

# Informatyka

## ▼ Od problemu do programu

### ▼ Obliczanie pierwiastka kwadratowego wzorem Herona

#### Warto wiedzieć:

Prawdziwa jest zależność :

Jeśli  $|a - x/a| < 0,001$ , to  $|\sqrt{x} - a| < 0.001$ . Wynika to stąd, że  $a$  i  $x/a$  są przybliżeniami  $\sqrt{x}$  z góry i z dołu (jeśli  $a > \sqrt{x}$ , to  $x/a < \sqrt{x}$  i na odwrót)

Jeśli odległość między  $a$  i  $x/a$  jest mniejsza od  $0,001$ , to tym bardziej odległość  $a$  od  $\sqrt{x}$  musi być mniejsza od  $0,001$

## Algorytm Herona

**1** za  $a$  przyjmij dowolną liczbę większą od  $0$

**2** dopóki wartość  $|a - x/a|$  jest większa niż  $0,001$ , powtarzaj kroki **3** i **4**

**3** przyjmij za  $a$  średnią arytmetyczną  $a$  i  $x/a$

**4** oblicz różnicę  $a$  i  $x/a$

## ▼ Wybieranie największej z 3 podanych liczb

## Algorytm wybierania największej z trzech liczb a, b, c

1. sprawdź, czy  $a > b$  i  $a > c$ .

Jeśli tak, to a jest największą liczbą, więc zamień miejscami wartości zmiennych a i c i przejdź do kroku 3

2. sprawdź, czy  $b > c$ .

Jeśli tak, to b jest największą liczbą więc zmień miejscami wartości zmiennych b i c

3. sprawdź, czy  $a + b > c$ .

Jeśli tak, to wypisz komunikat "TAK". W przeciwnym wypadku wypisz "NIE"

Jednym z kluczowych elementów algorytmu jest zamiana wartości dwóch zmiennych np. a i b. Można do tego celu użyć zmiennej pomocniczej np. p, która przechowa tymczasowo wartość jednej ze zmiennych.

Liczby wskazują kolejność wykonywanych podstawień (przypisywania wartości zmiennym).

```
1 a = int(input())
2 b = int(input())
3 c = int(input())
4 if a > b and a > c: # a jest największą liczbą
5     p = a
6     a = c
7     c = p
8 else:
9     if b > c: # b jest największą liczbą
10        p = b
11        b = c
12        c = p
13
14 if a + b > c:
15     print("TAK")
16 else:
17     print("NIE")
18
```

## ▼ Mnożenie pisemne liczb

### Specyfikacja:

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

Wynik: iloczyn liczb a i b

### Algorytm w postaci listy kroków:

Krok 1: Zapisz liczbę a

Krok 2: Zapisz liczbę b poniżej liczby a tak, żeby jej cyfry jedności, dziesiątek, setek itd. znajdowały się odpowiednio pod cyframi jedności, dziesiątek, setek itd. liczby a

Krok 3: Dla każdej z cyfr liczby b, począwszy od cyfry jedności, traktowanych jako liczby jednocyfrowe, wykonaj kroki 4 i 5

Krok 4: Oblicz iloczyn bieżącej liczby jednocyfrowej i liczby a.

Krok 5: Jeśli iloczyn jest pierwszym cząstkowym iloczynem, to zapisz go poniżej liczby b (jedności pod jednościami, dziesiątki pod dziesiątkami itd.). Kolejne iloczyny zapisuj z odpowiednim przesunięciem w lewo, tj. cyfrę jedności iloczynu zapisz pod cyfrą dziesiątek poprzedniego iloczynu, cyfrę dziesiątek iloczynu zapisz pod cyfrą setek poprzedniego iloczynu itd.

Krok 6: Zsumuj cząstkowe iloczyny.

## ▼ Ilość dzielników (Podział na równoliczne grupy)

### Algorytm – wersja 1:

- Krok 1: Ustaw licznik\_dzielników na 0
- Krok 2: Dla liczb od 2 do połowy n wykonuj krok 3
- Krok 3: Jeśli reszta z dzielenia n przez liczbę jest równa zero, powiększ wartość zmiennej licznik\_dzielników o 1

Zapiszmy inny algorytm rozwiązujący ten sam problem. Zwróć uwagę, że gdy znajdziemy jeden dzielnik, automatycznie znaleźliśmy drugi.

Jeśli liczba n dzieli się bez reszty przez liczbę  $d_1$ , to dzieli się także przez liczbę

$$d_2 = n : d_1 \quad (n = d_1 \cdot d_2).$$

Należy tylko uważać na sytuację, gdy n jest kwadratem jakiejś liczby, czyli  $d_1 = d_2$ . Wówczas trzeba zwiększyć wartość zmiennej licznik\_dzielników o jeden, a nie o dwa

### Algorytm – wersja 2:

- Krok 1: Ustaw licznik\_dzielników na 0
- Krok 2: Ustaw dzielnik d na 2
- Krok 3: Do póki  $d \cdot d < n$ , wykonuj kroki 4 i 5
- Krok 4: Jeśli reszta z dzielenia n przez d jest równa zero, powiększ licznik\_dzielników o 2
- Krok 5: Powiększ d o jeden
- Krok 6: Jeśli  $d \cdot d = n$ , powiększ licznik\_dzielników o 1

Zapis tego algorytmu w pseudokodzie może być następujący:

