```
–1
 3
10
 0
```

Do obliczenia pola zastosujemy **metodę trapezów** opisaną dokładnie w treści zadania.

pas

```
program z22;
uses Crt;
{ zmienne globalne }
var stopien: integer;
   wsp: array [0..1000] of real;

{ obliczanie wartości wielomianu za pomocą schematu Hornera i funkcji rekurencyjnej }
function horner(k: integer; x: real): double;
begin
   if k = 0 then horner := wsp[0] else horner := horner(k - 1, x) * x +
   ↪wsp[k];
end;

procedure pole;
var i, n, stopien: integer;
   a, b, pd, pg, h, dx: real;
   p: double;
   s: string; kod: byte;
   plik: text;    { deklaracja zmiennej plikowej dla pliku tekstowego }

begin
   stopien := 0;
   p := 0;
   assign(plik, 'dane.txt');
   reset(plik);
   repeat
```

```
   readLn(plik, s);    { <--- wczytanie do zmiennej tekstowej, a nie bezpośrednio jako
   liczby, w celu uniknięcia wprowadzania pustych linii, szczególnie tych na końcu pliku! }
   if s <> '' then
   begin
      val(s, wsp[stopien], kod); inc(stopien);
   end;
   until eof(plik);
   close(plik);
   stopien := stopien - 1;

   writeLn('  Stopień wielomianu = ', stopien);

   writeLn('  Podaj przedział <a; b> '  );
   write('          a = '); readLn(a);
   write('          b = '); readLn(b);
   write('   Ile części? '); readLn(n);

   { odległość między punktami }
   dx := (b - a) / n;

   { wysokość każdego z trapezów }
   h := dx;

   { obliczanie pól trapezów }
   for i := 1 to n do
   begin
      pd := horner(stopien, a + (i - 1) * dx);
      pg := horner(stopien, a + i * dx);
      p := p + abs((pg + pd) * h / 2);
   end;
   writeLn('Pole figury wynosi ', p, '. ');
end;

{ ------------ program główny ----------------- }
begin
   ClrScr;
   pole();
   readLn;
end.
```

cpp

```
#include <iostream>
#include <cmath>
#include <fstream>
using namespace std;

// zmienne globalne
int stopien;
double wsp[1000];

// obliczanie wartości wielomianu za pomocą schematu Hornera i funkcji rekurencyjnej
double horner(int k, double x)
{
   if k == 0) return wsp[0]; else return horner(k - 1, x) * x + wsp[k];
}

double pole()
{
```

```
    int i, n, stopien = 0;
    double a, b, d, dx, p = 0;
    double pd, pg, h;
    fstream plik;
    plik.open("dane.txt", ios :: in);

    // odczyt danych z pliku
    while(plik >> d)
    {
       wsp[stopien] = d;
       stopien++;
    }
    plik.close();

    stopien = stopien - 1;
    cout << "  Stopień wielomianu = " << stopien << endl;
    cout << "  Podaj przedział <a; b> " << endl;
    cout << "           a = "; cin >> a;
    cout << "           b = "; cin >> b;
    cout << "  Ile części? "; cin >> n;

    // odległość między punktami
    dx = (b - a) / n;

    // wysokość każdego trapezu
    h = dx;

    // obliczanie pól trapezów
    for(i = 1; i <= n; i++)
    {
       pd = horner(stopien, a + (i - 1) * dx);
       pg = horner(stopien, a + i * dx);
       p = p + abs((pg + pd) * h / 2);
    }
    return p;
}
// ---------- program główny ----------
main()
{
    system("CLS");
    cout<<"Pole figury wynosi " << pole() << "." << endl;
    system("PAUSE");
    return 0;
}
```

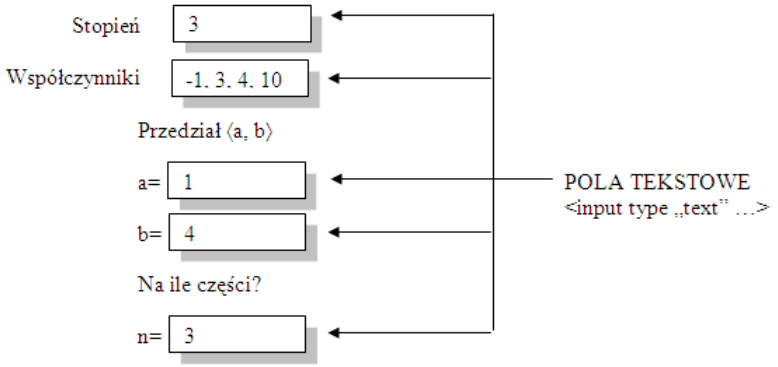
js

Funkcja `split()` pozwala na podział tekstu na tzw. **tokeny**, czyli ciągi znaków oddzielone wyróżnionym jednym znakiem, np. spacją. W przypadku czystego tekstu **tokeny** mogą być całymi wyrazami.

W przypadku strumienia zawierającego liczby można je oddzielić także spacją albo innym separatorem.

W naszym rozwiązaniu zastosowaliśmy przecinek.

Struktura danych:



Współczynniki należy oddzielić przecinkami (bez żadnych spacji). Liczby dziesiętne zapisujemy z kropką oddzielającą część całkowitą od ułamkowej.

```
<script type="text/javascript">
// zmienne globalne
var wsp = new Array(20);    // wymiar tablicy można zwiększyć w miarę potrzeb

var stopien;

// obliczanie wartości wielomianu za pomocą schematu Hornera i funkcji rekurencyjnej

function horner(k, x)
{
   if(k==0) return wsp[0]; else return horner(k - 1, x) * x + wsp[k];
}

function z22()
{
   var i;
   var s = "";
   var d, dx, p = 0, g, h;

   var uchwyt1 = document.getElementById("pole1");
   var uchwyt2 = document.getElementById("pole2");
   var uchwyt3 = document.getElementById("pole3");
   var uchwyt4 = document.getElementById("pole4");
   var uchwyt5 = document.getElementById("pole5");
   var uch = document.getElementById("dif");
   uch.innerHTML = "";

   var a = uchwyt3.value; a = parseFloat(a);
   var b = uchwyt4.value; b = parseFloat(b);
   var n = uchwyt5.value; n = parseInt(n);

   var st = uchwyt1.value; stopien = parseInt(st);

   // odczyt współczynników do zmiennej typu "string"
   s = uchwyt2.value;

   // podzielenie tekstu na "tokeny"
     var t =  s.split(',');
```

```
    // wypełnianie tablicy współczynników "wsp"
    for(i = 0; i <= stopien; i++) wsp[i] = parseFloat(t[i]);

    // odległość między punktami
    dx = (b - a) / n;

    // wysokość każdego trapezu
    h = dx;

    // obliczanie pól trapezów
    for(i = 1; i <= n; i++)
    {
       pd = horner(stopien, a +(i - 1) * dx);
       pg = horner(stopien, a + i * dx);
       p = p + Math.abs((pg + pd) * h / 2);
    }

    uch.innerHTML = uch.innerHTML + p + "<br>";
}
</script>
```

py

Aby rozwiązać to zadanie, trzeba umieć policzyć wartość wielomianu stopnia d w zadanym punkcie. Wielomian będzie zadany przez listę swoich $d + 1$ współczynników. Powinniśmy każdy współczynnik wymnożyć przez odpowiadającą mu potęgę i dodać do siebie wyniki. Obliczanie każdej potęgi z osobna nie jest wydajnym pomysłem. Generując listę potęg i tworząc kolejny element poprzez pomnożenie poprzedniego przez x, wykonamy d mnożeń. Następne d mnożeń wykonamy, mnożąc kolejne współczynniki. Liczbę wszystkich mnożeń można zredukować do d, stosując schemat Hornera: mnożymy a_d przez x, dodajemy a_{d-1} i zapamiętujemy wynik jako w. Następnie mnożymy go przez x, dodajemy a_{d-2} i nadpisujemy nową wartością. Postępujemy tak aż do wyczerpania współczynników. Łatwo się przekonać, że wtedy w będzie wartością wielomianu w punkcie x. Implementujemy schemat Hornera:

```
def W(x):
   wynik = 0
   for i in l:
      wynik = wynik *x + i
   return wynik
```

W liście `l` współczynniki ułożone są w kierunku malejących potęg. Wywołujemy pewną funkcję na poprzednim wyniku tej funkcji i kolejnym elemencie listy, zaczynając od wartości `0`. Sugeruje to zapisanie tej funkcji za pomocą instrukcji `reduce`:

```
def W(x):
   return reduce(lambda i, j : i * x + j, l)
```

Gdy już potrafimy policzyć wartość wielomianu dla zadanej wartości, obliczamy szerokość podziału odcinka (dx), generujemy listę x równo rozłożonych punktów w odcinku oraz listę y odpowiadających im wartości. Sumujemy pola trapezów tworzonych przez odcinek dx, pionowe boki o długościach `y[i]`, `y[i+1]` i odcinek ukośny łączący ich końce. Z sumy pól

wysokość trapezu *dx* można wyłączyć przed nawias. W nawiasie zostanie suma połówek wartości funkcji. Wszystkie połówki z wyjątkiem pierwszej i ostatniej występują dwa razy, suma połówek będzie zatem równa `((y[0] + y[-1]) / 2 + sum(y[1 : -1]))`.

Całość funkcji liczącej pole pod wykresem wygląda tak:

```
def z22(a, b, n, l):    # l = lista współczynników, od najwyższego
   def W(x):
      return reduce(lambda i, j : i * x + j, l)
   dx = 1.0 * (b - a) / n
   x = [a + dx * i for i in range(n + 1)]
   y = map(W, x)
   return ((y[0] + y[-1]) / 2 + sum(y[1 : -1])) * dx
```

Zadanie 23. Rzut ukośny

Rzut ukośny to ruch ciała, któremu nadano prędkość skierowaną pod pewnym kątem do poziomu. Zakładamy, że ruch ten odbywa się bez żadnych oporów, np. oporu powietrza.

Dana jest odległość (*x*) oraz wysokość (*y*) w metrach. Napisz program, który obliczy, jaką prędkość początkową trzeba nadać, aby „pocisk" osiągnął odległość *x* i wysokość *y* dla kątów 1°, 2°, 3°, … 88°, 89°.

Wzór na zasięg rzutu ukośnego: $z = \frac{v_0^2 \cdot \sin 2\alpha}{g}$, gdzie:

v_0 — prędkość początkowa

α — kąt rzutu

$g \approx 9{,}81$ m/s^2 — przyśpieszenie ziemskie

$$y = x \cdot tg\alpha - \frac{g}{2v_0^2 \cdot \cos^2 \alpha} x^2 \qquad g \approx 9{,}81 \frac{m}{s^2}$$

Podana odległość (*x*) nie może przekraczać zasięgu (*Z*), a wysokość (*y*) nie może przekraczać wysokości maksymalnej (H_{max}) dla kolejnych prędkości początkowych (v_0). Jeżeli tak się stanie, to powinien pojawić się komunikat, np. „Cel poza zasięgiem!".

