

Wstęp do informatyki

Wykład 4

Uniwersytet Wrocławski

Instytut Informatyki

Plan na dziś

1. Pseudokod/schemat blokowy – język programowania – kod maszynowy
2. O translacji programów na kod maszynowy
3. Programujemy wydajnie i analizujemy programy!

Schematy blokowe a języki programowania

Przykład 1

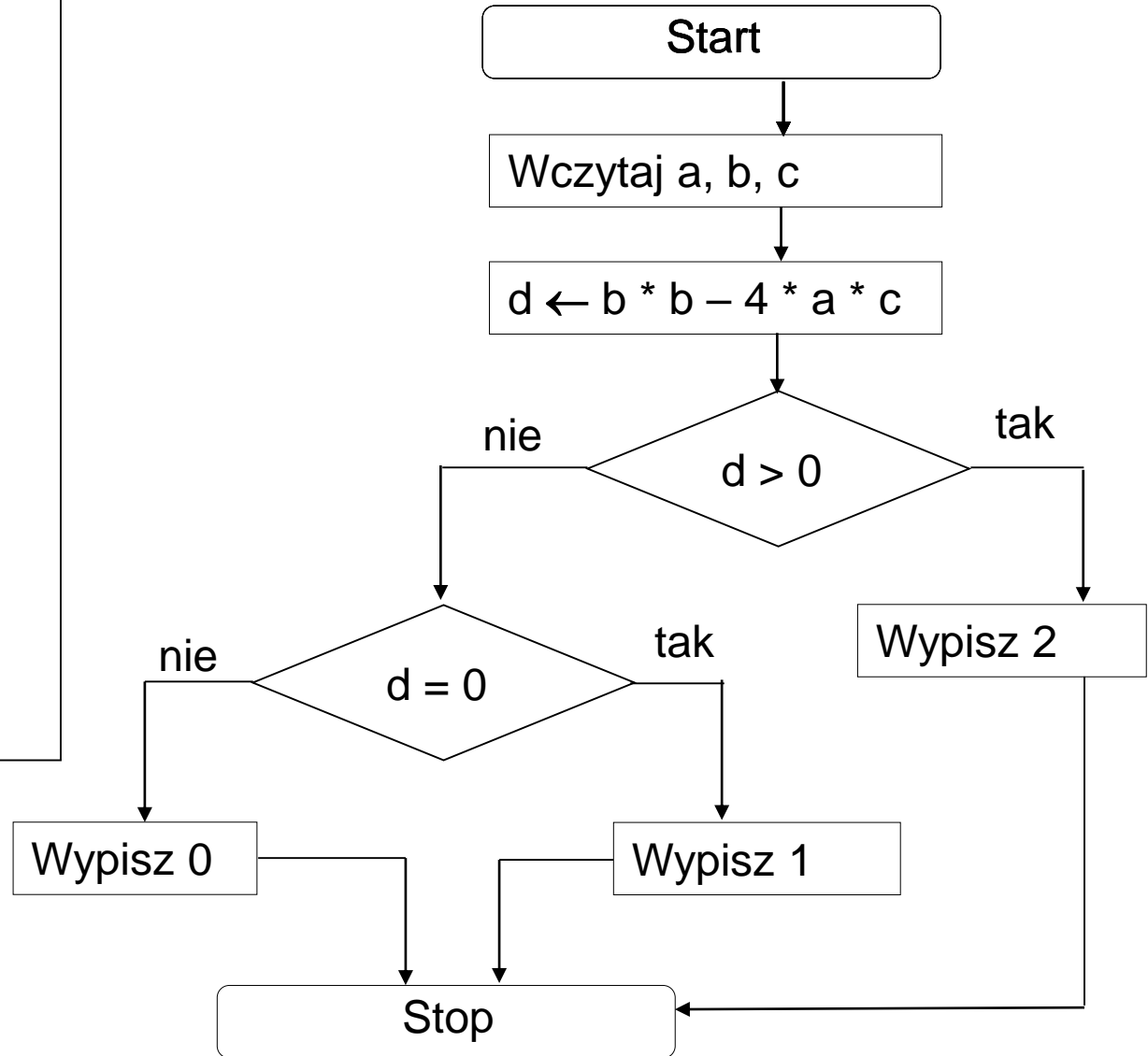
Specyfikacja

Wejście:

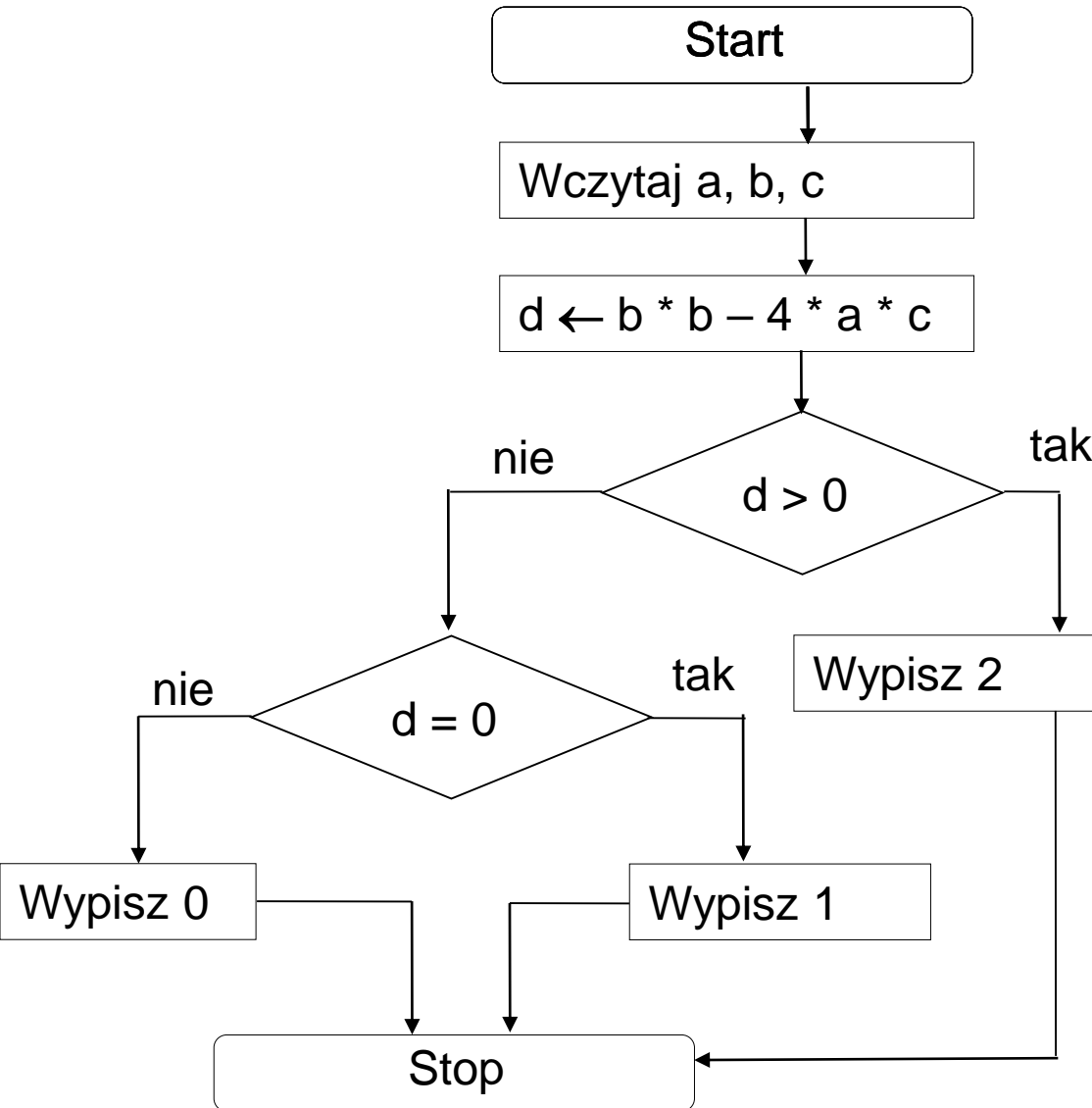
a, b, c – liczby
rzeczywiste, $a \neq 0$