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

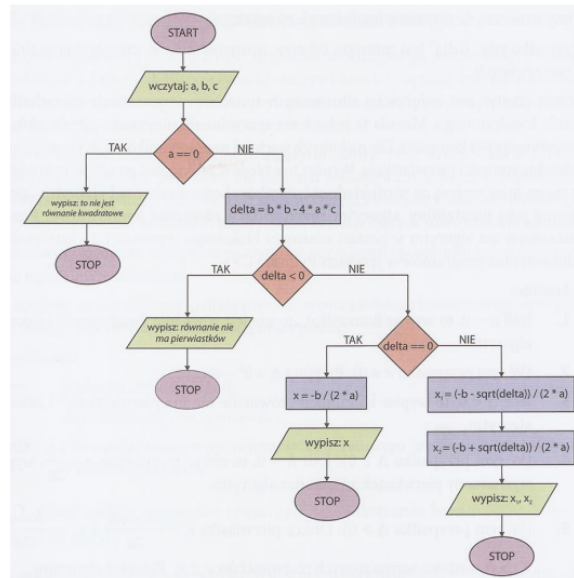
## ▼ Podział liczby na cyfry

### Algorytm wersja 1 – podział liczb na cyfry

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

## ▼ Niestabilny algorytm obliczania równania kwadratowego

## ▼ Schemat blokowy



## ▼ Lista kroków

### Lista kroków:

**Krok 1:** Jeśli  $a = 0$ , to wypisz komunikat „to nie jest równanie kwadratowe” i zakończ algorytm.

**Krok 2:** (W tym przypadku  $a \neq 0$ ). Przypisz  $\Delta = b^2 - 4ac$ .

**Krok 3:** Jeśli  $\Delta < 0$ , to wypisz komunikat „równanie nie ma pierwiastków” i zakończ algorytm.

**Krok 4:** (W tym przypadku  $\Delta \geq 0$ ). Jeśli  $\Delta = 0$ , to oblicz pierwiastek  $x = \frac{-b}{2a}$ , wypisz wyznaczony pierwiastek  $x$  i zakończ algorytm

**Krok 5:** (W tym przypadku  $\Delta > 0$ ). Oblicz pierwiastki  $x_1 = \frac{-b - \sqrt{\Delta}}{2a}$   $x_2 = \frac{-b + \sqrt{\Delta}}{2a}$ , wypisz wartości wyznaczonych pierwiastków  $x_1$  i  $x_2$ . Zakończ algorytm.

## ▼ Pseudokod + Python

### Pseudokod:

```
jeżeli a = 0, to
    wypisz komunikat "to nie jest równanie kwadratowe"
w przeciwnym razie
    przypisz  $\Delta = b^2 - 4ac$ 
    jeżeli  $\Delta < 0$ , to
        wypisz komunikat "równanie nie ma pierwiastków"
    w przeciwnym razie
        jeżeli  $\Delta = 0$ , to
            oblicz pierwiastek  $x = \frac{-b}{2a}$ 
            wypisz x
        w przeciwnym razie
            oblicz pierwiastki  $x_1 = \frac{-b - \sqrt{\Delta}}{2a}$   $x_2 = \frac{-b + \sqrt{\Delta}}{2a}$ 
            wypisz  $x_1$  i  $x_2$ 
```

### Program w języku Python

```
from math import *

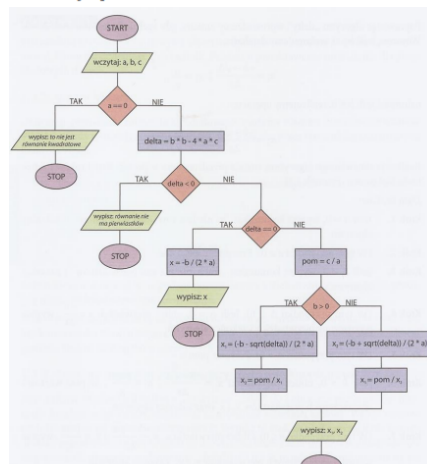
def rownanie_kwadratowe(a, b, c):
    if a == 0:
        return "to nie jest równanie kwadratowe"
    else:
        delta = b * b - 4 * a * c
        if delta < 0:
            return "równanie nie ma pierwiastków"
        elif delta == 0:
            x1 = -b / (2 * a)
            return x1
        else:
            x1 = (-b - sqrt(delta)) / (2 * a)
            x2 = (-b + sqrt(delta)) / (2 * a)
            return x1, x2

print(rownanie_kwadratowe(2, 1, 3))
```

## ▼ Stabilny algorytm obliczania równania kwadratowego

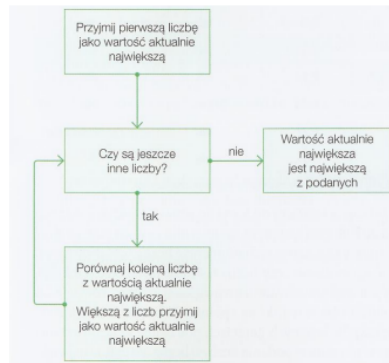
### Lista kroków:

- Krok 1:** Jeżeli  $a = 0$ , wypisz komunikat „to nie jest równanie kwadratowe” i zakończ algorytm.
- Krok 2:** (W tym przypadku  $a \neq 0$ ). Przypisz  $\Delta = b^2 - 4ac$ .
- Krok 3:** Jeżeli  $\Delta < 0$ , wypisz komunikat „równanie nie ma pierwiastków” i zakończ algorytm.
- Krok 4:** (W tym przypadku  $\Delta \geq 0$ ).  
Jeżeli  $\Delta = 0$ , oblicz pierwiastek  $x = \frac{-b}{2a}$ , wypisz wyznaczony pierwiastek  $x$  i zakończ algorytm.
- Krok 5:** (W tym przypadku  $\Delta > 0$ ). Przypisz  $\text{pom} = \frac{c}{a}$ .
- Krok 6:** Jeżeli  $b > 0$ , oblicz pierwiastki  $x_1 = \frac{-b - \sqrt{\Delta}}{2a}$ ,  $x_2 = \frac{\text{pom}}{x_1}$ , wypisz wartości wyznaczonych pierwiastków  $x_1$  i  $x_2$  oraz zakończ algorytm.
- Krok 7:** (W tym przypadku  $b \leq 0$ ). Oblicz pierwiastki  $x_2 = \frac{-b + \sqrt{\Delta}}{2a}$  i  $x_1 = \frac{\text{pom}}{x_2}$ , wypisz wartości wyznaczonych pierwiastków  $x_1$  i  $x_2$ . Zakończ algorytm.