Wysokość maksymalną obliczymy z rzędnej wierzchołka paraboli $y = ax^2 + bx + c$, czyli $-\frac{\Delta}{4a}$, natomiast zasięg — wstawiając do równania paraboli $y = ax^2 + bx + c$ za *y* wartość 0 i obliczając *x* (rozwiązanie równania kwadratowego dodatnie).

pas

```
program z23;
uses crt;
{ ---- zamiana m/sek. na km/godz. ----}
function km(m: double): double;
begin
   km := 18 * m / 5;
end;
```

```
{ ---- prędkość ----}
function v(alfa, x, y: double): double;
var a, b, v0, Hmax, Z, tan, g: double;
begin
   { zamiana stopni na radiany }
   alfa := PI * alfa / 180;

   { prędkość początkowa }
   g := 9.81;
   tan := sin(alfa) / cos(alfa);
   if(g / (2 * (x * tan - y)) < 0) then v := 0
   else
      v0 := abs(x / cos(alfa)) * sqrt(g / (2 * (x * tan - y)));

   { maksymalny zasięg rzutu Z }
   Z := (2 * sqr(v0) * sqr(cos(alfa)) * tan) / g;

   { maksymalna wysokość Hmax }
   Hmax := (sqr(v0) * sqr(cos(alfa)) * sqr(tan)) / (2 * g);
   if (x > Z) or (y > Hmax) then v := 0 else v := v0;
end;

{ ---- program główny ---- }
var i: integer;
   width, height: double;
begin
   write('  Podaj odległość w metrach:  '); readLn(width);
   write('  Podaj  wysokość w metrach:  '); readLn(height);
   for i := 1 to 89 do
      if (v(i, width, height) = 0) then writeLn(' Dla kąta ', i, ' st. cel
      ➥poza zasięgiem!')
      else
      begin
         write(' Dla kąta ', i, ' st. ---> ');
         write(v(i, width, height) : 0 : 2);
         write('  m/sek = ');
         write(km(v(i, width, height)) : 0 : 2);
         writeLn('  km/godz. ');
      end;
   readLn;
end.
```

cpp

```
#include <iostream>
#include <cmath>
#include <iomanip>

using namespace std;

// zamiana m/sek. na km/godz.
double km(double m)
{
   return 18 * m / 5;
}
double q(double wyr)
{
  return wyr * wyr;
}
```

```
// wyznaczanie prędkości początkowej
double v(double alfa, double x, double y)
{
   const double g = 9.81;
   double v0, Hmax, Z;
   // zamiana stopni na radiany
   alfa =  M_PI * alfa / 180;
   // prędkość początkowa
   if(g / (2 * (x * tan(alfa) - y)) < 0)
      return 0;
   else
      v0 = abs(x / cos(alfa)) * sqrt(g / (2 * (x * tan(alfa) - y)));
   // maksymalny zasięg rzutu Z
   Z = (2 * q(v0) * q(cos(alfa)) * tan(alfa)) / g;
   // maksymalna wysokość Hmax
   Hmax = (q(v0) * q(cos(alfa)) * q(tan(alfa))) / (2 * g);
   if (x > Z || y > Hmax) return 0; else return v0;
}

// funkcja główna
main()
{
   int i;
   double width, height;
   cout << fixed << setprecision(2);
   cout <<"  Podaj odległość w metrach: "; cin >> width;
   cout <<"  Podaj  wysokość w metrach: "; cin >> height;
   for(i = 1; i < 90; i++)
   if(v(i, width, height) == 0)
      cout << " Dla kąta " << i << " st. cel poza zasięgiem! " << endl;
   else
      cout << " Dla kąta " << i << " st. --->  " << v(i, width, height) << "
      ➥m/sek = " << km(v(i, width, height)) << "  km/godz. " << endl;

   system("PAUSE");
   return 0;
}
```

js

```
function q(wyr)
{
   return wyr * wyr;
}
// funkcja wyznaczająca prędkość początkową
function v(alfa, x, y)
{
   const g = 9.81;
   var v0, Hmax, Z;

   // zamiana stopni na radiany
   alfa = Math.PI * alfa / 180;

   // prędkość początkowa
   if (g / (2 * (x * Math.tan(alfa) - y)) < 0)
      return 0;
   else
      v0 = Math.abs(x / Math.cos(alfa)) * Math.sqrt(g /
      ➥(2 *(x * Math.tan(alfa) - y)));
```

```
    // maksymalny zasięg rzutu Z
    Z = (2 * q(v0) * q(Math.cos(alfa)) * Math.tan(alfa)) / g;

    // maksymalna wysokość Hmax
    Hmax = (q(v0) * q(Math.cos(alfa)) * q(Math.tan(alfa))) / (2 * g);
    if((x > Z) || (y > Hmax)) return 0; else return v0;
}

function z23(angle, w, h)
{
    if(v(angle, w, h) == 0)
       console.log(" Dla kąta ", angle, " st. cel poza zasięgiem! ");
    else
       console.log(" Dla kąta ", angle," stopni -> ", v(angle, w, h)," m/sek ");
}
```

py

```
from math import sin

def z23(d):
    return [(d * 9.81 / sin(2 * pi * alpha / 180)) ** .5 for alpha in
    ↪range(1, 90)]
```

Zadanie 24. Równanie liniowe

Równanie liniowe z jedną niewiadomą ma postać $ax + b = 0$.

Napisz program rozwiązujący takie równanie.

Należy rozpatrzyć trzy przypadki:

1. $a \neq 0$ — równanie posiada jedno rozwiązanie $x = -b/a$.
2. $a = 0$ i $b \neq 0$ — równanie jest sprzeczne, nie posiada rozwiązań.
3. $a = 0$ i $b = 0$ — równanie jest tożsamościowe, każda liczba jest jego rozwiązaniem.

pas

```
procedure z24(a, b: double);
var x:double;
begin
    if a <> 0 then begin x := -b / a; writeLn('x = ', x); end
    else
       if b <> 0 then writeLn('Równanie sprzeczne.')
       else
          writeLn('Równanie tożsamościowe.');
end;
```

cpp

```
void z24(double a, double b)
{
    double x;
    if(a ! =0) { x = -b / a; cout << "x = " << x << endl; }
    else
       if (b != 0) cout << "Równanie sprzeczne." << endl;
       else
          cout << "Równanie tożsamościowe." << endl;
}
```

js

```
function z24(a, b)
{
   var x;
   if (a != 0) { x = -b / a; console.log("x = ", x); }
   else
      if (b!=0) console.log("Równanie sprzeczne.");
      else
         console.log("Równanie tożsamościowe.");
}
```

py

```
def z24(a, b):
   return -1.0 * b / a
```

Zadanie 25. Równanie kwadratowe

Równanie kwadratowe ma postać $ax^2 + bx + c = 0$.

Napisz program rozwiązujący takie równanie.

Jeżeli $a = 0$, to równanie nie jest kwadratowe.

Jeżeli $a \neq 0$, to należy obliczyć wartość wyrażenia $\Delta = b^2 - 4ac$ i rozpatrzyć trzy przypadki:

1. $\Delta < 0$ — równanie jest sprzeczne, nie posiada rozwiązań.
2. $\Delta = 0$ — równanie posiada jedno rozwiązanie $x = \frac{-b}{2a}$.
3. $\Delta > 0$ — są dwa rozwiązania $x_1 = \frac{-b + \sqrt{\Delta}}{2a}$ i $x_2 = \frac{-b - \sqrt{\Delta}}{2a}$.

pas

```
procedure z25(a, b, c: double);
var x1, x2, delta: double;
begin
   if a = 0 then writeLn('Równanie nie jest kwadratowe.')
   else
   begin
      delta := b * b - 4 * a * c;
      if delta < 0 then writeLn('Równanie nie ma rozwiązań.')
      else
         if delta = 0 then
         begin
            x1 := -b / (2 * a); writeLn('Równanie ma jedno rozwiązanie:
            ↪x = ', x1);
         end
         else
         begin
            x1 := (-b - sqrt(delta)) / (2 * a); x2 := (-b+sqrt(delta)) /
            ↪(2 * a);
            writeLn('Równanie ma dwa rozwiązania: x1 = ', x1, ', x2 = ', x2);
         end;
   end;
end;
```

cpp

```
void z25(double a, double b, double c)
{
   double x1, x2, delta;
   if(a == 0) cout << "Równanie nie jest kwadratowe.";
   else
   {
      delta = b * b - 4 * a * c;
      if(delta < 0) cout << "Równanie nie ma rozwiązań.";
      else
         if (delta==0)
         {
            x1=-b/(2*a); cout<<"Równanie ma jedno rozwiązanie. x="<<x1;
         }
         else
         {
            x1 = (-b - sqrt(delta)) / (2 * a); x2 = (-b + sqrt(delta)) /
            ↪(2 * a);
            cout << "Równanie ma dwa rozwiązania. x1 = " << x1 << ",
            ↪x2 = " << x2;
         }
   }
   cout << endl;
}
```

js

```
function z25(a, b, c)
{
   var x1, x2, delta;

   if(a == 0)
      console.log("Równanie nie jest kwadratowe");
   else
   {
      delta = b * b - 4 * a * c;
      if(delta < 0) console.log("Równanie nie ma rozwiązań.");
      else
         if(delta == 0)
         {
            x1 = -b / (2 * a);
            console.log("Równanie ma jedno rozwiązanie: x = ", x1);
         }
      else
      {
         x1 = (-b -Math.sqrt(delta)) / (2 * a); x2 = (-b + Math.sqrt(delta)) /
         ↪(2 * a);
         console.log("Równanie ma dwa rozwiązania: x1 = ", x1, " x2 = ", x2);
      }
   }
}
```

py

```
def z25(a, b, c):    # None, gdy pierwiastki nie są rzeczywiste
   d = b ** 2 - 4 * a * c
   if d > 0:
      return (1.0 * (-b + d ** .5) / (2 * a), 1.0 * (-b - d ** .5) / (2 * a))
```

Zadanie 26. Układ równań

Układ trzech równań liniowych z trzema niewiadomymi ma postać

$$\begin{cases} a_1 x + b_1 y + c_1 z = d_1 \\ a_2 x + b_2 y + c_2 z = d_2 \\ a_3 x + b_3 y + c_3 z = d_3 \end{cases}$$

Napisz program rozwiązujący taki układ.

Rozwiążemy układ metodą wyznacznikową.

W tym celu obliczymy najpierw tzw. wyznacznik główny $W = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix}$,

stosując metodę Sarussa.