Wyjście:

liczba rozwiązań
równania
 $ax^2+bx+c=0$



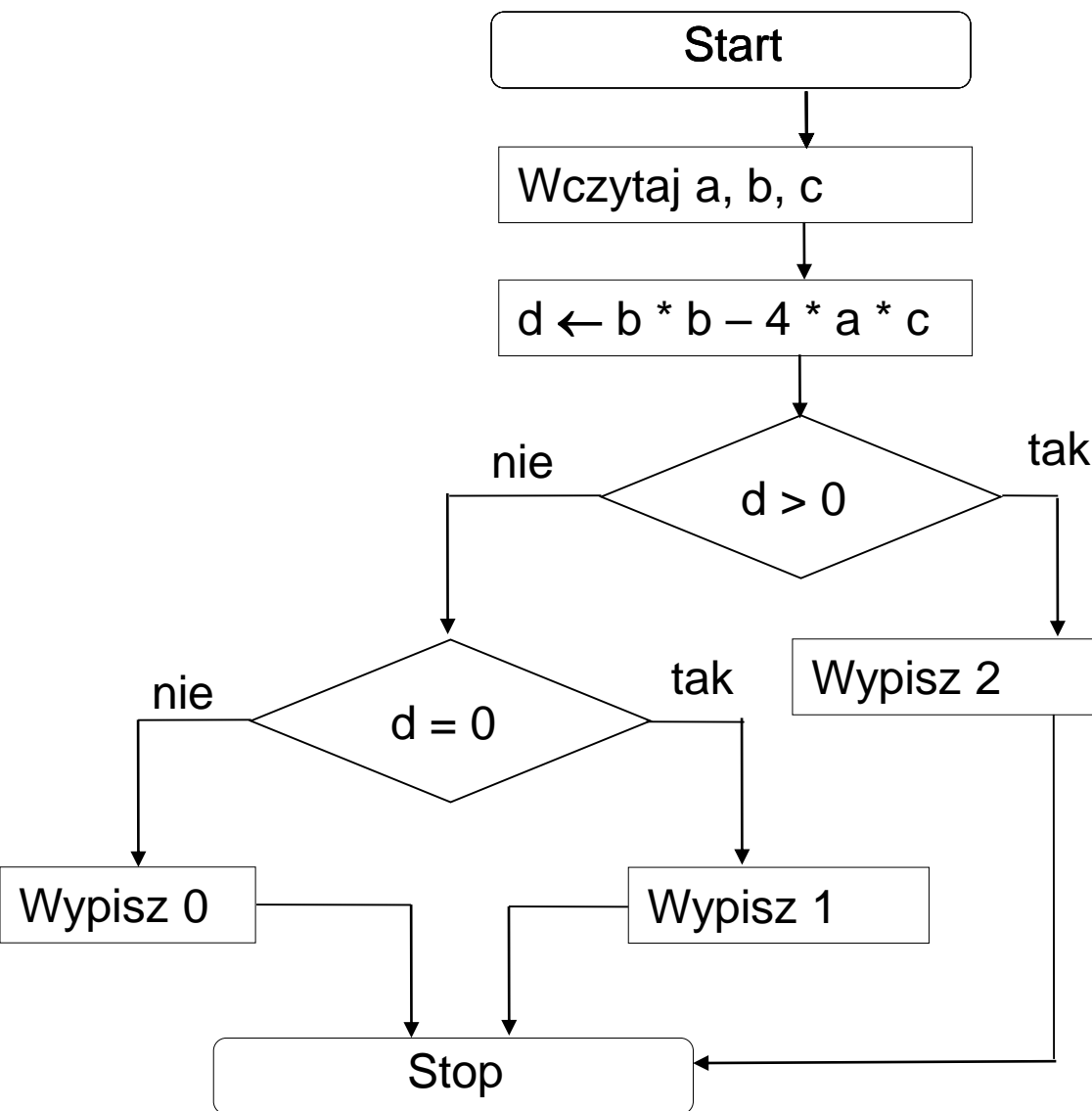
Przykład 1: Ansi C



main()

```
{  
  int  a, b, c, d;  
  
  scanf("%d %d %d",&a,&b, &c);  
  d = b * b - 4 * a * c;  
  if (d>0) printf("%d\n", 2);  
  else  
    if (d==0) printf("%d\n", 1);  
    else printf("%d\n", 0);  
}
```

Przykład 1: Python



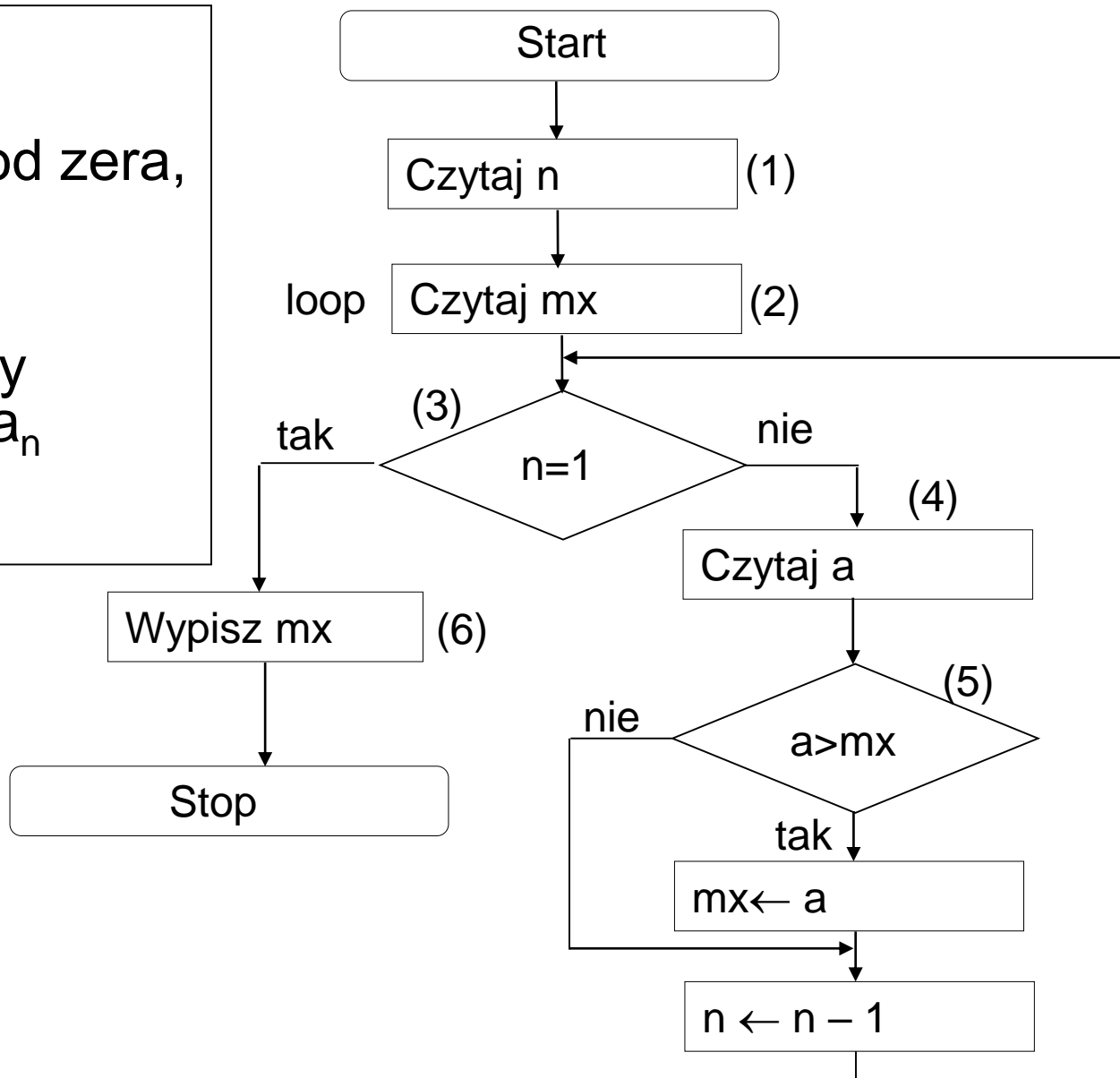
```
a=input("Podaj liczbe a\n")
b=input("Podaj liczbe b\n")
c=input("Podaj liczbe c\n")
d = b * b - 4 * a * c;
if d>0:
    print "2"
else:
    if d==0:
        print "1"
    else:
        print "0"
```