## ▼ Systemy liczbowe i reprezentacja danych w komputerze

## ▼ Największa liczba z podanych



Lista kroków algorytmu wyznaczania największej liczby z ciągu liczb:

1. Jako wartość aktualnie największą (maksimum) przyjmij 0 i **wczytaj** pierwszą liczbę
2. **dopóki** ostatnia wczytana liczba jest różna od 0,  
    **wykonuj** krok 3 i krok 4
3. **jeśli** ostatnia wczytana liczba jest większa od maksimum,  
    to podstaw ją za maksimum
4. **wczytaj** kolejną liczbę z klawiatury
5. **wypisz** maksimum na ekranie

## ▼ Noty sędziowskie

1. Wczytaj pierwszą liczbę i przyjmij ją jako wartość minimum oraz jednocześnie wartość maksimum. Jako wartość sumy przyjmij wartość wczytanej liczby.
2. Powtórz cztery razy kroki od 3 do 6
3. Wczytaj kolejną liczbę
4. Zwiększ wartość sumy o wartość wczytanej liczby
5. Jeśli wczytana liczba jest mniejsza od minimum, to podstaw ją za minimum
6. Jeśli wczytana liczba jest większa od maksimum, to podstaw ją za maksimum
7. Wypisz na ekranie minimum, maksimum oraz sumę pomniejszoną o minimum i maksimum

## ▼ Zamiana binarnej na dziesiętną

## Zamiana liczby binarnej na liczbę dziesiętną

Zapis dziesiętnej liczby binarnej można wyznaczyć, sumując kolejne potęgi liczby 2 odpowiadające wagom kolejnych cyfr.

Algorytm w postaci listy kroków:

- 1 Ponumeruj cyfry liczby binarnej od prawej do lewej, zaczynając od 0
- 2 Każdą cyfrę potraktuj odpowiednio jako liczbę dziesiętną 0 albo 1 i pomnóż przez liczbę 2 podniesioną do takiej potęgi, jaki jest numer cyfry.
- 3 Zsumuj otrzymane iloczyny.

Zamianę liczby binarnej 100 na postać dziesiętną można zilustrować w następujący sposób:

$$100 = 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 = 0 \cdot 1 + 0 \cdot 2 + 1 \cdot 4 = 4$$

Schemat zamiany liczby binarnej na dziesiętną

## ▼ Zamiana dziesiętnej na binarną

### Specyfikacja

**Dane:** liczba całkowita (nieujemna) nie większa niż 255

**Wynik:** zapis binarny liczby całkowitej

Aby wyznaczyć postać binarną liczby dziesiętnej, możemy postępować w sposób przedstawiony wcześniej w tabeli. Wystarczy zapamiętywać kolejne ilorazy z dzielenia przez 2 oraz otrzymywane reszty z dzielenia. Dokładny zapis działania algorytmu przedstawia poniższa lista kroków.

- 1 Podziel liczbę przez 2, zapisz resztę z dzielenia i zapamiętaj iloraz (część całkowitą)
- 2 Dopóki ostatnio otrzymany iloraz jest większy od 0, powtarzaj kroki 3, 4 i 5
- 3 Wyznacz resztę z dzielenia ostatnio zapamiętanego ilorazu przez 2
- 4 Zapisz resztę z dzielenia przed ostatnią wcześniej zapisaną resztą
- 5 Podziel iloraz przez 2 i zapamiętaj część całkowitą wyniku jako nowy iloraz

## ▼ Liczenie występowania liter w zdaniu

### Specyfikacja

**Dane:** **słowo** – słowo złożone z liter alfabetu łacińskiego, **litera** – alfabetu łacińskiego

**Wynik:** liczba wystąpień litery w słowie.

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

Oto zapis algorytmu w pseudokodzie:

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

## ▼ Algorytmy zamiany reprezentacji liczb między systemami liczbowymi

### ▼ Zamiana na binarną

## Specyfikacja

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

Wynik:  $b$  – napis reprezentujący zapis binarny liczby  $d$

Zapis algorytmu w pseudokodzie może wyglądać następująco:

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

### ▼ Zamiana binarna na dziesiętna

Oto zapis algorytmu w pseudokodzie:

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

Implementacja tego algorytmu w języku C++ jest następująca.

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

### ▼ Zamiana na wybraną podstawę



Zapis algorytmu w pseudokodzie może być następujący:

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

Ograniczmy na razie podstawę systemu do wartości  $2 \leq p \leq 10$ .

Dzięki temu uwzględnimy tylko kody ASCII dla cyfr od 0 do 9. Zaczynają się one od 48, tzn. znak 0 ma kod 48, znak 1 kod 49 itd. Wystarczy więc do wartości wyrażenia  $d \bmod p$  dodać liczbę 48, aby otrzymać kod ASCII odpowiedniej cyfry.

Oto kod źródłowy programu.