Dopisujemy dwie pierwsze kolumny, aby lepiej zrozumieć tę metodę:

$$+ \quad + \quad + \quad - \quad - \quad -$$

$$W = \begin{vmatrix} a_1 & b_1 & c_1 & a_1 & b_1 \\ a_2 & b_2 & c_2 & a_2 & b_2 \\ a_3 & b_3 & c_3 & a_3 & b_3 \end{vmatrix} \text{ wtedy}$$

$$W = a_1 \cdot b_2 \cdot c_3 + b_1 \cdot c_2 \cdot a_3 + c_1 \cdot a_2 \cdot b_3 - b_1 \cdot a_2 \cdot c_3 - a_1 \cdot c_2 \cdot b_3 - c_1 \cdot b_2 \cdot a_3$$

Jeżeli $W \neq 0$, to układ ma jedno rozwiązanie, którym jest trójka liczb (x, y, z).

W celu znalezienia tych liczb obliczamy wyznaczniki W_x, W_y, i W_z.

Wyznacznik W_x powstaje przez zastąpienie pierwszej kolumny wyznacznika W kolumną wyrazów wolnych, czyli

$$W_x = \begin{vmatrix} d_1 & b_1 & c_1 \\ d_2 & b_2 & c_2 \\ d_3 & b_3 & c_3 \end{vmatrix} = d_1 \cdot b_2 \cdot c_3 + b_1 \cdot c_2 \cdot d_3 + c_1 \cdot d_2 \cdot b_3 - b_1 \cdot d_2 \cdot c_3 - d_1 \cdot c_2 \cdot b_3 - c_1 \cdot b_2 \cdot d_3$$

Wyznacznik W_y powstaje przez zastąpienie drugiej kolumny wyznacznika W kolumną wyrazów wolnych, czyli

$$W_y = \begin{vmatrix} a_1 & d_1 & c_1 \\ a_2 & d_2 & c_2 \\ a_3 & d_3 & c_3 \end{vmatrix} = a_1 \cdot d_2 \cdot c_3 + d_1 \cdot c_2 \cdot a_3 + c_1 \cdot a_2 \cdot d_3 - d_1 \cdot a_2 \cdot c_3 - a_1 \cdot c_2 \cdot d_3 - c_1 \cdot d_2 \cdot a_3$$

Wyznacznik W_z powstaje przez zastąpienie trzeciej kolumny wyznacznika W kolumną wyrazów wolnych, czyli

$$W_z = \begin{vmatrix} a_1 & b_1 & d_1 \\ a_2 & b_2 & d_2 \\ a_3 & b_3 & d_3 \end{vmatrix} = a_1 \cdot b_2 \cdot d_3 + b_1 \cdot d_2 \cdot a_3 + d_1 \cdot a_2 \cdot b_3 - b_1 \cdot a_2 \cdot d_3 - a_1 \cdot d_2 \cdot b_3 - d_1 \cdot b_2 \cdot a_3$$

Jeśli wyznacznik główny $W \neq 0$ i wszystkie wyznaczniki W_x, W_y, i W_z są równe zero, to układ ma nieskończenie wiele rozwiązań.

Jeśli wyznacznik główny $W \neq 0$ i chociaż jeden z wyznaczników W_x, W_y, i W_z jest różny od zera, to układ jest sprzeczny.

Przykład:

$a_1 = -1$, $b_1 = 2$, $c_1 = 5$, $d_1 = 1$, $a_2 = 0$, $b_2 = -3$, $c_2 = 5$, $d_2 = 0$, $a_3 = 3$, $b_3 = 0$, $c_3 = 0$, $d_3 = 2$

jedno rozwiązanie = (0,66…, 0,33…, 0,2);

$a_1 = -1$, $b_1 = 2$, $c_1 = 5$, $d_1 = 1$, $a_2 = 2$, $b_2 = -4$, $c_2 = -10$, $d_2 = 0$, $a_3 = 3$, $b_3 = 0$, $c_3 = 0$, $d_3 = 2$

brak rozwiązań;

$a_1 = -1$, $b_1 = 2$, $c_1 = 5$, $d_1 = 1$, $a_2 = 2$, $b_2 = -4$, $c_2 = -10$, $d_2 = -2$, $a_3 = 3$, $b_3 = 0$, $c_3 = 0$, $d_3 = 2$

nieskończenie wiele rozwiązań.

pas

```
procedure z26(a1, b1, c1, d1, a2, b2, c2, d2, a3, b3, c3, d3: double);
var x, y, z, w, wx, wy, wz: double;
begin
   w: = a1*b2*c3+b1*c2*a3+c1*a2*b3-b1*a2*c3-a1*c2*b3-c1*b2*a3;
   wx: = d1*b2*c3+b1*c2*d3+c1*d2*b3-b1*d2*c3-d1*c2*b3-c1*b2*d3;
   wy: = a1*d2*c3+d1*c2*a3+c1*a2*d3-d1*a2*c3-a1*c2*d3-c1*d2*a3;
   wz: = a1*b2*d3+b1*d2*a3+d1*a2*b3-b1*a2*d3-a1*d2*b3-d1*b2*a3;

   if w <> 0 then
   begin
      x := wx / w; y := wy / w; z := wz / w;
      writeLn('Rozwiązanie układu: (', x, ', ', y, ', ', z, ')');
   end
   else
      if ((wx = 0) and (wy = 0) and (wz = 0)) then
         writeLn('Układ ma nieskończenie wiele rozwiązań.')
      else
         writeLn('Układ jest sprzeczny.');
end;
```

cpp

```
void z26(double a1, double b1, double c1, double d1,
            double a2, double b2, double c2, double d2,
            double a3, double b3, double c3, double d3)
{
   double x, y, z, w, wx, wy, wz;

   w = a1*b2*c3+b1*c2*a3+c1*a2*b3-b1*a2*c3-a1*c2*b3-c1*b2*a3;
   wx = d1*b2*c3+b1*c2*d3+c1*d2*b3-b1*d2*c3-d1*c2*b3-c1*b2*d3;
   wy = a1*d2*c3+d1*c2*a3+c1*a2*d3-d1*a2*c3-a1*c2*d3-c1*d2*a3;
   wz = a1*b2*d3+b1*d2*a3+d1*a2*b3-b1*a2*d3-a1*d2*b3-d1*b2*a3;

   if(w != 0)
   {
      x = wx / w; y = wy / w; z = wz / w;
      cout << "Rozwiązanie układu: (" << x << ", " << y << ", " << z << ")";
```

```
    }
    else
       if(wx == 0 && wy == 0 && wz == 0)
          cout << "Układ ma nieskończenie wiele rozwiązań.";
       else
          cout << "Układ jest sprzeczny.";

    cout << endl;
}
```

js

```
function z26(a1, b1, c1, d1, a2, b2, c2, d2, a3, b3, c3, d3)
{
   var y, z, w, wx, wy, wz;
   w = a1*b2*c3+b1*c2*a3+c1*a2*b3-b1*a2*c3-a1*c2*b3-c1*b2*a3;
   wx = d1*b2*c3+b1*c2*d3+c1*d2*b3-b1*d2*c3-d1*c2*b3-c1*b2*d3;
   wy = a1*d2*c3+d1*c2*a3+c1*a2*d3-d1*a2*c3-a1*c2*d3-c1*d2*a3;
   wz = a1*b2*d3+b1*d2*a3+d1*a2*b3-b1*a2*d3-a1*d2*b3-d1*b2*a3;
   if(w != 0)
   {
      x = wx / w; y = wy / w; z = wz / w;
      console.log("Rozwiązanie układu: ", x, ", ", y, ", ", z);
   }
   else
      if(wx == 0 && wy == 0 && wz == 0)
         console.log("Układ ma nieskończenie wiele rozwiązań.");
      else
         console.log("Układ jest sprzeczny.");
}
```

py

Zadanie wykonamy dwoma sposobami: metodą wyznacznikową oraz metodą eliminacji, i porównamy ich efektywności. Metoda wyznacznikowa została omówiona dokładnie na początku rozwiązania zadania.

Do rozwiązania przez wyznaczniki najwygodniej wymagać danych wejściowych w postaci czterech list trójelementowych — współczynników przy x w kolejnych równaniach, współczynników przy y w kolejnych równaniach, współczynników przy z w kolejnych równaniach, wyrazów wolnych w kolejnych równaniach. Definiujemy funkcję liczącą wyznacznik macierzy trójwymiarowej zadanej przez listę kolumn.

```
def det(kx ,ky ,kz ):
   return kx[0] * (ky[1] * kz[2] - kz[1] * ky[2]) + kx[1] * (ky[2] *
   ↪kz[0] - kz[2] * ky[0]) + kx[2] * (ky[0] * kz[1] - kz[0] * ky[1])
```

Obliczamy wyznaczniki W_x, W_y, i W_z i wyznacznik główny W, wywołując powyższą funkcję z odpowiednimi kolumnami.

Ostatecznie funkcja rozwiązująca układ równań ma postać:

```
def z26(kx, ky, kz, kww):
   W = 1.0 * det(kx ,ky ,kz)
   if W == 0:
      return None
   return (det(kww, ky, kz) / W, det(kx, kww, kz) / W, det(kx, ky, kww) / W)
```

W rozwiązywaniu przez eliminację z pierwszego równania obliczamy x i wstawiamy do równań następnych, następnie powtarzamy tę procedurę

dla kolejnych niewiadomych i kolejnych równań. Po dwóch podstawieniach znamy wartość z i możemy je wyeliminować z kolejnych równań, co daje wartość y, po wyeliminowaniu y mamy wartość x. Podstawienie x do drugiego równania oznacza odjęcie od drugiego równania pierwszego pomnożonego przez współczynnik a_2 / a_1, co zagwarantuje wyeliminowanie x z drugiego równania. Dlatego w tym rozwiązaniu najlepiej wymagać danych wejściowych jako listy trzech czteroelementowych wierszy. Przy eliminowaniu $x - a$ z pozostałych równań trzeba dwa razy dzielić. Efektywniej byłoby podzielić pierwsze równanie przez a_1, a następnie tylko mnożyć przy odejmowaniu. Możemy się spotkać z problemem, że $a_1 = 0$, dlatego równania najpierw posortujemy, aby wybrać to, w którym przy x stoi współczynnik o największej wartości bezwzględnej. Jeżeli po posortowaniu $a_1 = 0$, oznacza to, że układ jest nieokreślony, i zwracamy wartość `None`. Sortowanie listy w Pythonie realizuje funkcja `sorted`, której argumentami są lista wejściowa i funkcja sortująca. W naszym przypadku funkcją sortującą będzie funkcja przypisująca wierszowi wartość bezwzględną jego i-tego elementu (współczynnika równania przy i-tej zmiennej), czyli `lambda x : abs(x[i])`.

Po wyeliminowaniu x z pozostałych równań zostajemy z układem dwóch równań na dwie niewiadome i dla tego prostszego układu powtarzamy procedurę.