Przykład 2

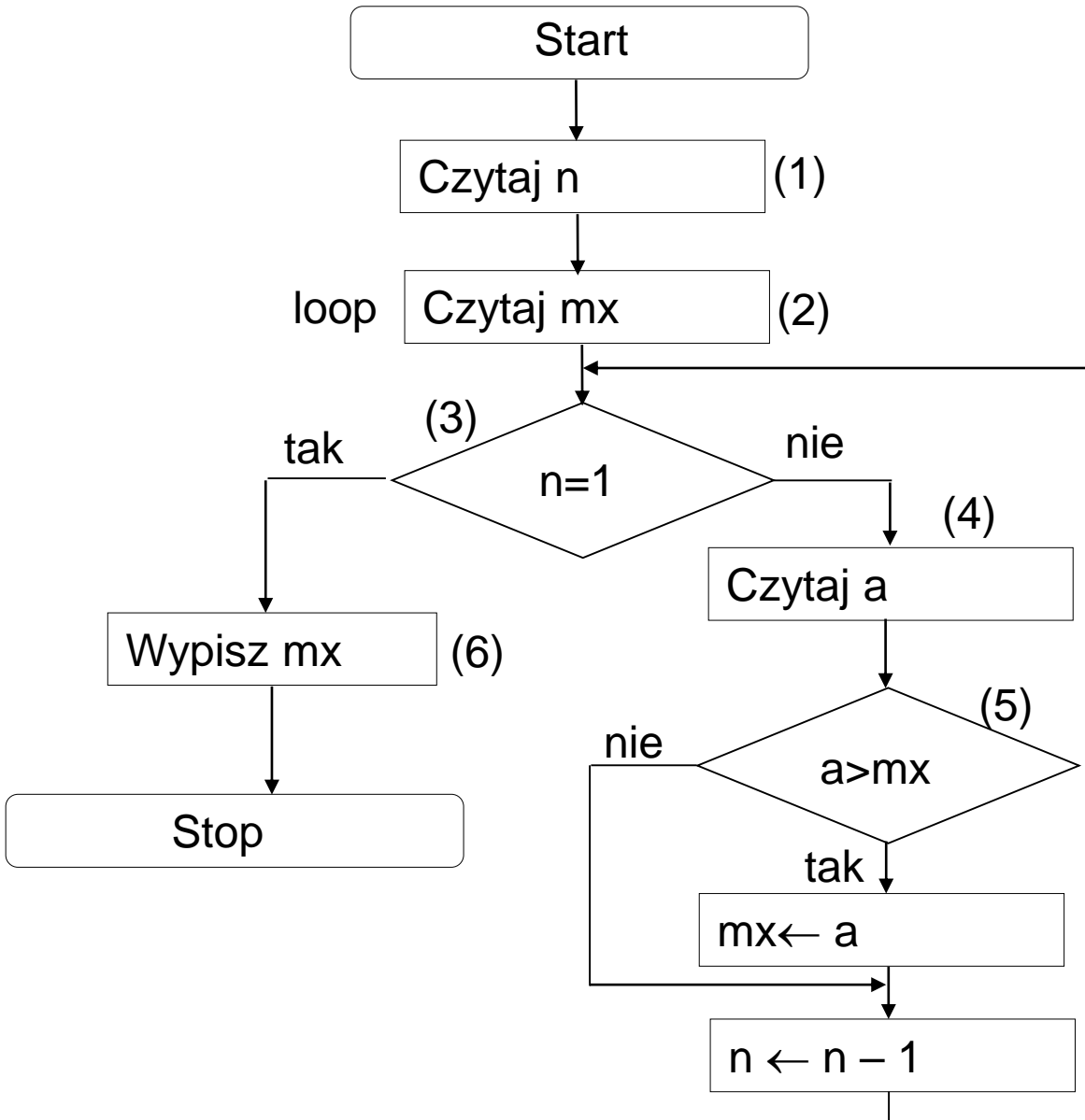
Specyfikacja

Wejście: n – liczba naturalna większa od zera, a_1, \dots, a_n – ciąg liczb

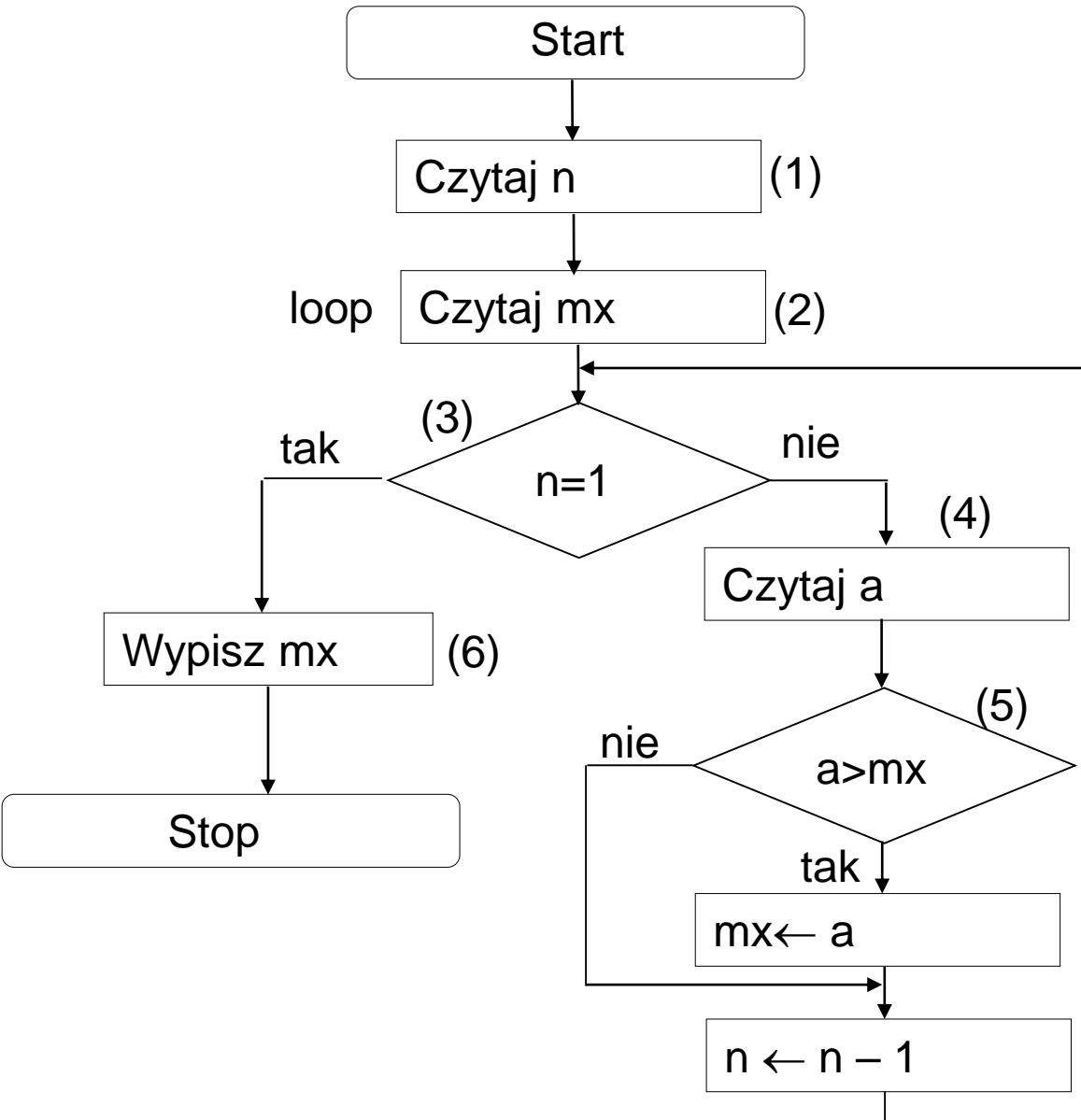
Wyjście: największy element ciągu $a_1 \dots a_n$