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

## ▼ Zamiana z wybranej podstawy na dziesiętną

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

UWAGA:

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

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

## ▼ Szybkie podnoszenie do potęgi

Przykładowy algorytm w pseudokodzie:

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

UWAGA:

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

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

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

## ▼ Palindromy

### ▼ Czy słowo jest palindromem z użyciem pointerów

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

```
1  palindrom ← prawda
2  i ← 0
3  długość wyrazu - 1
4  dopóki palindrom oraz (i < j) wykonuj
5      jeśli wyraz[i] = wyraz[j] to
6          i ← i + 1
7          j ← j - 1
8      w przeciwnym przypadku palindrom ← fałsz
9  jeśli palindrom to wypisz „TAK”
10 w przeciwnym przypadku wypisz „NIE”
```

Na początku zakładamy, że badany wyraz jest palindromem, dlatego zmiennej palindrom typu logicznego przypisujemy wartość prawda.

Dodatkowo przyjmujemy, że słowo puste bądź słowo złożone tylko z jednej litery jest palindromem. Zmienna i wskazuje kolejne litery z początku wyrazu (wartość początkowa tej zmiennej jest równa 0, ponieważ znaki napisu indeksowane są od zera), a zmienna j – kolejne litery od końca wyrazu (wartość początkowa jest równa indeksowi ostatniej litery wyrazu, czyli długości wyrazu pomniejszonej o 1).

Jeśli w aktualnie porównywanej parze są takie same litery, wartość zmiennej i jest powiększana o 1, a zmiennej j pomniejszana o 1.

W przypadku, gdy litery w badanej parze są różne, zmienna palindrom przyjmuje wartość fałsz, co powoduje zakończenie wykonywania pętli.

Jeśli badany wyraz jest palindromem, instrukcje wykonują się, dopóki i < j.

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

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

## ▼ Czy słowa w zadaniu są palindromami

Oto zapis algorytmu w pseudokodzie:

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

## ▼ Liczby pierwsze

### ▼ Podstawowy, czy liczba jest pierwsza

## Test pierwszośc - algorytm najprostszy

Aby stwierdzić, czy jakaś liczba **n** jest pierwsza czy złożona, należy sprawdzić, czy ma ona jakikolwiek dzielnik różny od **1** i od niej samej. Można to zrobić, dzieląc **n** kolejno przez **2**, przez **3** itd. aż do **n - 1**.

Przed napisaniem programu komputerowego zapiszemy specyfikację.

#### Specyfikacja

**Dane:** liczba naturalna większa od 1.

**Wynik:** komunikat „**Liczba pierwsza**”, jeśli liczba jest liczbą pierwszą, komunikat „**Liczba złożona**” - w przeciwnym wypadku.

Poniżej znajduje się kod źródłowy programu **Test pierwszośc**.

```
1 def CzyPierwsza(n):  
2     for i in range(2, n):  
3         if n % i == 0:  
4             return 0  
5     return 1  
6  
7     liczba = int(input("Podaj liczbę: "))  
8  
9     if CzyPierwsza(liczba) == 1:  
10        print("Liczba pierwsza")  
11    else:  
12        print("Liczba złożona")  
13
```

### ▼ Bez liczb parzystych (Bo jedyna parzysta pierwsza to 2)

Aby zaimportować funkcję `sqrt`, należy na początku kodu dodać instrukcję:

```
from math import sqrt
```

Poniżej znajduje się tekst programu *Ulepszony test pierwszości*.

```
1  from math import sqrt
2
3  def CzyPierwsza(n):
4      if n == 2:
5          return 1
6      if n % 2 == 0:
7          return 0
8
9      pierwiastek = int(sqrt(n))
10     for i in range(3, pierwiastek+1, 2):
11         if n % i == 0:
12             return 0
13     return 1
14
15     liczba = int(input("Podaj liczbę:"))
16
17     if CzyPierwsza(liczba) == 1:
18         print("Liczba pierwsza")
19     else:
20         print("Liczba złożona")
21
```

▼ Czy pierwsza do pierwiastka z liczby

$$u \geq \sqrt{n}$$

W tym przypadku maksymalna liczba sprawdzeń jest mniejsza, dlatego skorzystamy z tego rozwiązania.

Zapis algorytmu w pseudokodzie może wyglądać następująco:

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

Na początku zakładamy, że jeśli liczba n jest większa od 1, to jest liczbą pierwszą – zmienna pierwsza typu logicznego przyjmuje wtedy wartość prawda. Dla n równego 1 zmienna pierwsza przyjmuje wartość fałsz. W pętli poszukujemy dzielnika d. Wartość początkowa tej zmiennej jest równa 2. Zwróć uwagę na warunek powtarzania pętli.

Zmienna pierwsza musi mieć wartość prawda (tzn. jeszcze nie został znaleziony dzielnik) i jednocześnie wartość zmiennej d nie może przekraczać wartości  $\sqrt{n}$ . Warunek  $d \leq \sqrt{n}$  zapisaliśmy jako  $d*d \leq n$ . Unikamy w ten sposób obliczania pierwiastka kwadratowego i wykonywania operacji na liczbach rzeczywistych. Jeśli zostanie znaleziony dzielnik d, zmiennej pierwsza zostanie przypisana wartość logiczna fałsz, co spowoduje zakończenie wykonywania pętli.