Całość funkcji wygląda następująco:

```
def z26p(l_wierszy):
   n = len(l_wierszy)
   for i in range(n):
      l_wierszy [i:]= sorted (l_wierszy [i:], key = lambda x : abs(x[i]))[:: -1]
      if l_wierszy [i][i] == 0:
         return None
      l_wierszy [i]=[l_wierszy[i][k] / l_wierszy[i][i] for k in range(n+1)]
      for j in range(i + 1, n):
         l_wierszy [j][i+1:] = [l_wierszy[j][k] - l_wierszy[i][k] *
         ↪l_wierszy[j][i] for k in range (i + 1, n + 1)]
         l_wierszy[j][i] = 0
   for i in range (n -1, -1, -1):
      for j in range (i):
         l_wierszy[j][n] -= l_wierszy[i][n] * l_wierszy[j][i]
         l_wierszy[j][i] = 0
   return [i[n] for i in l_wierszy ]
```

Pomiar czasu wykonania pokazuje, że metoda eliminacji działa ok. 3 razy wolniej. Zastanówmy się, jak stosunek wydajności będzie się zmieniać dla większych układów równań. Żeby obliczyć wyznacznik dla n równań, należy wykonać $n!$ mnożeń. Wyznaczników obliczamy $(n + 1)$ i wykonujemy n dzieleń. Razem wykonujemy $(n + 1)! + n$ kosztownych operacji. W metodzie eliminacji, skalując równanie pierwsze, wykonujemy n dzieleń, a następnie eliminując x, wykonujemy $n(n - 1)$ mnożeń. Dla drugiej zmiennej jest to już $n - 1$ dzieleń i $(n - 1)(n - 2)$ mnożeń. Dla wszystkich równań będzie to $n(n + 3)/2$ dzieleń i $n(n\ 2 - 1)/3$ mnożeń. Następnie w fazie podstawiania podstawienie ostatniej zmiennej i wyeliminowanie jej z równań wymaga $n - 1$ mnożeń (tyle, ile pozostałych równań). Postępując tak dalej, wykonamy $n(n - 1)/2$ mnożeń. Razem wykonamy $n(n + 1)(n + 2)/3$ kosztownych operacji. W metodzie eliminacji wraz ze wzrostem rozmiaru układu równań koszt rośnie o wiele wolniej.

Zadanie 27. Liczba wyrazów w zdaniu

Dane jest zdanie, np. „Ala ma kota.”. Program ma obliczyć liczbę wyrazów w zdaniu.

Zakładamy, że na końcu zdania nie znajduje się spacja, natomiast liczba spacji między wyrazami może być dowolna, co w odniesieniu do pisma drukowanego nie jest błędem gramatycznym, ale edytorskim. Tak samo po znakach przestankowych, np. po przecinku, powinna wystąpić spacja.

pas

```
function z27(s: string): integer;
var d, ile, i: integer;
begin
   d := length(s);
   ile := 0;
   for i := 1 to d - 1 do
      if s[i] =' ' then if s[i + 1] <> ' ' then inc(ile);
   z27 := ile + 1;
end;
```

cpp

Odczyt zdania powinien odbyć się za pomocą funkcji `getline()`, gdyż standardowe wprowadzenie przez `cin` spowoduje odczytanie tylko części zdania, mianowicie do pierwszej napotkanej spacji.

```
main(){string t; getline(cin,t); cout << wyrazy(t); return 0;}
```

A oto rozwiązanie:

```
double z27(string s)
{
   int i, d, ile = 0;
   d = s.length();
   for(i = 0; i < d - 1; i++)
      if(s[i] == ' ') if(s[i + 1] != ' ') ile++;
   return ile + 1;
}
```

js

W przypadku wywołania funkcji przy użyciu przycisku, gdy parametrem funkcji jest tekst, należy wpisać go w apostrofach.

```
<input type = "button" value = "wyrazy" onclick = "z27('ALA ma kota') ">
```

A oto rozwiązanie:

```
function z27(s)
{
   var i, d, ile = 0;
   d = s.length;
   for(i = 0; i < d - 1; i++)
      if(s[i] == ' ') if(s[i + 1] != ' ') ile++;
   console.log("Liczba wyrazów = ", ile);
}
```

py

Metoda `split` typu `string` przyjmuje za argument podłańcuch będący separatorem i zamienia łańcuch w listę podłańcuchów zawartych pomiędzy separatorami. Jeżeli chcemy rozbić zdanie na słowa, należy podzielić

zawierający je łańcuch ze względu na znak spacji: `return len(s.split(' '))`. Ponieważ odstęp między słowami może zawierać więcej niż jedną spację, powinniśmy odrzucić z listy podłańcuchów elementy puste.

```
def z27(s):
   return len(filter(bool, s. split(' ')))
```

Wykorzystaliśmy tutaj fakt, że łańcuchy puste rzutują się na `False`, a niepuste na `True`. Rozwiązanie działa ponad 10 razy szybciej niż rozwiązanie, w którym zastosowano pętlę `for`.

```
def z27(s):
   slowo = False
   liczba = 0
   for i in s:
      if i != ' ' and not slowo:
         slowo = True
         liczba += 1
      elif i == ' ' and slowo:
         slowo = False
   return liczba
```

Zadanie 28. Palindromy

Palindrom to wyrażenie, które czytane od końca (wspak) brzmi tak samo jak wyjściowe.

Np. „A to kanapa pana Kota”.

Napisz program, który sprawdza, czy podane zdanie jest palindromem.

Nie przewidujemy sprawdzania poprawności wprowadzanego zdania. Zakładamy, że teksty nie zawierają liter ze znakami diaktrycznymi („polskich ogonków”). Wyrazy oddzielone są spacjami. Mogą zawierać litery małe i duże, przecinki, kropki, znaki zapytania, wykrzykniki, myślniki, cyfry, ale nie mogą zawierać żadnych innych znaków.

Przykłady:

Zakopane na pokaz.

A to kanapa pana Kota.

Tolo ma samolot.

Ma tarapaty ta para tam.

A to idiota.

Mamuta tu mam.

I car komedia i demokraci!

Ela tropi portale.

A to kiwi zdziwi kota!

Popija rum as, samuraj i pop.

O, mam karabin i barak mamo!

A ma boki Fafik? Oba ma!

Able was I ere I saw Elba

No, it is open on one position.

pas

```
function z28(s: string): boolean;
var nowy: string;
   czy: boolean;
   i, dlug1, dlug2, kod : integer;
begin
   nowy := '';  dlug2 := 0;
   dlug1 := length(s);
   czy := true;
   for i := 1 to dlug1 do
      if (s[i] <> ' ')
            and (s[i] <> '.')
            and (s[i] <> ',')
            and (s[i] <> '-')
            and (s[i] <> '?')
            and (s[i] <> '!')
      then
      begin
           nowy := nowy + s[i]; dlug2 := dlug2 + 1;
      end;

   for i := 1 to dlug2 do nowy[i]: = upcase(nowy[i]);

   for i := 1 to dlug2 do if nowy[i] <> nowy[dlug2 - i + 1] then czy := false;

   z28 := czy;
end;
```

cpp

```
bool z28(string s)
{
   string nowy = "";
   bool czy = true;
   int i, dlug1, dlug2 = 0, kod;
   char znak;
   dlug1 = s.length();

   for(i = 0; i < dlug1 ;i++)
      if ((s[i] != ' ')
            && (s[i] != '.')
            && (s[i] != ',')
            && (s[i] != '-')
            && (s[i] != '?')
            && (s[i] != '!'))
      {
         nowy = nowy + s[i]; dlug2 = dlug2 + 1;
      }

   for(i = 0; i < dlug2; i++)
   {
      znak = toupper(nowy[i]);
      nowy[i] = znak;
   }
   for(i = 0; i < dlug2; i++) if (nowy[i] != nowy[dlug2 - i - 1]) czy = false
   return czy;
}
```

js

```
function z28(s)
{
   var czy = 1;
   var nowy = "";
   var i, dlug1, dlug2 = 0, kod; znak = '';

   dlug1 = s.length;
   for(i = 0; i < dlug1; i++)
   if (s[i] != ' ' && s[i] != '. ' && s[i] != ', ' && s[i] != '-' && s[i] !=
   ↪'? ' && s[i] != '! ')
   {
      nowy = nowy + s[i]; dlug2 = dlug2 + 1;
   }

   nowy = nowy.toUpperCase();

   for(i = 0; i < dlug2; i++) if(nowy[i] != nowy[dlug2 - i - 1]) czy = 0;

   if(czy == 1) console.log("PALINDROM");
   else console.log("To nie jest PALINDROM.")
}
```

py

Najpierw pozostawiamy w łańcuchu znakowym tylko litery, filtrując go poprzez funkcję `str.isalpha`, która zwraca `True` dla liter i `False` dla pozostałych znaków. Następnie zmieniamy wszystkie litery na małe, odwzorowując łańcuch za pomocą funkcji `str.lower`. Dostaniemy w wyniku listę. Odwrócenie listy uzyskujemy za pomocą składni pobierania wycinka `s[::1]`. Pozostaje je porównać.

```
def z28(s):
   s = map(str.lower, filter(str.isalpha, s))
   return s == s[:: -1]
```

Zauważmy, że sprawdzamy równości w parach: znak pierwszy i ostatni, drugi i przedostatni itp. Dlatego wystarczy wykonać tylko połowę z tych porównań, które wykonujemy do tej pory — pozostałe to te same porównania, z odwróconą kolejnością argumentów. Poprawiamy zatem kod do postaci:

```
def z28(s):
   s = map(str.lower, filter(str.isalpha, s))
   l = len(s) / 2
   return s[:l] == s[:l: -1]
```

Porównujemy pierwszą połowę (zaokrągloną w dół) oryginalnego łańcucha z pierwszą połową łańcucha odwróconego. Poprawiony kod działa dwukrotnie szybciej.

Zadanie 29. Szyfr Cezara

Szyfrowanie tekstów metodą Cezara.

W szyfrowaniu metodą Cezara każdemu znakowi tekstu jawnego odpowiada znak przesunięty o określoną liczbę znaków w alfabecie. Przyjmujemy, że jest to alfabet łaciński, który zawiera 26 liter: A, B, C, D, E, F, G, H, I, J, K, L, M,

N, O, P, Q, R, S, T, U, V, W, X, Y, Z. W metodzie tej nie rozróżnia się liter wielkich i małych, tak więc tekst, który będziemy podawać, będzie zamieniany na litery wielkie.

Po przesunięciu o 3 znaki litera A staje się literą D, litera B staje się literą E, a litera Z literą C itd.