Przykład 2: Ansi C

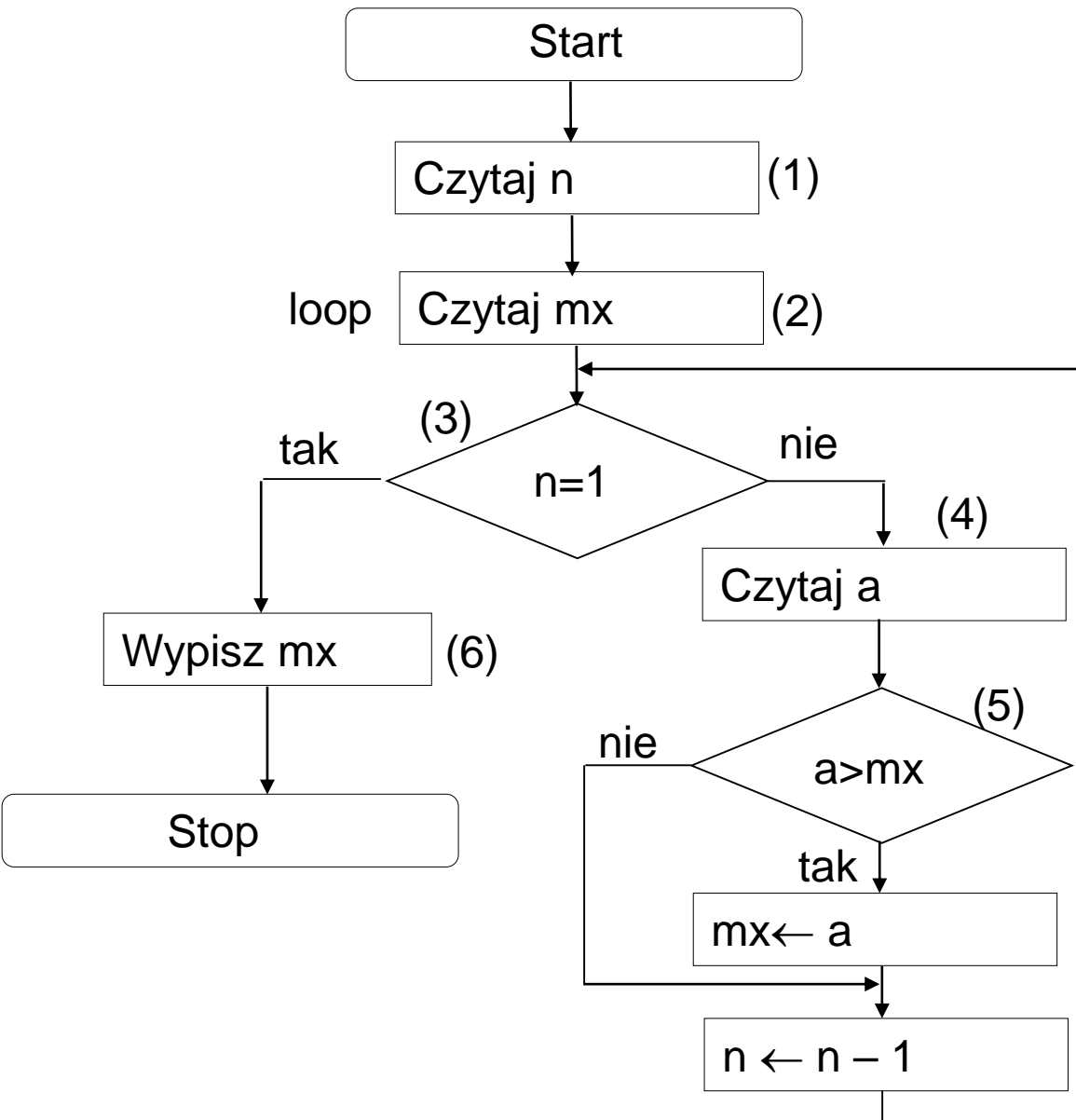


Przykład 2: Ansi C



```
main()  
{  
  int n, mx, a;  
  
  scanf("%d %d",&n,&mx);  
  while (n != 1) {  
    scanf("%d",&a);  
    if (a>mx) mx = a;  
    n - -;  
  }  
  printf("Max=%d\n",mx);  
}
```

Przykład 2: Python



```
n=input("Podaj dlugosc\n ciagu:\n")
mx=input("Podaj liczbe:\n")
while n != 1:
    a=input("Podaj : \n")
    if a>mx:
        mx = a
    n=n-1

print "Max=", mx
```

Nieformalne wprowadzenie do translacji programów

Program w C a kod maszynowy

Kompilacja: „tłumaczenie” programu (napisanego w języku wysokiego poziomu) do postaci wykonywalnej przez komputer.

Kompilator: program służący do kompilacji.

Interpreter: program tłumaczący i „wykonujący” pojedyncze instrukcje programu napisane w języku **wysokiego poziomu** (np. Python, C), w trakcie jego uruchamiania.

Postać wykonywalna przez komputer: program w kodzie **maszynowym** (z podobnym zestawem instrukcji do kodu **RAM**), np. wynik działania kompilatora.

Tłumaczenie programu na kod RAM

Kompilacja – przykład poglądowy:

1. Uproszczenie kodu źródłowego programu.
2. Skojarzenie zmiennych z komórkami (obszarami pamięci)
3. Automatyczne metody tłumaczenia instrukcji podstawienia / warunkowych... pętli i in.

Kompilację programu rozkładamy na „tłumaczenie” jego poszczególnych instrukcji.

Tłumaczenie programu na kod RAM

Przykład poglądowy, etap 1 – uproszczenie:

```
main()
{
    int m, k, rez, i;

    rez = 1;
    scanf("%d %d", &m, &k);
    for(i=0; i<k; i++) rez *= m;
    printf("%d\n",rez);
}
```

```
main()
{
    int m, k, rez, i;

    rez = 1;
    scanf("%d", &m);
    scanf("%d", &k );
    i = 0;
    while (k - i > 0) {
        rez = rez * m;
        i = i + 1; }
    printf("%d",rez);
}
```

Tłumaczenie programu na kod RAM

Przykład poglądowy, etap 2 – skojarzenie zmiennych z kom.:

```
main()
{
    int m, k, rez, i;

    rez = 1;
    scanf("%d", &m);
    scanf("%d", &k );
    i = 0;
    while (k - i > 0) {
        rez = rez * m;
        i = i + 1; }
    printf("%d",rez);
}
```

```
rez = 1
m=input("Podstawa:\n")
k=input("Wykladnik:\n")
i = 0
while k - i > 0:
    rez = rez * m
    i = i + 1
print "Wynik:", rez
```

Komórka	Zmienna
1	
2	m
3	k
4	rez
5	i

Tłumaczenie programu na kod RAM

Przykład poglądowy, etap 3 – tłumaczenie:

```
main()
{
    int m, k, rez, i;

    rez = 1;
    scanf("%d", &m);
    scanf("%d", &k);
    i = 0;
    while (i - k < 0) {
        rez = rez * m;
        i = i + 1;
    }
    printf("%d", rez);
}
```

```
rez = 1
m=input("Podstawa:\n")
k=input("Wykładnik:\n")
i = 0
while k - i > 0:
    rez = rez * m
    i = i + 1
print "Wynik:", rez
```

```
load =1
store 4
read 2
read 3
load =0
store 5
petla: load 5
      sub 3
      jzero wypisz
      jgtz wypisz
      load 4
      mult 2
      store 4
      load 5
      add =1
      store 5
      jump petla
wypisz: write 4
```

Komórka	Zmienna
1	
2	m
3	k
4	rez
5	i

Kompilacja w praktyce...

Problemy:

- Pamięć zajmowana przez kod („treść”) programu a pamięć zajmowana przez zmienne?
- Tłumaczenie i uruchamianie funkcji?
- Zmienne globalne (wspólne dla całego programu) a zmienne lokalne?

Pamięć programu

Pamięć programu podzielona jest na kilka obszarów, m.in.:

1. obszar zajmowany przez **zmienne globalne** (inaczej zewnętrzne);
2. obszar zajmowany przez instrukcje programu (jego „treść”);
3. obszar przeznaczony na **stos wywołań funkcji**;
4. obszar przeznaczony na obiekty dynamiczne (tzw. **sterta**).

Zmienna globalna:

- (zadeklarowana) poza funkcjami
- wspólna dla całego programu (wszystkich funkcji)

Program:

- zbiór funkcji i struktur danych
- „program główny” (funkcja main w Ansi C)

Stos wywołań funkcji

Stos wywołań przechowuje :

- informacje o tym jakie funkcje są aktualnie uruchomione i w jakiej kolejności (hierarchia)
- informacje o tym co należy zrobić gdy uruchomione funkcje zakończą działanie
- zmienne lokalne wywołanych funkcji
- parametry uruchomionych funkcji

Stos wywołań funkcji

Kompilator przekłada instrukcję wywołania funkcji na ciąg rozkazów, który m.in.:

1. Rezerwuje na *szczycie stosu wywołań* odpowiedni dla danej funkcji fragment pamięci, w którym zostanie przydzielone miejsce m.in. na:
 - Parametry funkcji
 - Zmienne lokalne
 - Poprzednią wartość *wskaźnika szczytu stosu wywołań*
 - *Adres powrotu* (...po wykonaniu funkcji)
2. Oblicza wartości wyrażeń z wywołania funkcji, przypisuje je odpowiednim parametrom.
3. Uaktualnia wskaźnik szczytu stosu wywołań.
4. Zmienia *licznik rozkazów* tak, by wskazywał na pierwszą instrukcję funkcji (adres w pamięci, w którym znajduje się pierwsza instrukcja funkcji).