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

Kod źródłowy programu może wyglądać następująco:

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

▼ Czy pierwsza z użyciem  $6k+1$ ;  $6k-1$

Dzielników o postaci  $6i-1$  oraz  $6i+1$  będziemy szukać w przedziale  $[5; \sqrt{n}]$

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

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

Kod źródłowy programu sprawdzającego, czy liczba jest pierwsza – **algorytm 3** (5.2.cpp)

## ▼ Rozkład liczby na czynniki pierwsze

```

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

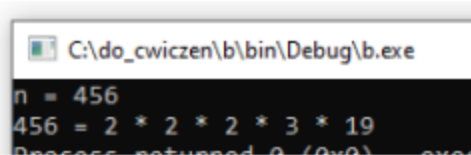
```

```

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

```

Kod źródłowy programu wypisującego czynniki pierwsze liczby (5.3.cpp)



```

C:\do_cwiczen\b\bin\Debug\b.exe
n = 456
456 = 2 * 2 * 2 * 3 * 19
Process returned 0 (0x0)

```

Przykład wywołania programu rozkładającego liczbę na czynniki pierwsze

## ▼ Liczby bliźniacze



```

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

```

część kodu źródłowego programu wypisującego liczby bliźniacze – definicja funkcji `Pierwsza` (sposób 2) 5.5.cpp)

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

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

```

17
18  int main()
19  {
20      int licznik=1, n, x=5;
21      cout<<"Podaj liczbę par: "; cin>>n;
22      cout<< 3 <<" " <<5<<endl;
23      while (licznik < n)
24      {
25          if (Pierwsza(x) && Pierwsza(x + 2))
26          {
27              cout<< x <<" " <<x + 2<<endl;
28              licznik++;
29          }
30          x = x + 6;
31      }
32      return 0;
33  }
34

```

II część kodu źródłowego programu wypisującego liczby bliźniacze – definicja funkcji `main` (5.5.cpp)

## ▼ Działania na liczbach w systemach innych niż dziesiętny

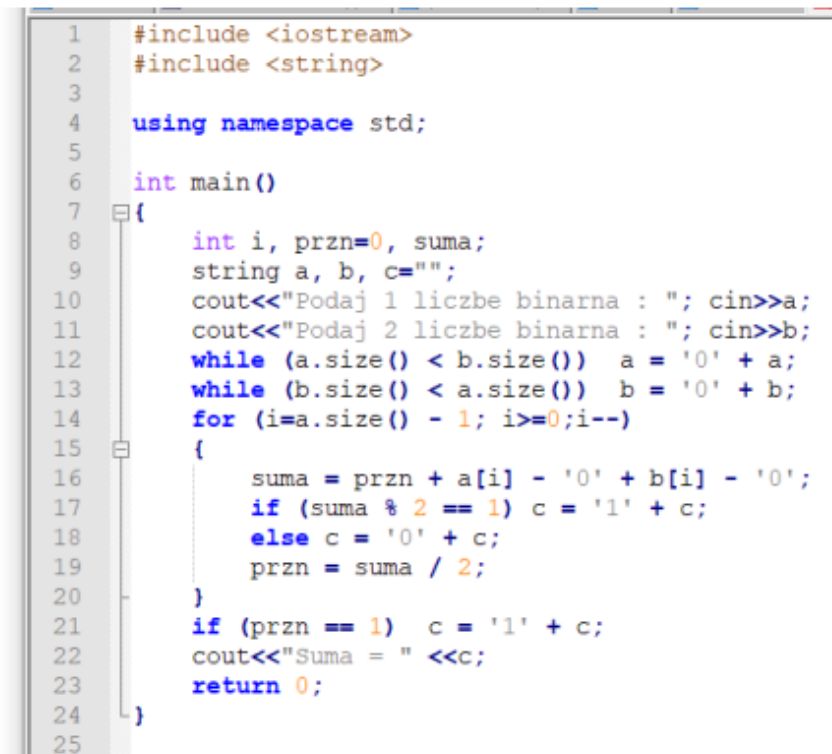
### ▼ Dodawanie w binarce

```

przn ← 0      //nadanie przeniesieniu wartości 0
c ← " "      //określenie wartości wyniku - napis pusty , dla liczb
              składających się z różnej liczby cyfr
              //dopisanie na początku krótszej liczby zer nieznaczących

dopóki długość a < długość b wykonuj a ← '0' + a
dopóki długość b < długość a wykonuj b ← '0' + b
//dodanie liczb a i b
dla i ← długość a - 1 , ... , 0 wykonuj
    suma ← przn + a[i] + b[i]
    jeśli suma mod 2 = 1 to c ← '1' + c
    w przeciwnym przypadku c ← '0' + c
    przn ← suma div 2
jeśli przn = 1 to c ← '1' + c

```



```

1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  int main()
7  {
8      int i, przn=0, suma;
9      string a, b, c="";
10     cout<<"Podaj 1 liczbe binarna : "; cin>>a;
11     cout<<"Podaj 2 liczbe binarna : "; cin>>b;
12     while (a.size() < b.size()) a = '0' + a;
13     while (b.size() < a.size()) b = '0' + b;
14     for (i=a.size() - 1; i>=0;i--)
15     {
16         suma = przn + a[i] - '0' + b[i] - '0';
17         if (suma % 2 == 1) c = '1' + c;
18         else c = '0' + c;
19         przn = suma / 2;
20     }
21     if (przn == 1) c = '1' + c;
22     cout<<"Suma = " <<c;
23     return 0;
24 }
25

```

## ▼ Dodawanie w różnych systemach

Oto pętla algorytmu realizująca dodawanie liczb a i b zapisana w pseudokodzie:

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

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

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

6.2.cpp

## ▼ Odejmowanie w binarce

```

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

```

6.4.cpp

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

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

Sprawdźmy działanie algorytmu dla liczby  $01111000_{U2}$ , odpowiadającej liczbie dziesiętnej 120. Po zamianie bitów na przeciwne otrzymujemy  $10000111$ , a po dodaniu 1 :  $100001000_{U2}$ , czyli w systemie dziesiętnym -120. Aby dodać liczbę 1 w naszym przykładzie, wystarczy zamienić – zaczynając od prawej strony – cyfry 1 na 0 do momentu, aż napotkamy pierwsze 0, które zamienimy na 1.