Jest to więc bardzo prymitywny szyfr, który można w prosty sposób złamać, nie gwarantuje zatem żadnego bezpieczeństwa. W metodzie tej nie rozróżnia się liter wielkich i małych, tak więc tekst, który będziemy podawać, będzie zamieniany na litery wielkie. Przypisując każdej literze kod ASCII, możemy zapisać szyfrowanie za pomocą funkcji `jeżeli` arkusza kalkulacyjnego (Excel, Calc) w następujący sposób:

	A	B	C	D	E
1	65	A	78	78	N
2	66	B	79	79	O
3	67	C	80	80	P
4	68	D	81	81	Q
5	69	E	82	82	R
6	70	F	83	83	S
7	71	G	84	84	T
8	72	H	85	85	U
9	73	I	86	86	V
10	74	J	87	87	W
11	75	K	88	88	X
12	76	L	89	89	Y
13	77	M	90	90	Z
14	78	N	91	65	A
15	79	O	92	66	B
16	80	P	93	67	C
17	81	Q	94	68	D
18	82	R	95	69	E
19	83	S	96	70	F
20	84	T	97	71	G
21	85	U	98	72	H
22	86	V	99	73	I
23	87	W	100	74	J
24	88	X	101	75	K
25	89	Y	102	76	L
26	90	Z	103	77	M
27					
28	KLUCZ:	13			

Użyte funkcje:

- w komórce *B1*: `=znak(A1)`
- w komórce *C1*: `=A1+$B$28`
- w komórce *D1*: `=jeżeli(C1>90;C1-26;C1)`
- w komórce *E1*: `=znak(D1)`

Przykład:

Tekst jawny: Ala ma kota → (przekształcamy na wielkie litery)
ALA MA KOTA
Klucz: 13
Szyfrogram: NYN ZN XBGN

Rozszyfrowanie wykonujemy tym samym sposobem. Spacje pozostawiamy bez zmian.

pas

```
function z29(s: string; klucz: integer): string;

var szyfr: string; c, d, i: integer;
begin
   d := length(s);
   szyfr := '';
   for i := 1 to d do s[i] := upcase(s[i]);
   for i := 1 to d do
      if s[i] = ' ' then szyfr := szyfr +  ' '
      else
      begin
         c := ord(s[i]) + klucz; if c>90 then c := c - 26;
         szyfr := szyfr + char(c);
      end;
   z29 := szyfr;
end;
```

cpp

```
string z29(string s, int klucz)
{
   int c, d, i;
   string szyfr = "", z = "";
   d = s.length();
   for(i =  0; i < d; i++) s[i] = toupper(s[i]);

   for(i = 0; i < d; i++)
      if(s[i] == ' ') szyfr = szyfr + " ";
      else
      {
         c = s[i] + klucz; if(c > 90) c = c - 26;
         szyfr = szyfr + char(c);
      }
   return szyfr;
}
```

js

```
function z29(s, klucz)
{
   var c, d, szyfr = "";
   var z = "";
   d =  s.length;
   s = s.toUpperCase();
   for(i = 0; i < d; i++)
      if(s[i] == ' ') szyfr = szyfr + " ";
      else
      {
         c = s.charCodeAt(i) +  klucz; if(c > 90) c = c - 26;
         szyfr = szyfr + String.fromCharCode(c);
      }
   return szyfr;
}
```

py

W szyfrze Cezara powinniśmy wykonać cykliczne przesunięcie wszystkich elementów alfabetu o *n* pozycji, gdzie *n* jest kluczem. Wielkie litery mają kody ASCII od 65 do 90, litery małe mają kody ASCII od 98 do 122. W zależności od tego, czy litera jest wielka, czy mała, dokonujemy zmiany jej kodu ASCII w tych zakresach. Ewentualne znaki spoza tych zakresów nie podlegają zmianie. Następnie łączymy uzyskane znaki w jeden łańcuch.

```
def z30(s, n):
   return ''.join([chr((ord(i) - 97 + n) % 26 + 97) if i.islower() else
   chr((ord(i) - 65 + n) % 26 + 65) if i.isupper() else i for i in s])
```

Jeżeli chcemy wykonać to samo dla polskiego alfabetu, argument powinien być ciągiem znaków, a nie bajtów. Możliwość tę daje typ *unicode*. Obiekt typu *unicode* tworzymy tak jak `string`, ale poprzedzamy go literką `u`. W standardowym kodowaniu *utf8* polskie znaki mają długość 2 bajtów, stąd:

```
>>> print len('żółw')
7
>>> print len(u'żółw')
4
```

Pierwszy wynik to liczba bajtów, drugi to liczba znaków.

Tworzymy ciągi liter dużych i liter małych. Obrotu cyklicznego dokonujemy nie w zakresach kodów ASCII, ale w indeksach powyższych ciągów:

```
def z29(s, n):
   a = u'aąbcćdeę fghijkl łmnńoóprsś tuwyz źż'
   A = u'AĄBCĆDEĘ FGHIJKL ŁMNŃOÓPRSŚ TUWYZ ŹŻ'
   return ''.join([a[(a.index(i) + n) % 32] if i in a else A[(A.index(i) + n)
   ↪% 32] if i in A else i for i in s])
```

Zadanie 30. Szyfr XOR

Szyfrowanie tekstów metodą XOR.

Algorytm:

Operacja logiczna XOR:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

Szyfrowanie będzie polegać na zastosowaniu operacji XOR na kodach ASCII znaków wejściowych i klucza (hasła) będącego jednym znakiem. Odszyfrowanie nastąpi w ten sam sposób, tym samym algorytmem, z zastosowaniem takiego samego klucza. Zbudowany program będzie zatem służył jednocześnie do szyfrowania i odszyfrowywania. Istnieje jednak pewne niebezpieczeństwo. Operowanie na kodach ASCII za pomocą funkcji XOR może doprowadzić do sytuacji, w której otrzymamy tzw. znak sterujący. W rozszerzonym ASCII są to znaki o kodach od 0 do 31. Znaki sterujące są niedrukowalne i sterują pracą urządzeń oraz mogą powodować zmianę interpretacji innych znaków. Jeśli chcielibyśmy zakodować np. tekst „Ala ma kota” z kluczem „a”, to na wszystkich znakach zgodnych z tym kluczem w wyniku operacji „a” XOR „a” uzyskamy wartość `0` (`NULL`). Lepiej jest dla tekstów literowych użyć klucza w postaci cyfry.

Przykład:

Tekst do zaszyfrowania: kot

Klucz: Z

kod(k) = $107_{(10)} = 1101011_{(2)}$

kod(o) = $111_{(10)} = 1101111_{(2)}$

kod(t) = $116_{(10)} = 1110100_{(2)}$

kod(Z) = $90_{(10)} = 1011010_{(2)}$

Operacje XOR:

```
0 1 1 0 1 0 1 1     0 1 1 0 1 1 1 1     0 1 1 1 0 1 0 0
0 1 0 1 1 0 1 0     0 1 0 1 1 0 1 0     0 1 0 1 1 0 1 0
-----------------   -----------------   -----------------
0 0 1 1 0 0 0 1     0 0 1 1 0 1 0 1     0 0 1 0 1 1 1 0
```

$00110001_{(2)} = 49_{(10)} \rightarrow$ znak = 1

$00110101_{(2)} = 53_{(10)} \rightarrow$ znak = 5

$00101110_{(2)} = 46_{(10)} \rightarrow$ znak = . (kropka)

Zatem kodowanie tekstu „kot” z kluczem „Z” dało tekst „15.”

Tekst do zaszyfrowania: 15

Klucz: Z

kod(1) = $49_{(10)} = 110001_{(2)}$

kod(5) = $53_{(10)} = 110101_{(2)}$

kod(.) = $46_{(10)} = 101110_{(2)}$

kod(Z) = $90_{(10)} = 1011010_{(2)}$

Operacje XOR:

```
0 0 1 1 0 0 0 1     0 0 1 1 0 1 0 1     0 0 1 0 1 1 1 0
0 1 0 1 1 0 1 0     0 1 0 1 1 0 1 0     0 1 0 1 1 0 1 0
-----------------   -----------------   -----------------
0 1 1 0 1 0 1 1     0 1 1 0 1 1 1 1     0 1 1 1 0 1 0 0
```

$01101011_{(2)} = 107_{(10)} \rightarrow$ znak = k

$01101111_{(2)} = 111_{(10)} \rightarrow$ znak = o

$01110100_{(2)} = 116_{(10)} \rightarrow$ znak = t

Zatem odkodowanie tekstu „15.” z kluczem „Z” dało tekst „kot”.

pas

```
function z30(s: string; klucz: char): string;
var szyfr: string; i: integer;
begin
   szyfr := ''; d := length(s);
   for i := 1 to d do szyfr := szyfr + char(ord(klucz) xor ord(s[i]));
   z30 := szyfr;
end;
```

cpp

```
string z30(string s, char klucz)
{
   string szyfr = "";
   int i, d;
   d = s.length();
   for(i =  0; i< d; i++) szyfr = szyfr + (char)(s[i]^ klucz);
   return szyfr;
}
```

js

```
function z30(s, klucz)
{
   var szyfr = "";
   var i, d, z;
   d = s.length;
   for(i = 0; i < d; i++)
   {
      z  = s.charCodeAt(i) ^ klucz.charCodeAt(0);
      szyfr = szyfr + String.fromCharCode(z);
   }
   return szyfr;
}
```

py

Argumentami funkcji szyfrującej są klucz — liczba z przedziału 0 – 255 — oraz ciąg znaków do zakodowania (precyzyjniej: bajtów, szyfrować będziemy bajt po bajcie, nie we wszystkich kodowaniach znakowi odpowiada jeden bajt). Dla pobranego jednobajtowego znaku z łańcucha znajdujemy jego reprezentację liczbową za pomocą funkcji wbudowanej `ord`, następnie na tej liczbie wykonujemy operację `xor` z kluczem `n`, używając operatora bitowego `^`, i przekształcamy z powrotem na znak za pomocą funkcji wbudowanej `chr`. Postępujemy tak po kolei dla wszystkich znaków z łańcucha. Uzyskaną listę zaszyfrowanych bajtów łączymy w łańcuch za pomocą metody `join` wywołanej dla łącznika będącego pustym łańcuchem.

```
def z29(n, s):
   return ''.join([chr(ord(i) ^ n) for i in s])
```

Rozdział 3.

Dodatek

T-komputer

T-komputer posiada pamięć zewnętrzną o pojemności 100 bajtów w postaci taśmy z danymi, która jest obsługiwana przez głowicę odczytu/zapisu, pamięć operacyjną wystarczającą do wykonania podanej sekwencji instrukcji oraz pamięć pomocniczą w postaci 1 bajta.

Taśma z danymi

A	9	4	R	T	U	3	6	F													
0	1	2	3	4	5	6	7	8	.	.	.	.	.	.	.	.	.	.	.	97	98	99

Głowica (po uruchomieniu T-komputera znajduje się na pozycji `0`)

Pamięć podręczna (pusta po włączeniu komputera)

T-komputer posiada wewnętrzny język programowania **PLOCZR**.