Stos wywołań – przykład (f1)

```
int f1(int x, int y)
```

```
{ int z;
```

```
  f2( x + y );    ⇐2
```

```
  return 1;
```

```
}
```

```
int f2(int a)
```

```
{ int b;
```

```
  f3( 2 * a );
```

```
  return 1;
```

```
}
```

```
int f3(int m)
```

```
{ int n;
```

```
  return 1;
```

```
}
```

```
main(){
```

```
  f1( 5, 7 )    ⇐ ⇐1
```

```
  printf("koniec");
```

```
}
```

f1	x = 5, y = 7, z=?, ⇐ ₁ , → ₁

⇐ ⇐ – pozycja licznika rozkazów

→ → - pozycja wskaźnika szczytu stosu

Stos wywołań – przykład (f2)

```
int f1(int x, int y)
{ int z;
  f2( x + y );    ⇐ ⇐2
  return 1;
}
int f2(int a)
{ int b;
  f3( 2 * a );    ⇐3
  return 1;
}
int f3(int m)
{ int n;
  return 1;
}

main(){
  f1( 5, 7 )      ⇐1
  printf("koniec");
}
```

→₂
→₁

f2	a = 12, b=?, ⇐ ₂ , → ₂
f1	x = 5, y = 7, z=?, ⇐ ₁ , → ₁

⇐ ⇐ – pozycja licznika rozkazów

→ → - pozycja wskaźnika szczytu stosu

Stos wywołań – przykład (f3)

```
int f1(int x, int y)
```

```
{ int z;
```

```
  f2( x + y );    ⇐2
```

```
  return 1;
```

```
}
```

```
int f2(int a)
```

```
{ int b;
```

```
  f3( 2 * a );    ⇐ ⇐3
```

```
  return 1;
```

```
}
```

```
int f3(int m)
```

```
{ int n;
```

```
  return 1;      ⇐
```

```
}
```

```
main(){
```

```
  f1( 5, 7 )      ⇐1
```

```
  printf("koniec");
```

```
}
```

f3	m = 24, n=?, ⇐ ₃ , → ₃
f2	a = 12, b=?, ⇐ ₂ , → ₂
f1	x = 5, y = 7, z=?, ⇐ ₁ , → ₁

⇐ ⇐ – pozycja licznika rozkazów

→ → - pozycja wskaźnika szczytu stosu

Zakończenie wykonania funkcji

Zakończenie wykonywania funkcji (poprzez instrukcje **return** czy też poprzez wykonanie ostatniej instrukcji) powoduje m.in.:

1. Przywrócenie **wskaźnikowi szczytu stosu** wartości jaką miał przed wywołaniem tej funkcji.
2. Przypisanie **licznikowi rozkazów** wartości adresu powrotu zapamiętanego w trakcie wywołania funkcji.

Stos wywołań – zakończ. f3

```
int f1(int x, int y)
```

```
{ int z;
```

```
  f2( x + y );    ←2
```

```
  return 1;
```

```
}
```

```
int f2(int a)
```

```
{ int b;
```

```
  f3( 2 * a );    ←3
```

```
  return 1;
```

```
}
```

```
int f3(int m)
```

```
{ int n;
```

```
  return 1;      ←←
```

```
}
```

```
main(){
```

```
  f1( 5, 7 )      ←1
```

```
  printf("koniec");
```

```
}
```

→→

→₃

→₂

→₁

f3	m = 24, n=?, ← ₃ , → ₃
f2	a = 12, b=?, ← ₂ , → ₂
f1	x = 5, y = 7, z=?, ← ₁ , → ₁

←← – pozycja licznika rozkazów

→→ - pozycja wskaźnika szczytu stosu

Stos wywołań – zakończ. f2

```
int f1(int x, int y)
{ int z;
  f2( x + y );      ⇐2
  return 1;
}
int f2(int a)
{ int b;
  f3( 2 * a );
  return 1;      ⇐ ⇐
}
int f3(int m)
{ int n;
  return 1;
}

main(){
  f1( 5, 7 )        ⇐1
  printf("koniec");
}
```

→→	f2 a = 12, b=?, ⇐ ₂ , → ₂
→ ₂	f1 x = 5, y = 7, z=?, ⇐ ₁ , → ₁

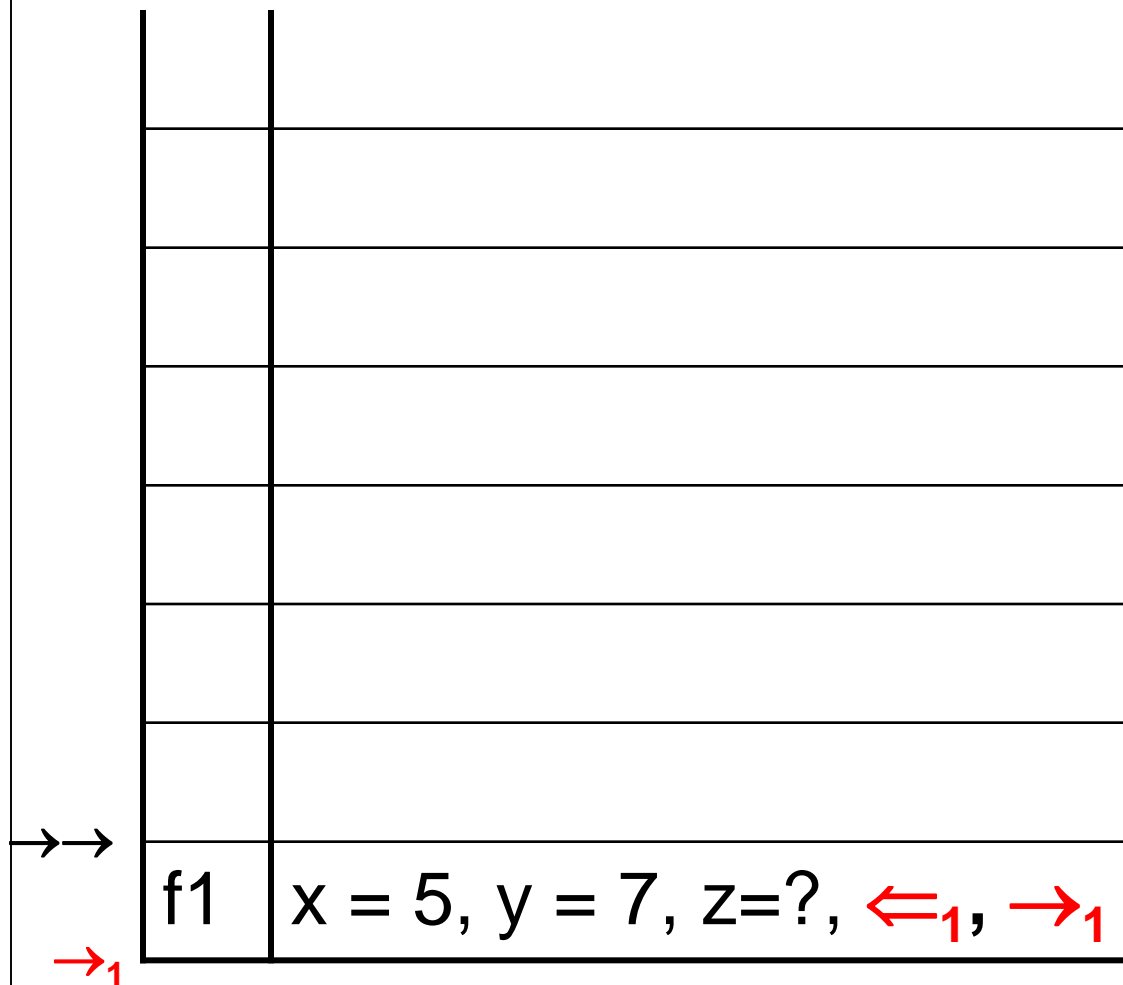
⇐ ⇐ – pozycja licznika rozkazów

→→ - pozycja wskaźnika szczytu stosu

Stos wywołań – zakończ. f1

```
int f1(int x, int y)
{ int z;
  f2( x + y );      ←2
  return 1;         ← ←
}
int f2(int a)
{ int b;
  f3( 2 * a );
  return 1;
}
int f3(int m)
{ int n;
  return 1;
}

main(){
  f1( 5, 7 )        ←1
  printf("koniec");
}
```