UWAGA:

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

## Ćwiczenie 4

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

Kod funkcji PrzeciwnaU2 – funkcja znajduje liczbę przeciwną do danej liczby (obie w kodzie U2)

```

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

```

6.5.cpp

### ▼ Odejmowanie w różnych systemach

Algorytm zapiszemy w postaci listy kroków

**Krok1:** Ustal wartość początkową wyniku **c** na napis pusty

**Krok2:** Ustal wartość początkową zmiennej **pozyczka** na fałsz

**Krok3:** Uzupełnij odjemnik zerami nieznaczącymi tak, aby liczba cyfr w odjemniku była taka sama jak liczba cyfr odjemnej

**Krok4:** Dla każdej z cyfr odjemnej i odjemnika, traktowanych jako liczby jednocyfrowe, w kolejności od prawej do lewej wykonaj kroki 5 – 12

**Krok5:** Jeśli zmienna **pozyczka** ma wartość prawda, wykonaj **Krok:** 6

**Krok6:** Odejmij od cyfry odjemnej 1

**Krok7:** Ustal wartość zmiennej **pozyczka** na fałsz

**Krok8:** Oblicz różnicę cyfr odjemnej i odjemnika

**Krok9:** Jeśli różnica jest ujemna, to wykonaj kroki 10 – 11

**Krok10:** Ustal wartość zmiennej **pozyczka** na prawda

**Krok11:** Do różnicy dodaj podstawę systemu

**Krok12:** Dołącz cyfrę różnicy do wyniku

**Krok13:** Usuń z wyniku zera nieznaczące.

## ▼ Mnożenie w binarce

Specyfikacja problemu:

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

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

Zapis algorytmu mnożącego liczby binarne zgodnie z podaną specyfikacją może wyglądać tak jak poniżej. Wykorzystujemy w nim funkcję Dodaj, dodającą dwie liczby binarne.

$c \leftarrow "0"$

**dla** i ← długość b - 1 , ..., 0 **wykonuj**

**jeśli** b[i] = '1' **to** c ← Dodaj(c, a)

    a ← a + '0'

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

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

6.6.cpp

## ▼ Mnożenie w różnych systemach

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

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

W powyższym algorytmie założyliśmy, że umiemy wykonać dwie operacje: mnożenie liczby przez liczbę jednocyfrową oraz dodawanie dwóch liczb. Rozwiązania obu tych problemów zapiszemy w postaci funkcji. Najpierw zdefiniujemy funkcję `MnozPrzezCfr`, mnożącą liczbę przez liczbę jednocyfrową. Parametrami funkcji

będą:  $cfr$  – liczba jednocyfrowa,  $liczba$  – liczba, którą mnożymy przez  $cfr$ ,  $podst$  – podstawa systemu. W pseudokodzie zapis algorytmu, który realizuje ta funkcja, może wyglądać następująco:

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

Algorytm mnożenia liczb przez liczbę jednocyfrową jest analogiczny do algorytmu dodającego dwie liczby.

Dodawanie zastąpiliśmy iloczynem. W przypadku dodawania ewentualnie przeniesienie może być równe tylko 1, przy mnożeniu może być większe, ale jest reprezentowane przez liczbę jednocyfrową w danym systemie.

Oto kod funkcji mnożącej liczbę reprezentowaną jako napis przez liczbę jednocyfrową w danym systemie pozycyjnym o podstawie z zakresu od 2 do 10.

```
1 string MnozPrzezCfr(int cfr, string liczba, int podst)
2 {
3     int i, przn=0, iloczyn;
4     string wynik="";
5     for (i=liczba.size() - 1; i >= 0; i--)
6     {
7         iloczyn = cfr * (liczba[i] - '0') + przn;
8         przn = iloczyn / podst;
9         wynik = char(iloczyn % podst + '0') + wynik;
10    }
11    if (przn > 0) wynik = char(przn + '0') + wynik;
12    return wynik;
13 }
14
```

6.7.cpp

Kod źródłowy funkcji ma algorytm realizujący mnożenie dwóch liczb, wykorzystujący funkcje

## ▼ Dzielenie w binarce

Zapis algorytmu w pseudokodzie:

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

W algorytmie odwołujemy się do problemu odejmowania dwóch liczb binarnych wcześniej omówionego.

## ▼ Algorytm Euklidesa i działania na ułamkach

### ▼ NWD Naiwne (Jak stara Bartosza)

```
Krok1 Jeśli a > b, to zamień wartości a i b miejscami.
Krok2 Dla wartości d równych kolejno a, a - 1, a - 2, ..., 1 powtarzaj
    kroki 3 , 4 i 5
Krok3      Wyznacz resztę z dzielenia a przez d
Krok4      Wyznacz resztę z dzielenia b przez d
Krok5      Jeśli obie reszty wyznaczone w krokach 3 i 4 są równe 0,
            to zwróć wartość d i zakończ algorytm.
```

### ▼ NWD Euklidesem



**Algorytm Euklidesa** można zapisać w postaci następującej listy kroków:

- Krok 1: **Dopóki**  $a \neq b$ , **powtarzaj** kroki 2 i 3
- Krok 2: **Jeśli**  $a > b$ , **to**  $a = a - b$
- Krok 3: **W przeciwnym razie**  $b = b - a$ .
- Krok 4: **Zwróć** wartość  $a$