Język PLOCZR zawiera następujące instrukcje:

1. `P` — przesunięcie głowicy o jedną pozycję w prawo.
2. `L` — przesunięcie głowicy o jedną pozycję w lewo.
3. `O` — odczyt znaku z aktualnej pozycji do pamięci podręcznej.
4. `C` — czyszczenie komórki na aktualnej pozycji (zawartość `NULL`).
5. `Z` — zapis znaku z pamięci podręcznej na aktualną pozycję na taśmie.
6. `R` — przewinięcie taśmy do początku (na pozycję `0`).

Po odczycie (`O`), czyszczeniu komórki (`C`) i zapisie (`Z`) głowica pozostaje na swoim miejscu!

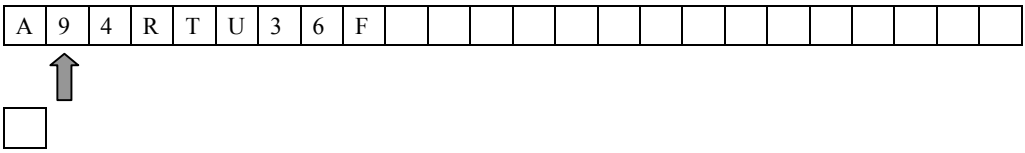
Jeśli głowica jest na końcu taśmy, to po wykonaniu instrukcji `P` taśma jest automatycznie przewinięta.

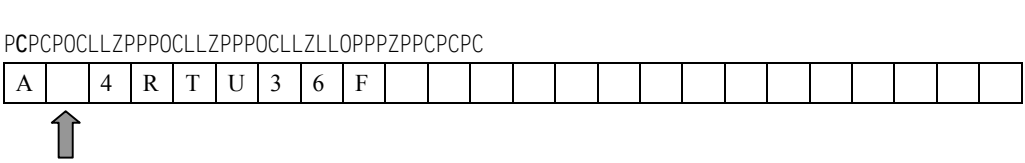
Przykład programu:

```
PCPCPOCLLZPPPOCLLZPPPOCLLZLLOPPPZPPCPCPC
```

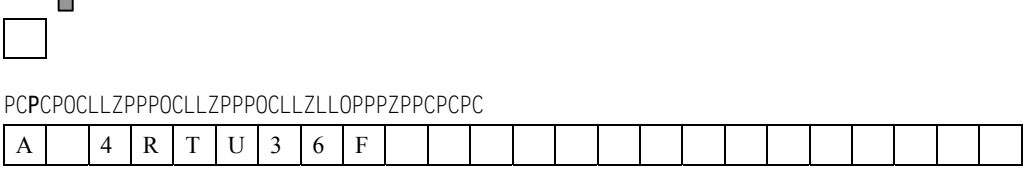
Etapy wykonania:

PCPCPOCLLZPPPOCLLZPPPOCLLZLLOPPPZPPCPCPC

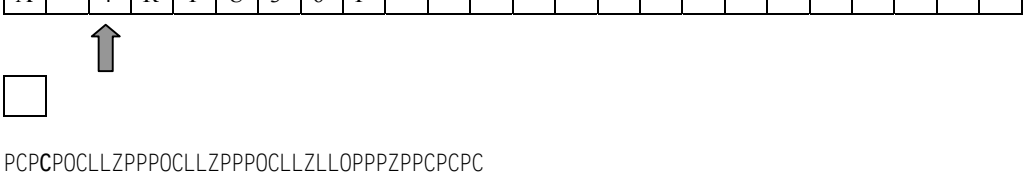




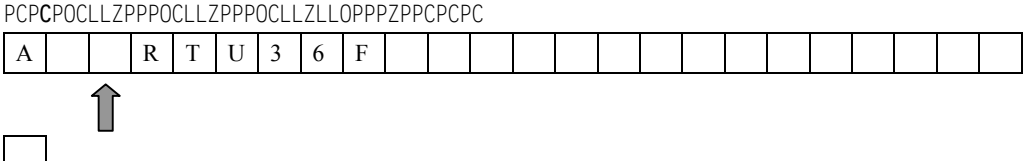
PC**P**CPOCLLZPPPOCLLZPPPOCLLZLLOPPPZPPCPCPC



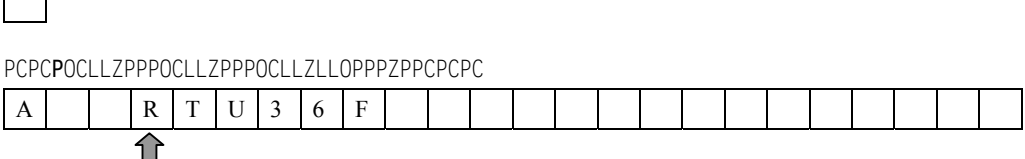
PCP**C**POCLLZPPPOCLLZPPPOCLLZLLOPPPZPPCPCPC



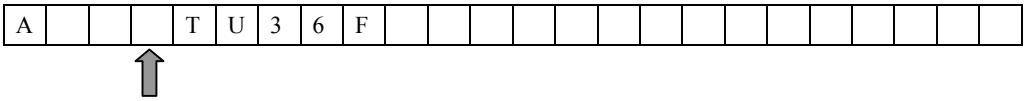
PCPC**P**OCLLZPPPOCLLZPPPOCLLZLLOPPPZPPCPCPC



PCPCP**O**CLLZPPPOCLLZPPPOCLLZLLOPPPZPPCPCPC

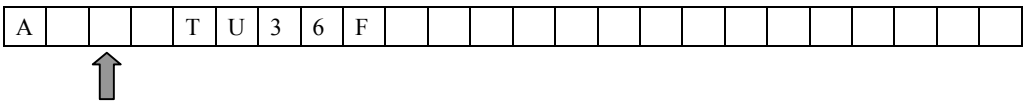


PCPCPO**C**LLZPPPOCLLZPPPOCLLZLLOPPPZPPCPCPC



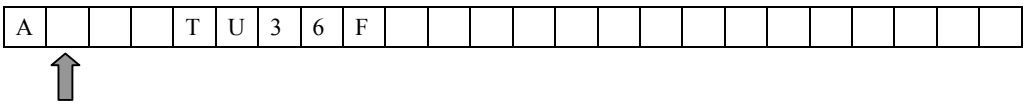
R

PCPCPOC**L**LZPPPOCLLZPPPOCLLZLLOPPPZPPCPCPC



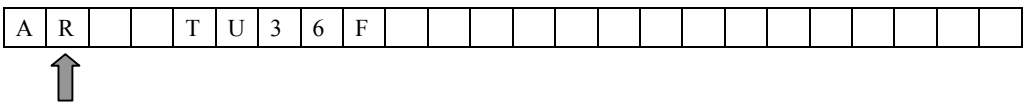
R

PCPCPOCL**L**ZPPPOCLLZPPPOCLLZLLOPPPZPPCPCPC



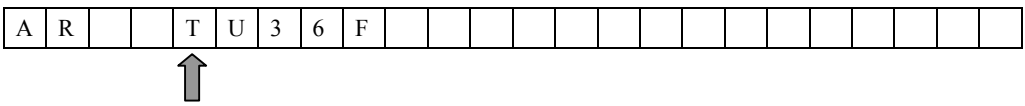
R

PCPCPOCLL**Z**PPPOCLLZPPPOCLLZLLOPPPZPPCPCPC



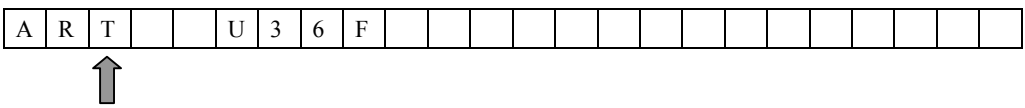
R

PCPCPOCLLZ**PPPO**CLLZPPPOCLLZLLOPPPZPPCPCPC



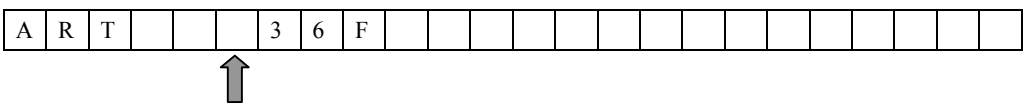
T

PCPCPOCLLZPPPO**CLLZ**PPPOCLLZLLOPPPZPPCPCPC



T

PCPCPOCLLZPPPOCLLZ**PPPOC**LLZLLOPPPZPPCPCPC



U

PCPCPOCLLZPPPOCLLZPPPOC**LLZ**LLOPPPZPPCPCPC

A	R	T	U			3	6	F													

U

PCPCPOCLLZPPPOCLLZPPPOCLLZ**LLOPPPZ**PPCPCPC

A	R	T	U	R		3	6	F													

R

PCPCPOCLLZPPPOCLLZPPPOCLLZLLOPPPZ**PPCPCPC**

A	R	T	U	R																	

R

Zadanie

Napisz program, który będzie emulatorem T-komputera i wykona program napisany w języku PLOCZR.

Rozwiązanie zadania pozostawiamy czytelnikowi.

Rozdział 4.

Trochę historii

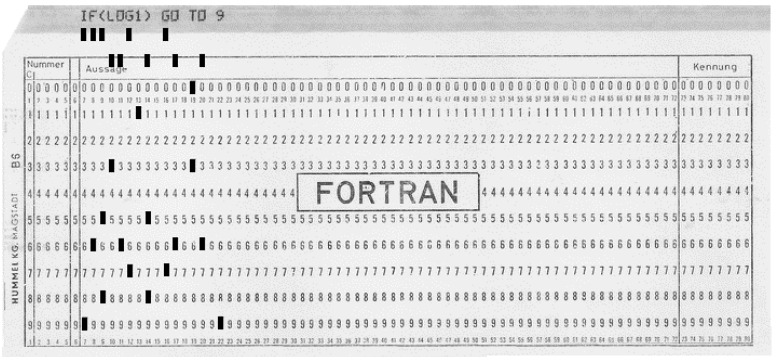
Programowanie z „myszką”

Słownik języka polskiego PWN powiedzenie „trącić myszką” tłumaczy jako „być przestarzałym, staroświeckim”. Wyraz „myszka” w tytule ma podobne znaczenie i nie dotyczy, jak mogłoby się wydawać, urządzenia zewnętrznego komputera, najczęściej połączonego z nim kablem przypominającym właśnie ogon tego niewątpliwie sympatycznego zwierzątka. Co ciekawe, urządzenie to wprowadzono do powszechnego użytku dopiero w roku 1983 wraz z komputerem Apple Macintosh. Mysz (komputerowa) nie trąci więc zbytnio samą sobą. Nie przyjęła się natomiast nazwa „kot” w odniesieniu do manipulatora kulkowego (ang. *trackball*), który wygląda jak odwrócona mysz, ale z dużo większą kulką, na której spoczywa ręka operującego nim użytkownika. Jak by nie patrzeć, kota to nie przypominało! Zanim jednak wymyślono mysz, podstawowym urządzeniem operatorskim służącym do komunikowania się z komputerem i przekazywania sygnałów była przez długi czas tylko klawiatura, natomiast teksty poleceń i komunikaty ukazywały się na papierze drukarki, a nie jak obecnie na ekranie monitora.