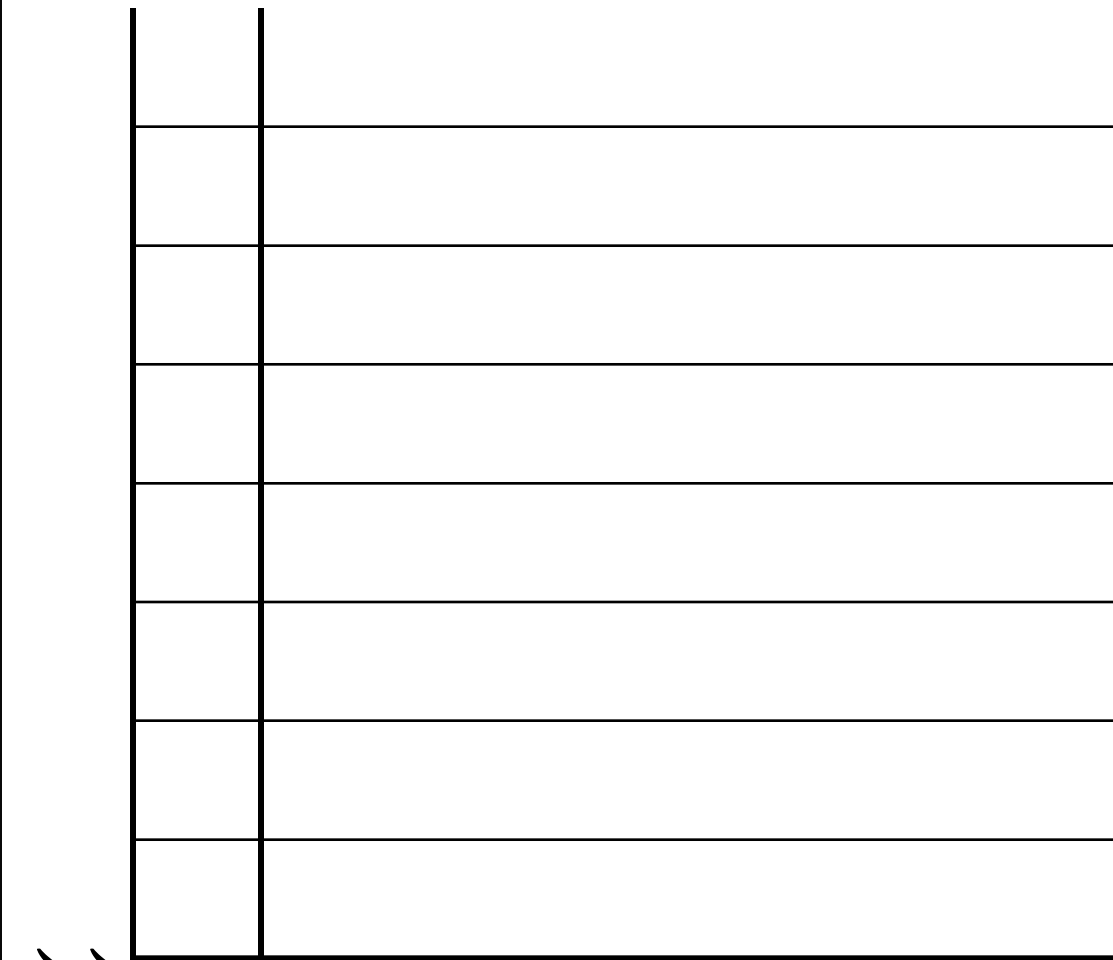
← ← – pozycja licznika rozkazów

→ → – pozycja wskaźnika szczytu stosu

Stos wywołań – zakończ. main

```
int f1(int x, int y)
{ int z;
  f2( x + y );
  return 1;
}
int f2(int a)
{ int b;
  f3( 2 * a );
  return 1;
}
int f3(int m)
{ int n;
  return 1;
}

main(){
  f1( 5, 7 )
  printf("koniec");
}
```



← ← – pozycja licznika rozkazów

→ → - pozycja wskaźnika szczytu stosu

Piszemy wydajne programy...

Największy wspólny dzielnik (nwd)

Specyfikacja:

Wejście: n, m – liczby naturalne większe od 0

Wyjście: największy wspólny dzielnik n i m
(czyli $\max \{ p : p \mid n \text{ oraz } p \mid m \}$)

Dodatkowa definicja (podzielność):

$a \mid b$ dla naturalnych dodatnich a, b gdy istnieje naturalna liczba c taka, że $b = c \cdot a$

nwd: rozwiązanie naiwne

Przypomnienie:

$a \mid b$: b jest podzielne przez a

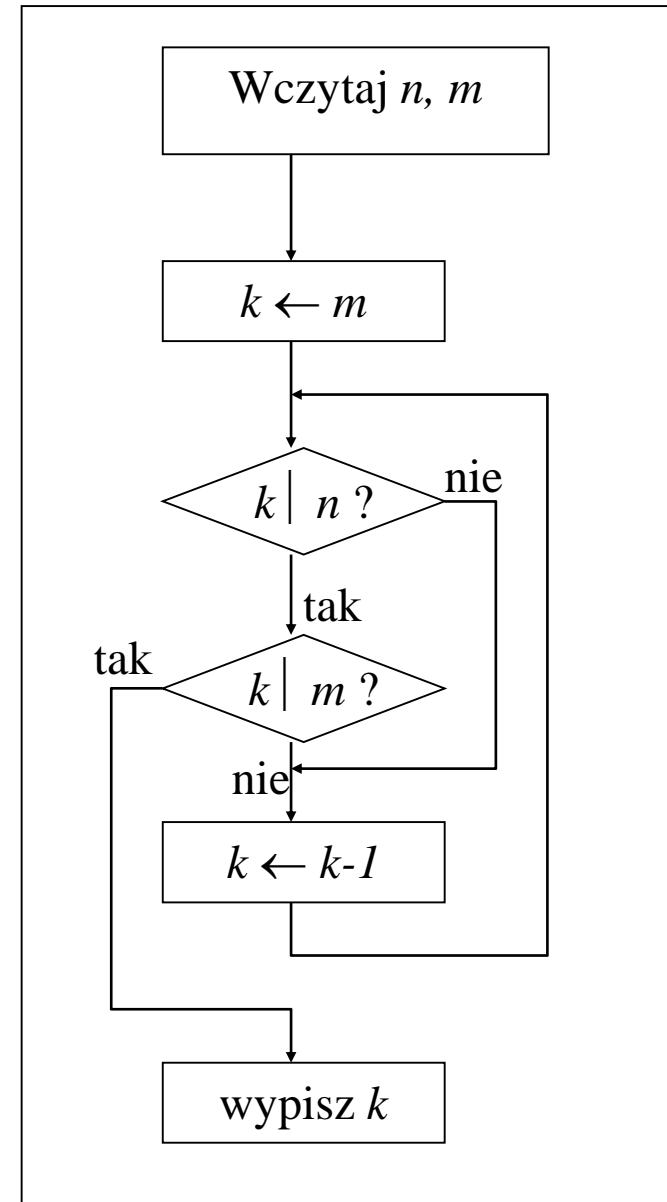
Algorytm:

wczytaj n, m

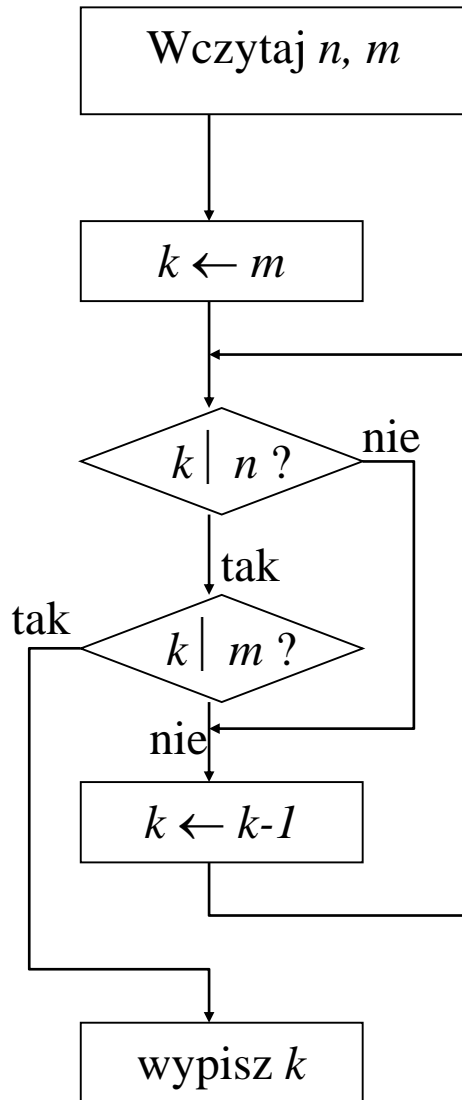
$k \leftarrow m$

powtarzaj:

- jeśli $k \mid n$ oraz $k \mid m$ to
wypisz k i STOP
- $k \leftarrow k - 1$



nwd: rozwiązanie naiwne



```
main()
{
    int n,m,k;
    scanf("%d %d", &n, &m);
    k = m;
    while (n%k!=0 || m % k != 0)
        k--;
    printf("Wynik: %d\n", k);
}
```

```
n=input("pierwsza:\n")
m=input("druga:\n")
k = m
while n%k!=0 or m%k != 0:
    k=k-1
print "Wynik: ", k
```


nwd w C: rozwiązanie naiwne (while/for)

```
main()
{
    int n,m,k;
    scanf("%d %d", &n, &m);
    k = m;
    while (n% k!=0 || m % k != 0)
        k--;
    printf("Wynik: %d\n", k);
}
```

```
main()
{
    int n,m,k;
    scanf("%d %d", &n, &m);
    for( k = m; n% k!=0 || m % k != 0; k- -) ;
    printf("Wynik: %d\n", k);
}
```

nwd: rozwiązanie naiwne

Złożoność:

- Pamięć: $O(1)$
- Czas: $O(m)$

Modyfikacja:

jeśli $n < m$, zamień n z m przed pętlą.

Wówczas:

- Czas: $O(\min(n, m))$

nwd: algorytm Euklidesa v.1

Obserwacja (kluczowa)

Niech d, m, n to liczby naturalne takie, że $m \leq n$. Wówczas

1. Jeśli $d \mid m$ oraz $d \mid n$, to $d \mid (n - m)$.
2. Jeśli $d \mid m$ oraz $d \mid (n - m)$, to $d \mid n$.

Dowód:

1. Jeśli $d \mid m$ oraz $d \mid n$ to:

- $m = ad$, $n = bd$ dla naturalnych $a \leq b$;
- zatem $n - m = (b - a)d$, czyli $d \mid (n - m)$.

2. Jeśli $d \mid m$ oraz $d \mid (n - m)$:

- $m = ad$, $(n - m) = bd$ dla naturalnych a oraz b ;
- zatem $n = m + (n - m) = (a + b)d$, czyli $d \mid n$.

nwd: algorytm Euklidesa v.1

Obserwacja (kluczowa)

Niech d, m, n to liczby naturalne takie, że $m < n$.

Wówczas

1. Jeśli $d \mid m$ oraz $d \mid n$, to $d \mid (n - m)$.

2. Jeśli $d \mid m$ oraz $d \mid (n - m)$, to $d \mid n$.

czyli zbiór wspólnych dzielników n i m oraz zbiór wspólnych dzielników $n - m$ i m są równe!

Wniosek

1. $\text{nwd}(m, m) = m$

2. $\text{nwd}(n, m) = \text{nwd}(n - m, m)$ dla $n > m > 0$.

3. $\text{nwd}(n, m) = \text{nwd}(m, n)$.

nwd: algorytm Euklidesa v.1

Wniosek

1. $\text{nwd}(m, m) = m$
2. $\text{nwd}(n, m) = \text{nwd}(n - m, m)$ dla $n > m > 0$.
3. $\text{nwd}(n, m) = \text{nwd}(m, n)$.

Algorytm:

1. Wczytaj n, m
2. Dopóki $n \neq m$
 - Jeśli $n > m$: $n \leftarrow n - m$
 - w przeciwnym przypadku: $m \leftarrow m - n$
3. Wypisz n

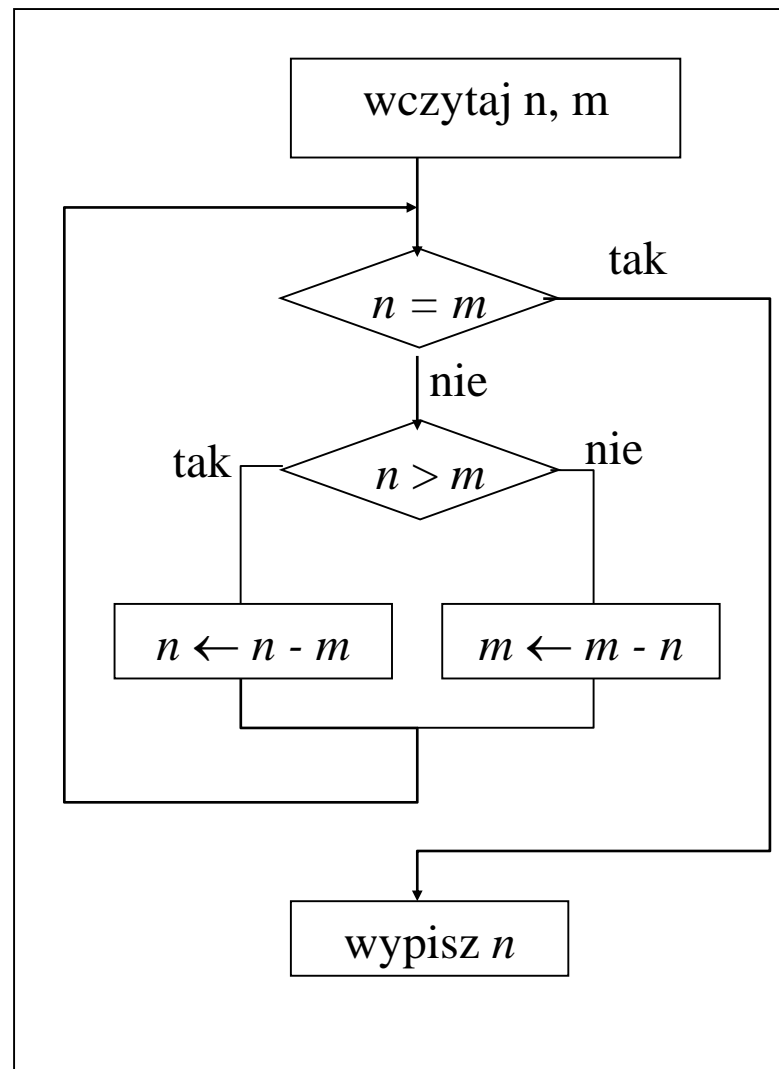
nwd: algorytm Euklidesa v.1

Algorytm:

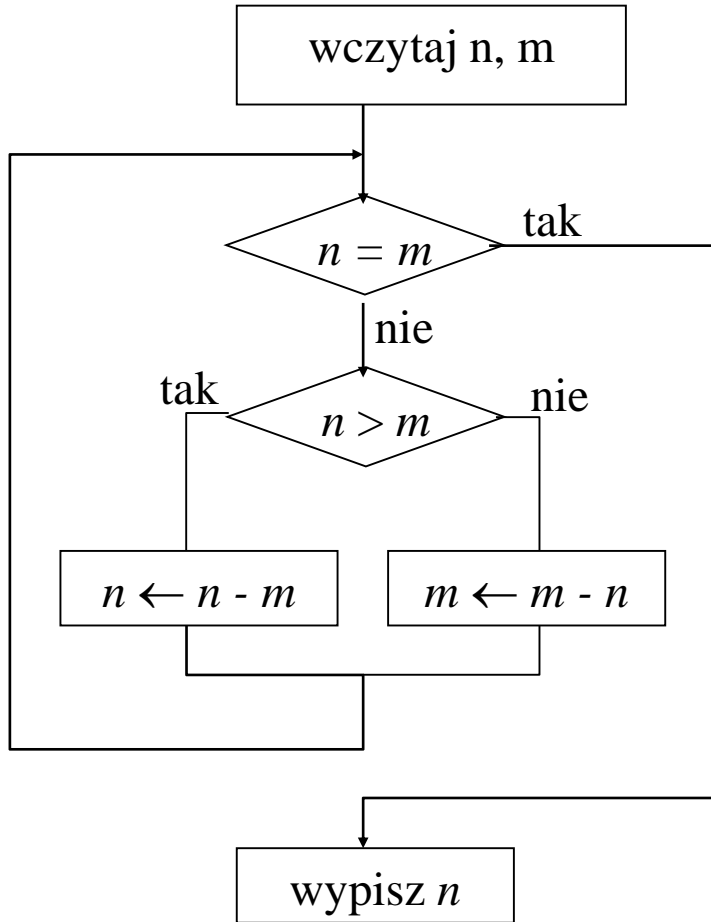
1. Wczytaj n, m

2. Dopóki $n \neq m$

- jeśli $n > m$: $n \leftarrow n - m$
- w przec. przyp.: $m \leftarrow m - n$
- wypisz n



nwd: algorytm Euklidesa v.1



```
main()
{
  int n,m;
```

```
  scanf("%d %d", &n, &m);
  while (n != m)
    if (n > m) n = n - m;
    else m = m - n;
  printf("Wynik: %d\n", n);
}
```

```
n=input("pierwsza:\n")
m=input("druga:\n")
while n != m:
    if n>m:
        n = n - m
    else:
        m = m - n
print "Wynik: ", n
```