Poniżej znajduje się kod funkcji NWD, który jest realizacją algorytmu Euklidesa.

```
1 def NWD(a, b):
2     while a != b:
3         if a > b:
4             a = a - b
5         else:
6             b = b - a
7     return a
8
```

#### ▼ NWD z dzieleniem

**Oto lista kroków algorytmu Euklidesa w wersji z dzieleniem.**

- Krok 1: **Dopóki**  $a \neq 0$  i  $b \neq 0$ , **powtarzaj** kroki 2 i 3
- Krok 2: **Jeśli**  $a > b$ , **to**  $a = a \% b$
- Krok 3: **W przeciwnym razie**  $b = b \% a$
- Krok 4: **Jeśli**  $a \neq 0$ , **to** zwróć  $a$
- Krok 5: **W przeciwnym razie** zwróć  $b$ .

Poniżej znajduje się kod funkcji NWD, który jest realizacją algorytmu Euklidesa w wersji z dzieleniem.

```
1 def NWD(a, b):
2     while a != 0 and b != 0:
3         if a > b:
4             a = a % b
5         else:
6             b = b % a
7     if a != 0:
8         return a
9     else:
10        return b
11
```

#### ▼ Algorytmy na tekstach

##### ▼ Wypisywanie znaków

## Program *Znaki*. Funkcja `chr`

Napišemy teraz program, który wyświetli fragment tablicy Unicode pokazany w tabeli 1.2. Kody widocznych tam znaków mieszczą się między 32 a 126. Program powinien wyświetlić znaki z odstępami w wierszach po 16 znaków (nie licząc spacji).

Oto kod źródłowy programu *Znaki*:

```
1 for i in range(32, 127):
2     znak = chr(i)
3     print(znak, end=" ")
4     if i % 16 == 15:
5         print()
6     print()
```

Wynik działania programu:



```
C:\Users\Dom\AppData\Local\Programs\Pyth
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ 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 [ \ ] ^ _
` 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 { | } ~

Process finished with exit code 0
```

▼ Poprawność e-mail

## Program *Znaki*. Funkcja `chr`

Napišemy teraz program, który wyświetli fragment tablicy Unicode pokazany w tabeli 1.2. Kody widocznych tam znaków mieszczą się między 32 a 126. Program powinien wyświetlić znaki z odstępami w wierszach po 16 znaków (nie licząc spacji).

Oto kod źródłowy programu *Znaki*:

```
1 for i in range(32, 127):
2     znak = chr(i)
3     print(znak, end=" ")
4     if i % 16 == 15:
5         print()
6     print()
```

Wynik działania programu:



```
C:\Users\Dom\AppData\Local\Programs\Pyth
! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ 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 [ \ ] ^ _
` 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 { | } ~

Process finished with exit code 0
```

### ▼ Popierdolone sprawdzanie e-mail

Algorytm sprawdzający poprawność adresu e-mail wygodnie będzie zapisać w kodzie źródłowym w formie funkcji:

```
1 def CzyPoprawnyAdres(adres):
2     dl = len(adres)
3
4     i = 0 #etap 1
5     while adres[i] != "@" and i < dl - 1:
6         i = i + 1
7     if adres[i] != "@" or i < 2:
8         return False
9
10    j = dl - 1 #etap 2
11    while adres[j] != "@":
12        j = j - 1
13    if i != j:
14        return False
15
16    k = dl - 1 #etap 3
17    while adres[k] != "." and k > 0:
18        k = k - 1
19    if adres[k] != "." or not (k == dl - 3 or k == dl - 4):
20        return False
21
22    if k - i <= 1: #etap 4
23        return False
24
25    return True
```

▼ Jeszcze bardziej zjedbane sprawdzanie e-mail

Oto zapis funkcji *CzyPoprawnyAdres* z użyciem powyższych metod:

```
1 def CzyPoprawnyAdres(adres):
2     dl = len(adres)
3
4     i = adres.find("@") #etap 1
5     if i < 2 or i == -1:
6         return False
7
8     j = adres.rfind("@") #etap 2
9     if i != j:
10        return False
11
12    k = adres.find(".") #etap 3
13    if k == -1:
14        return False
15    if not (k == dl - 3 or k == dl - 4):
16        return False
17
18    if k - i <= 1: #etap 4
19        return False
20
21    return True
```

## ▼ Usuwanie duplikatów z tekstu

Oto fragment kodu źródłowego programu *Usuwanie powtórzeń*:

```
1 N = 20
2
3 wynik = []
4 i = 1
5 nowy = input()
6 wynik.append(nowy)
7
8 stary = nowy
9 nowy = input()
10
11 while nowy != "***" and i < N:
12     if nowy != stary:
13         i = i + 1
14         wynik.append(nowy)
15         stary = nowy
16         nowy = input()
17
18 for j in range(i):
19     print(wynik[j])
```

## ▼ Szukanie wzorca w tekście

Oto kod źródłowy programu *Szukaj wzorca*:

```
1  TEKST = "ALA ALBO ADA"
2
3  def Porownaj(wzorzec):
4      n = len TEKST
5      m = len(wzorzec)
6      for poz in range(0, n - m + 1):
7          j = 0
8          while j < m and TEKST[poz + j] == wzorzec[j]:
9              j = j + 1
10         if j == m:
11             return poz
12     return -1
13
14     print("Tekst : ", TEKST)
15     print("Podaj wzorzec : ", end=" ")
16     wzorzec = input()
17
18     print("Pozycja : ", Porownaj(wzorzec))
```

## ▼ Szyfry

### ▼ Szyfrowanie kolumnowe

Lista kroków algorytmu szyfrowania kolumnowego może wyglądać następująco:

**Krok 1:** Wczytaj tekst jawny.

**Krok 2:** Wczytaj klucz szyfrowania i oznacz tę liczbę jako klucz.

**Krok 3:** Oznacz długość tekstu jawnego jako dl.

**Krok 4:** Dla każdej liczby *i* z zakresu od 0 do klucz - 1, traktowanej jako kolejna wartość licznika pętli, wykonuj krok 5

**Krok 5:** Dołącz do tekstu szyfrogramu następujące litery tekstu jawnego: tę na pozycji *i* oraz wszystkie te, których pozycja jest liczbą *i* powiększoną o wielokrotność klucza.