Samo przygotowanie danych i tekstów programów źródłowych napisanych w jakimś języku wymagało sporo pracy. Bardzo długo jedynym dostępnym nośnikiem informacji był papier. Dane i instrukcje dziurkowane były na specjalnych kartach lub taśmach wykonanych właśnie z papieru. Dlatego nazywało się je perforowanymi — nie mylić z peryferyjnymi. Tysiące ton tego typu wyrobów przewinęło się przez ośrodki komputerowe — był to „złoty” okres dla firm, które je produkowały. Za pomocą urządzenia wielkości dzisiejszego biurka wykonywało się dziurki w papierowych kartach, odpowiadające poszczególnym bitom i bajtom informacji. Urządzenia te były wyposażone w klawiaturę podobną do klawiatury współczesnego komputera (rysunek 4.1).

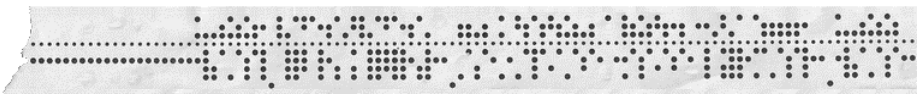
Każda z prostokątnych kart miała ścięty jeden róg (rysunek 4.2), aby nie dopuścić do ich przemieszczenia i ewentualnego odwrócenia podczas odczytu. Można z grubsza przyjąć, że jedna karta odpowiadała jednej instrukcji języka programowania. Umieszczenie kodu źródłowego programu wymagało więc użycia od kilkudziesięciu do kilkuset egzemplarzy takiego nośnika. Wystarczyła jedna usterka techniczna podczas perforowania, polegająca na przesunięciu otworu albo zagięciu karty, aby nie można było uruchomić programu. Należało wówczas znaleźć wadę i ponownie wykonać dziurkowanie.

Rysunek 4.1. *Urządzenie do perforowania kart*



Rysunek 4.2. *Papierowa karta perforowana*

Przypomnijmy, że pomysł wykorzystania „dziurek” do kodowania informacji nie jest niczym nowym. Został zastosowany na masową skalę już w roku 1890, kiedy to Amerykanin Herman Hollerith postanowił zautomatyzować prace przy powszechnym spisie ludności, jaki w Stanach Zjednoczonych przeprowadzany jest co 10 lat. Wyobraźmy sobie ogrom danych, jakie należało zapamiętać. Hollerith zastosował karty perforowane i wymyślił też elektryczne urządzenie, które potrafiło odczytywać informacje z tych kart, a nawet wykonać ich sortowanie. Innym papierowym nośnikiem, ale o podobnym charakterze była taśma dziurkowana (rysunek 4.3). Nie było jednak ogólnej normy na sposób kodowania i liczbę dziurek.



Rysunek 4.3. *Taśma perforowana*

Jednym ze standardów był pięciościeżkowy system zapisywania znaków, tzn. do zapisania każdego znaku potrzeba było 5 miejsc do ewentualnego wykonania dziurki. Łatwo policzyć, że ten sposób pozwalał na zakodowanie tylko 2 do potęgi 5, tzn. 32 znaków alfanumerycznych, i dlatego przed ciągiem liter albo cyfr czy też innych znaków (np. *, *A* itp.) umieszczany był specjalny symbol sygnalizujący rodzaj zapisu. Taki sposób kodowania kojarzy się nam z alfabetem Morse’a, który także wykorzystuje kombinacje „dziurek” do zapisywania liter i cyfr.

Informacja jest obecnie najbardziej poszukiwanym i cenionym towarem. Jeszcze zanim powstały pierwsze komputery, istniał zawsze problem jej zapamiętywania, przechowywania i przekazywania. Nie wszyscy może wiedzą, że oprócz **informatyki**, która zajmuje się przede wszystkim metodami i technikami przetwarzania informacji, istnieje jeszcze inna nauka o nazwie **teoria informacji**, będąca działem matematyki stosowanej i zajmująca się kodowaniem i przekazywaniem danych, z uwzględnieniem różnego rodzaju zakłóceń podczas transmisji. Co ciekawe, dział ten wyrósł z praktycznych potrzeb telekomunikacji, a jego początki można datować na lata trzydzieste XIX wieku, kiedy to Samuel Morse rozpoczął prace nad telegrafem elektrycznym. Dla przeciętnego użytkownika komputera najistotniejsza jest przede wszystkim pojemność pamięci zewnętrznych oraz szybkość przekazywania danych. Człowiek, używając mowy, przekazuje informacje z szybkością ok. 3,8 słowa na sekundę, a więc przekazuje w przybliżeniu 30 bitów informacji na jedną sekundę. Zależy to oczywiście od wyboru słów i sprawności mówiącego. Znam jednak kilka osób (są to kobiety, notabene bardzo sympatyczne), które wynik ten poprawiłyby co najmniej dwukrotnie. Prędkość przesyłania danych na dysku twardym wynosić może 188 Mb (megabitów) na sekundę, a więc mogą być przekazywane przeszło 6 milionów razy szybciej niż informacja mówiona. Ulotność tej ostatniej powodowała, że od zarania dziejów człowiek poszukiwał optymalnego nośnika i narzędzi do jej przechowywania, zaczynając od ścian jaskini i kamiennego rylca, przez długi okres i do tej pory — stosując papier i druk, a kończąc na optycznych pamięciach z użyciem lasera.

Zastosowanie tablic

„Układ, rubryka, spis, rejestr danych liczbowych lub innych, rozmieszczonych na arkuszu w określonym porządku" — tak określona jest tablica (tabela) w *Słowniku PWN*. Mówi się, że „tablica jest strukturą umożliwiającą przechowywanie danych w posegregowanych szufladkach". W programowaniu używa się najczęściej tablic dwuwymiarowych do zapamiętania pewnych sekwencji danych, które można zapisać w postaci tabeli. Dostęp do poszczególnych elementów uzyskuje się przez podanie nazwy tablicy i miejsca określonego przez tzw. indeksy — w przypadku dwóch wymiarów jest to numer kolumny, czyli miejsce w wierszu tabeli, oraz numer wiersza, czyli miejsce w kolumnie. Przykładem zastosowań tablic może być np. arkusz kalkulacyjny, stosowany do różnych obliczeń, ale także do przechowywania określonych danych.

Fizycznym modelem dwuwymiarowej tablicy może być natomiast ekran monitora. Pierwsze monitory zastosowane w komputerach osobistych były monochromatyczne, tzn. pozwalały wyświetlać tylko jeden kolor (nie licząc koloru tła!) — zielony lub bursztynowy — i pokazywały w 20 liniach po 40 znaków lub w 25 liniach po 80 znaków. Mikrokomputer ZX Spectrum pozwalał na wyświetlenie rysunku w rozdzielczości 256×176, a Atari520ST nawet 640×400 punktów (pikseli). Angielski wyraz *pixel* to skrót od słów *picture element* i oznacza najmniejszy element obrazu wyświetlanego na ekranie monitora, umożliwiający jego programowe adresowanie. Tablice i im podobne struktury odgrywają ogromną rolę w programowaniu w każdym języku.

Tablice zostały wielokrotnie użyte w rozwiązaniach zadań w tej książce.

Języki programowania

„Pierwsze programy musiały być składane bezpośrednio z operacji oferowanych przez konkretną maszynę, często w notacji binarnej. Każdy model używał innego zestawu komend, co ograniczało przenośność. W późniejszych latach zaprojektowano pierwsze asemblery, gdzie programista mógł wpisywać instrukcje w formacie tekstowym z wykorzystaniem zapisu symbolicznego zamiast numeru rozkazu, np. ADD X, TOTAL. W 1954 roku stworzony został pierwszy język programowania wysokiego poziomu, FORTRAN, gdzie programiści mogli bezpośrednio formułować wyrażenia matematyczne w podobnym stylu, do jakiego jesteśmy przyzwyczajeni: y = x^2 + 5*x – 7."

— Wikipedia

Ogromną zaletą języka Fortran była prostota składni, szybkość działania programów i wysoki stopień standaryzacji.

W następnych latach XX wieku powstawały kolejne języki programowania, jak Algol, Lisp, Cobol, Plan, PL/I, Basic, Pascal jako następca Algola. Najbardziej znaną wersją Pascala był Turbo Pascal napisany specjalnie dla komputerów IBM. W latach 1969 – 73 tworzony był i udoskonalany język C, ale jego standard został zatwierdzony dopiero w roku 1989. Na język C zostało przepisane jądro systemu Unix.

Profesor Włodzisław Duch w swojej książce *Fascynujący świat programów komputerowych* napisał, że o języku C krążyła opinia, iż „pisanie programów w C jest jak szybki taniec z brzytwami na śliskiej podłodze". Powstały też języki specjalnie do nauczania dzieci, jak np. Logo, który doczekał się też wersji polskiej. Programowanie z wykorzystaniem grafiki „żółwia" zostało też zastosowane w Elektronicznym Laboratorium Informatyki „MultiPlus" profesora Macieja M. Sysły, zalecanym do nauczania podstaw programowania przez Ministerstwo Edukacji Narodowej.

Oto przykład kodu w języku Basic dla komputera ZX Spectrum z procesorem Z80:

```
5      FOR i=1 TO 32*6-4
10     PRINT "Halo"
20     RANDOMIZE USR 3582
30     NEXT i
```

Kod ten powoduje przesuwanie linii na ekranie poprzez odwołanie do procedury systemowej.

Język Basic jest stosowany do tej pory w wersji Visual Basic np. do tworzenia makr w arkuszu kalkulacyjnym Excel. W języku VB poszczególne instrukcje muszą znajdować się w osobnych liniach, gdyż nie używa się separatorów poleceń i w odróżnieniu od protoplasty, czyli Basica, nie trzeba numerować linii programu.

Podany poniżej przykład makra powoduje wyświetlenie w arkuszu od komórek *A1* do *J10* wielu przeróżnych kolorów, których kody z palety barw Excela zapisane są w obszarze *L1:U10*.