Alg.Euklidesa v.1: złożoność czasowa

- jeden obrót pętli: parę (n', m') zastępujemy przez
 - $(m' - n', n')$ gdy $m' > n'$ oraz
 - $(n' - m', m')$ gdy $m' < n'$
- niech $\underline{R} = \underline{n' + m'}$
- wówczas obrót pętli powoduje zmniejszenie \underline{R} o $\min(n', m') \geq 1$
- na koniec działania algorytmu:
 $n' = m' = \text{nwd}(n, m)$,
więc $\underline{R} = 2 \cdot \text{nwd}(n, m) \geq \underline{2}$
- CZAS: $O(n + m - 2) = O(n + m)$

Alg.Euklidesa v.1: złożoność

Najgorszy przypadek: n duże, $m=1$:

- redukcja w jednym kroku $(n, 1) \rightarrow (n - 1, 1)$
- liczba iteracji (obrotów pętli): $n+m-2 = n-2$

Uwaga!

- Poprzedni slajd uzasadnia, że złożoność alg. Euklidesa v.1 jest $O(n+m)$
(nie jest większa!)
- Powyższy przykład ilustruje, że rzeczywiście ta złożoność może być aż tak duża, czyli „rzędu” $O(n+m)$.
(w przyszłości poznacie notację Ω opisującą tą kwestię)

nwd: algorytm Euklidesa v.2

Obserwacja

Jeśli $n = a \cdot m + b$ dla naturalnych a, b oraz $b < m$ to:

$$\begin{aligned} \text{nwd}(n, m) &= \text{nwd}(n - m, m) = \text{nwd}(n - 2m, m) = \\ &= \text{nwd}(n - 3m, m) = \dots = \text{nwd}(n - am, m) = \text{nwd}(b, m) \end{aligned}$$

Czyli: dla wejścia $n = a \cdot m + b$ oraz m :

Algorytm Euklidesa v.1 a -krotnie „odejmuje m ”!

Z drugiej strony:

- $a = \lfloor n / m \rfloor$ // dzielenie całkowite
- $b = n \bmod m$ // reszta z dzielenia

nwd: algorytm Euklidesa v.2

Wniosek

Niech $m < n$ to liczby naturalne. Wówczas

$$\text{nwd}(n, m) = \text{nwd}(n \bmod m, m)$$

Przykład

$$\begin{aligned} \text{nwd}(46, 18) &= \text{nwd}(46 \bmod 18, 18) = \\ \text{nwd}(18, 10) &= \text{nwd}(18 \bmod 10, 10) = \\ \text{nwd}(10, 8) &= \text{nwd}(10 \bmod 8, 8) = \\ \text{nwd}(8, 2) &= \text{nwd}(2, 0) \end{aligned}$$

nwd: algorytm Euklidesa v.2

Wniosek

1. $\text{nwd}(m, 0) = \text{nwd}(0, m) = m$
2. $\text{nwd}(n, m) = \text{nwd}(n \bmod m, m)$ gdy $n > m > 0$.
3. $\text{nwd}(n, m) = \text{nwd}(m, n)$.

Algorytm (pseudokod)

1. wczytaj n, m
2. jeśli $n < m$: zamień(n, m)
3. dopóki $m > 0$:
 - $n \leftarrow n \bmod m$
 - zamień(n, m)
4. wypisz n

nwd: algorytm Euklidesa v.2

Algorytm (pseudokod)

1. wczytaj n , m
2. jeśli $n < m$: zamień(n, m)
3. dopóki $m > 0$:
 - $n \leftarrow n \bmod m$
 - zamień(n, m)
4. wypisz n

nwd: algorytm Euklidesa v.2

Algorytm (pseudokod)

1. wczytaj n, m
2. jeśli $n < m$: zamień(n, m)
3. dopóki $m > 0$:
 - $n \leftarrow n \bmod m$
 - zamień(n, m)
4. wypisz n

```
main()
{
    int n,m,k;

    scanf("%d %d", &n, &m);
    if (n<m) {
        k = m; m = n; n = k; }
    while (m>0) {
        n = n % m;
        k = n;
        n =m;
        m =k;
    }
    printf("Wynik: %d\n", n);
}
```

nwd: algorytm Euklidesa v.2

Algorytm (pseudokod)

1. wczytaj n , m
2. jeśli $n < m$: zamień(n, m)
3. dopóki $m > 0$:
 - $n \leftarrow n \bmod m$
 - zamień(n, m)
4. wypisz n

```
n=input("pierwsza:\n")
m=input("druga:\n")
if n<m:
    k = m
    m = n
    n = k
while m>0:
    n = n % m
    k = n
    n =m
    m =k
print "Wynik: ", n
```

nwd: algorytm Euklidesa v.2 (sprytniej)

Algorytm (pseudokod)

1. wczytaj n, m
2. jeśli $n < m$: zamień(n, m)
3. dopóki $m > 0$:
 - $n \leftarrow n \bmod m$
 - zamień(n, m)
4. wypisz n

```
main()
{
    int n,m,k;

    scanf("%d %d", &n, &m);
    if (n<m) {
        k = m; m = n; n = k; }
    while (m!=0) {
        k = n % m;
        n = m;
        m = k;
    }
    printf("Wynik: %d\n", n);
}
```


nwd: algorytm Euklidesa v.2 (sprytniej)

Algorytm (pseudokod)

1. wczytaj n , m
2. jeśli $n < m$: zamień(n, m)
3. dopóki $m > 0$:
 - $n \leftarrow n \bmod m$
 - zamień(n, m)
4. wypisz n

```
n=input("pierwsza:\n")
```

```
m=input("druga:\n")
```

```
if n<m:
```

```
    k = m
```

```
    m = n
```

```
    n = k
```

```
while m>0:
```

```
    k = n % m
```

```
    n = m
```

```
    m = k
```

```
print "Wynik: ", n
```

Uwaga:

Instrukcja

$n, m = m, n$

nie wykonuje się „magicznie” bez pomocniczej „komórki”

nwd: alg. Euklidesa v.2

Kwestia implementacyjna

•**Pyt.:** Dlaczego zamieniamy n z m w każdej iteracji?

Odp.: ponieważ $(n \bmod m) < m$ dla każdego n, m

•**Pyt.:** Dlaczego sprawdzamy warunek $n > m$ tylko raz?

Odp.: Zamiana w pętli gwarantuje spełnienie: $m < n$ po każdym jej obrocie.

Alg. Euklidesa v. 2.0 - złożoność

Pamięć: $O(1)$

Czas:

Fakt. Jeśli $n \geq m$ to $n \bmod m \leq n/2$.

Dowód.

Przypadek 1: $n \geq 2m$

$$n \bmod m < m = 2m/2 \leq n/2$$

Przypadek 2: $m \leq n < 2m$

$$n \bmod m = n - m \leq n - n/2 = n/2$$

Alg. Euklidesa v. 2.0 - złożoność

Obserwacja 1

Każde iteracja (obrót) pętli „zamienia” parę (n, m) na taką parę, że:

- jedna z liczb to minimum z n, m ;
- druga z liczb jest nie większa od **połowy** z maksimum z n i m .

Obserwacja 2

Alg. Euklidesa v.2.0 kończy działanie gdy mniejsza z liczb n, m to zero.

Alg. Euklidesa v. 2.0 - złożoność

Wniosek

Alg. Euklidesa v.2.0 działa w czasie

$$O(\log_2 n + \log_2 m) =$$

$$O(\log_2 (n \cdot m)) =$$

$$O(\log_2 (n+m))$$

Pytania kontrolne:

- Potrafisz uzasadnić powyższe równości?
- Potrafisz uzasadnić, że po $O(\log n)$ „dzieleniach przez 2 liczby n ” uzyskamy zero?

Podsumowanie

1. Uruchamianie programów napisanych w językach wysokiego poziomu:
 1. Kompilacja lub translacja
 2. Stos wywołań funkcji
2. Algorytm Euklidesa jako przykład nieoczywistego rozwiązania problemu algorytmicznego
 1. Złożoność czasowa algorytmu Euklidesa jest „logarytmiczna”, rozwiązania naiwnego „liniowa” [to **baardzo** duża różnica]