Zapiszmy kod źródłowy programu *Szyfr kolumnowy*:

```
1  szfrogram = ""
2
3  jawny = input("Podaj tekst jawany (bez spacji) : ")
4  klucz = int(input("Podaj klucz szyfrowania : "))
5
6  dl = len(jawny)
7
8  for i in range(0, klucz):
9      for j in range(i, dl, klucz):
10         szfrogram = szfrogram + jawny[j]
11
12  print("\nSzyfrogram : ", szfrogram)
13
```

## ▼ Szyfrowanie Cezara

## Specyfikacja

**Dane:** tekst jawny składający się wyłącznie z wielkich liter alfabetu łacińskiego; klucz szyfrowania w postaci liczby całkowitej z zakresu od 1 do 25.

**Wynik:** tekst zaszyfrowany szyfrem Cezara z podanym kluczem.

Lista kroków algorytmu szyfrującego metodą Cezara pojedyncze słowo może wyglądać następująco:

**Krok 1:** Wczytaj tekst jawny.

**Krok 2:** Wczytaj klucz szyfrowania i oznacz tę liczbę jako klucz.

**Krok 3:** Dla każdej litery tekstu jawnego **wykonuj** kolejno kroki 4 i 5

**Krok 4:** Oznacz kod Unicode tej litery jako kod.

**Krok 5:** Jeżeli kod + klucz jest liczbą nie większą od kodu Unicode litery Z, to dołącz do szyfrogramu literę o kodzie Unicode równym kod + klucz. W przeciwnym razie dołącz do szyfrogramu literę o kodzie Unicode wynoszącym kod + klucz - 26.

Kody Unicode kolejnych liter szyfrogramu zapamiętamy w zmiennej pomocniczej kod. Aby wyznaczyć kod Unicode litery szyfrogramu, do kodu litery tekstu jawnego, otrzymanego jako wartość funkcji `ord`, dodajemy przesunięcie, czyli wartość klucza. Jeśli otrzymana wartość będzie większa od kodu litery Z (liczby 90), musimy odjąć 26 (liczbę liter alfabetu), aby otrzymać kod litery z początku alfabetu.

Na przykład podczas szyfrowania litery Y w słowie INFORMATYKA z kluczem 3 otrzymamy literę B. Kod Unicode litery Y to 89. Po dodaniu 3 uzyskujemy 92, a więc wartość większą od kodu litery Z. Odejmujemy 26 i otrzymujemy 66, czyli kod litery B.

Oto kod źródłowy programu *Szyfr Cezara*:

```
1  szfrogram = ""
2
3  jawny = input("Podaj słowo : ")
4  klucz = int(input("Podaj klucz szyfrowania : "))
5
6  dl = len(jawny)
7
8  for i in range(0, dl):
9      kod = ord(jawny[i]) + klucz
10     if kod > ord("Z"):
11         kod = kod - 26
12     szfrogram = szfrogram + chr(kod)
13
14  print("\nSzyfrogram : ", szfrogram)
```

### ▼ Cezar z pliku



```

1  int main()
2  {
3      ifstream wejscie("tekst_jawny.txt");
4      ofstream wyjscie("szfrogram.txt");
5      string s;
6      int klucz;
7      cout<<"Klucz : "; cin>>klucz;
8      while (!wejscie.eof())
9      {
10         getline(wejscie, s);
11         wyjscie<<Cezar(s, klucz)<<endl;
12     }
13     wejscie.close();
14     wyjscie.close();
15     cout<<"Plik szyfrogram.txt zostal utworzony";
16     return 0;
17 }

```

## ▼ Cezar PL

Oto definicja stałych w naszym programie:

```

const string alfabet_m="aąbcćdeęfghijklmńnoópqrsstuvwxyzżź";
const string alfabet_w="AĄBCĆDEĘFGHIJKŁŁMNŃNOÓPQRSSTUVWXYZŻŻ";

```

Litery alfabetu będziemy numerować od 0 do 34. Sprawdzanie, czy znak tekstu jawnego jest literą, będzie polegało na poszukiwaniu tego znaku w jednym z napisów będących wartościami stałych. Wykorzystamy do tego metodę find. Jeśli znak zostanie odnaleziony, program wyznaczy odpowiadający mu znak szyfrogramu.

Oto kod źródłowy funkcji szyfrującej pojedynczy znak (literę alfabetu):

```

1  char Cezar_PL(char znak, int klucz)
2  {
3      int i=alfabet_m.find(znak);
4      if (i >= 0 && i < 35)
5      {
6          i=(i + klucz) % 35;
7          return alfabet_m[i];
8      }
9      i = alfabet_w.find(znak);
10     if (i >= 0 && i < 35)
11     {
12         i=(i + klucz) % 35;
13         return alfabet_w[i];
14     }
15     return znak;
16 }

```