Przykład:

	L	M	N	O	P	Q	R	S	T	U
1	48	16	31	51	7	36	32	38	2	30
2	45	35	49	8	35	49	12	2	29	14
3	11	12	9	2	47	22	5	31	11	20
4	31	34	31	7	46	45	31	41	14	54
5	11	1	5	44	41	55	43	37	16	10
6	27	29	20	27	37	21	9	7	48	49
7	24	45	31	37	32	50	8	43	3	20
8	46	27	32	51	12	52	35	20	41	40
9	47	21	32	46	1	43	8	52	39	16
10	50	8	25	24	52	55	6	20	38	38

```
Sub mozaika()
   Dim w As Integer
   Dim k As Integer
   Selection.Interior.Pattern = 1
   For w = 1 To 10
      For k = 1 To 10
         Cells(w, k).Select
         Selection.Interior.ColorIndex = Range("L1").Offset(w - 1, k - 1)
      Next
   Next
End Sub
```

Literatura

Algorytmy

- Thomas Carmen, Charles Leiserson, Ronald Rivest, Clifford Stein, *Wprowadzenie do algorytmów*, PWN, Warszawa 2013.
- Maciej M. Sysło, *Algorytmy*, WSiP, Warszawa 2005.
- Harel David, Feldman Yishai, *Rzecz o istocie informatyki. Algorytmika*, WNT, Warszawa 2008.
- Niklaus Wirth, *Algorytmy + struktury danych = programy*, WNT, Warszawa 2004.
- Piotr Wróblewski, *Algorytmy, struktury danych i techniki programowania*, Helion, Gliwice 2009.

Pascal

- Andrzej Marciniak, *Turbo Pascal 7.0*, NAKOM, Poznań 1997.
- Erik Wischniewski, *100 przepisów w języku Turbo Pascal*, NAKOM, Poznań 1995.
- Norbert Kilen, *Z Turbo Pascalem w głąb systemu*, LYNX-SFT, Warszawa 1994.

C++

- Jerzy Grębosz, *Symfonia C++ standard*, Edition 2000, Kraków 2005.
- Bruce Eckel, *Thinking in C++. Edycja polska*, Helion, Gliwice 2002.
- Alan R. Neibauer, *Języki C i C++*, HELP, Michałowice 1994.
- Herbert Schildt, *Sztuka programowania C++*, Helion, Gliwice 2004.

JavaScript

- Dawid Borycki, Jacek Matulewski, *JavaScript i jQuery*, Helion, Gliwice 2014.
- Larry Ullman, *Nowoczesny język JavaScript*, Helion, Gliwice 2013.
- Marcin Lis, *JavaScript. Ćwiczenia praktyczne. Wydanie III*, Helion, Gliwice 2013.

Python

- Peter Norton i inni, *Python od podstaw*, Helion, Gliwice 2006.
- Mark Lutz, David Ascher, *Python. Wprowadzenie*, Helion, Gliwice 2010.
- *https://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie.*

Polecamy

- Włodzisław Duch, *Fascynujący świat programów komputerowych*, Nakom, Poznań 1997.

Słowniczek informatyczny

Abort	przerwanie
Abs	wartość bezwzględna
Accept	zatwierdzanie
Active	aktywny
Add	dodawanie
Adjust	wyrównanie
Advanced	zaawansowany
Alias	nazwa zastępcza
Align	wyrównanie
All	wszystko
Angle	kąt
Animate	animacja
Applet	niewielki program użytkowy
Append	dołączanie
Arc	łuk
Array	tablica z danymi
AssignFile	przydzielanie pliku
BackGround	tło
BackGroundColor	kolor tła
Backup	kopia zapasowa
Bar	pasek, słupek
Batch	plik wsadowy
Begin	początek programu, procedury, bloku
Bitmap	grafika punktowa (bitmapa)
Binary	dwójkowy (binarny)
Bold	pogrubienie
Boolean	typ logiczny (`true` albo `false`)

Border	brzeg, granica
Box	pole (np. pole wyboru)
Break	przerwanie
Brush	pędzel
Build	budowanie
Button	przycisk
ButtonClick	naciśnięcie klawisza
Byte	bajt (= 8 bitów)
Cancel	anulowanie, usuwanie
Canvas	„płótno”, pewien obszar obiektu graficznego
Caption	opis, nagłówek
Change	zmiana
Char	znak
CheckBox	przycisk wyboru
Checked	wybranie, zaznaczenie
Circle	okrąg
Class	klasa (w sensie kategoria)
Clear	czyszczenie
ClearSelection	wybrane miejsce czyszczenia
Click	naciśnięcie przycisku myszy, „kliknięcie”
Close	zamknięcie
CloseFile	zamknięcie pliku
Color	kolor
ColorGrid	kolor siatki
Column	kolumna
ComboBox	pole kombinowane, lista rozwijana
Compile	kompilacja (tłumaczenie kodu źródłowego)
Compute	obliczanie
Const	stała
Copy	kopiowanie
Cos	cosinus
Create	utworzenie
CreateForm	utworzenie formularza
Cursor	kursor, wskaźnik
Cut	wycięcie
Data	dane
DataBase	baza danych
DataSource	źródło danych

Date	data
DbClick	podwójne kliknięcie
Decimal	dziesiętny
Delete	kasowanie
Destination	cel, miejsce przeznaczenia
Detect	wykrywanie, detekcja
DialogBox	pole dialogowe
Display	ekran monitora
Do	wykonanie, robienie
Double	podwójne
DownTo	w dół aż do…
Draw	rysowanie
Edit	edycja
Eject	wysunięcie
Ellipse	elipsa
Else	w przeciwnym razie
Enabled	umożliwiony, aktywny
Encoding	kodowanie
End	koniec
Error	błąd
Events	zdarzenia
Execute	wykonanie
Exit	wyjście
Exp	liczba *e* do potęgi
Extension	rozszerzenie
False	fałsz
Field	pole
File	plik
Find	znalezienie
First	pierwszy
Fixed	ustalony, utrwalony
Flip	przerzucenie
FloodFill	wylanie farby, wypełnienie
Floor	zaokrąglenie liczby w dół
Font	czcionka
For	dla
ForeGroundColor	kolor pierwszoplanowy
Form	formularz

Format	formatowanie
Found	odnaleziony
Frame	ramka, szkielet
Free	uwolnienie
FreeMem	zwolnienie pamięci
Function	funkcja
Get	otrzymać
GoTo	iść do…
Grid	siatka
GroupBox	pole grupy
Handle	uchwyt, np. uchwyt pliku
Height	wysokość
Help	pomoc
Hexadecimal	szesnastkowy
Hide	schowany
If	jeżeli
Ignore	zignorowanie
Image	obraz, rysunek
ImageList	lista obrazów
Implementation	implementacja, wdrożenie
Index	indeks
Info	informacja
Input	wejście
InputBox	pole wejściowe
Insert	wstawienie
Integer	całkowity
Interface	interfejs, złącze
Interrupt	przerwanie
IntToStr	„z całkowitego na string”
Italic	kursywa, pochylenie
Items	pozycja
Key	klawisz
KeyDown	klawisz wciśnięty
KeyPress	klawisz naciśnięty
KeyUp	klawisz puszczony
Kind	rodzaj, klasa
Label	etykieta, napis
Large	wielki

Left	lewy
Length	długość
Line	linia
LineTo	linia do…
List	lista
ListBox	pole listy
Ln	logarytm naturalny
Load	ładowanie, wprowadzenie
LoadFromFile	wprowadzenie z pliku
LongInt	liczba całkowita długa
Main	główny
Mask	maska, matryca
Max	maksymalny
MaxLength	maksymalna długość
MaxValue	maksymalna wartość
MediaPlayer	odtwarzacz multimediów
Memo	pamięć
Menu	menu, lista
Method	metoda, sposób
Min	minimalny
MinValue	minimalna wartość
Mode	tryb, rodzaj
MouseClick	kliknięcie myszą
MouseDown	naciśnięcie przycisku myszy
MouseMove	przesunięcie myszy
MouseUp	puszczenie przycisku myszy
MoveTo	przesunięcie do…
Name	nazwa
New	nowy
New Application	nowy program komputerowy (aplikacja)
Next	następny
None	żaden
Object	obiekt
ObjectInspector	zarządca obiektów
Octal	ósemkowy
Open	otwarcie
Output	wyjście
Page	strona

PaintBox	pole rysunkowe
Palette	paleta
Panel	panel
Paste	wklejenie
Pen	pióro, kreślenie
Pi	liczba π
Picture	rysunek, obraz
Pie	wycinek koła
Pixel	punkt graficzny
Play	odtworzenie
Pointer	wskaźnik
Polygon	wielokąt
Pos	pozycja (skrót od *position*)
Position	pozycja, miejsce
PreView	podgląd
Print	drukowanie
Printer	drukarka
Procedure	procedura
ProgressBar	pasek postępu
Project	projekt
Prompt	znak zachęty
Properties	właściwości
Put	umieszczenie
RadioButton	przycisk opcji
Read	czytanie
Real	rzeczywisty
Record	rekord lub zapisywanie
Rectangle	prostokąt
Refresh	odświeżenie
Register	rejestr
Rename	zmiana nazwy
Resize	zmiana rozmiaru
Retry	powtórzenie próby
Rewrite	powtórzenie zapisu
RGB	standard kodowania kolorów *Red*, *Green*, *Blue*
Right	prawy
Round	zaokrąglony
RoundRect	zaokrąglony prostokąt

Row	rząd, wiersz
Run	uruchomienie
Scale	skala
Screen	ekran
Scroll	przewijanie
ScrollBar	pasek przewijania
Search	wyszukiwanie
Seek	szukanie
Select	wybranie
Set	ustawienie (ustalenie)
SetFocus	ustawienie punktu aktywności
Shape	kształt, figura
ShortInt	liczba całkowita krótka
ShortString	łańcuch tekstowy krótki
Show	pokaz
Sin	sinus
Size	rozmiar
Small	mały
Sort	sortowanie
Sqr	podnoszenie do kwadratu
Sqrt	obliczanie pierwiastka kwadratowego
Square	kwadrat, czworobok
Step	krok
Stretch	rozciąganie
StretchDraw	rysowanie z rozciąganiem
String	łańcuch tekstowy
Style	styl
Table	tablica
Tan	tangens
Text	tekst
TextColor	kolor tekstu
TextHeight	wysokość tekstu
TextOut	wyprowadzenie tekstu
TextWidth	szerokość tekstu
Then	wtedy
Time	czas
Timer	zegar
Title	tytuł

To	do
Tool	narzędzie
ToolBar	pasek narzędzi
ToolButton	przycisk narzędzi
Top	góra, szczyt
True	prawda
Type	typ
UnderLine	podkreślenie
UpDate	aktualizacja
UperCase	wielkie litery
Uses	użycie
Value	wartość
Var	zmienne
View	wyświetlenie, widok
Visible	widzialność
Void	nieokreślony, typ nieznany
While	dopóki
Width	szerokość
Window	okno
Word	liczba naturalna dwubajtowa
Write	zapis

Skorowidz

Ł

N

O

P

R

S

T

U

W