# 1. Wprowadzenie

## 1.1 Algorytm, analiza i projektowanie algorytmów

Algorytmy komputerowe i struktury danych to z jednej strony dział informatyki teoretycznej (a więc matematyki) ale jednocześnie i szeroko pojętej elektroniki. Coraz częściej bowiem algorytmy komputerowe obok naturalnych realizacji programowych realizowane są również sprzętowo w postaci specjalizowanych układów scalonych. Jeśli zależy nam na szybkości algorytmu to sięgamy często po rozwiązania sprzętowe.

Algorytmy komputerowe i struktury danych nazywa się też czasami algorytmiką.

Algorytmy komputerowe i struktury danych to przedmiot podstawowy należący współcześnie do zasadniczego wykształcenia każdego inżyniera elektronika, inżyniera informatyka i inżyniera telekomunikacji.

**Algorytm** jest dobrze określoną procedurą obliczeniową, która wychodząc z dowolnych danych wejściowych (ang. input data) oblicza w skończonej ilości kroków dane wyjściowe (ang. output data). Dane wejściowe należą przy tym do pewnego ustalonego zbioru danych dopuszczalnych a dane wyjściowe do pewnego ustalonego zbioru możliwych danych wyjściowych.

Algorytm powinien zawsze zatrzymywać się po skończonej liczbie kroków. Powinien mieć dobrze określoną, dobrze zdefiniowaną tzw. regułę stopu.

Algorytm inaczej mówiąc to precyzyjny przepis, który podaje jak przejść w skończonej liczbie kroków od danych wejściowych do danych wyjściowych. Uzyskanie danych wyjściowych jest celem działania algorytmu a same dane wyjściowe stanowią na ogół rozwiązanie jakiegoś zadania.

**Uwaga 1.** Użyte w powyższej definicji sformułowanie „dobrze określona procedura obliczeniowa" jest z matematycznego punktu widzenia nieprecyzyjne. Można jednak ująć to ściśle wykorzystując formalne modele obliczeń takie jak maszyny Turinga lub maszyny RAM (ang. *random-access machines*). Prościej jest jednak **myśleć o algorytmie** jako programie komputerowym (napisanym w ustalonym **języku programowania** dla konkretnego komputera), który pobiera dane wejściowe i po skończonej ilości kroków wyprowadza dane wyjściowe.

**Uwaga 2.** Staramy się oczywiście dla rozwiązania danego problemu obliczeniowego znaleźć algorytm najbardziej **efektywny** tzn. najszybszy i o możliwie skromnym zapotrzebowaniu na pamięć. Do oceny efektywności algorytmu służy tzw. złożoność obliczeniowa.

**Czas** jaki algorytm poświęca na obliczenie danych wyjściowych zależy oczywiście od **„rozmiaru danego problemu"** a dokładniej od rozmiaru danych wejściowych. Również jednostka czasu powinna być sprecyzowana, jeśli chcemy porównywać algorytmy.

**Uwaga 3.** Algorytm komputerowy może być realizowany na komputerze uniwersalnym, takim jak np. IBM PC, ale może być również realizowany w specjalizowanym układzie specjalnie zaprojektowanym dla danego algorytmu. Z reguły układ taki projektuje się tak by wykorzystać możliwości zrównoleglenia obliczeń tkwiące w samym algorytmie.

**Zasadnicze cechy algorytmu:**

1. Algorytm powinien być zawsze poprawny (to znaczy poprawny dla każdego z założonego dopuszczalnego zestawu danych wejściowych)

2. Algorytm powinien się zawsze kończyć po skończonej liczbie kroków tzn. powinna istnieć pewna poprawnie działająca reguła stopu algorytmu.

3. Efektywność algorytmu (czy też klasy rozważanych algorytmów) a dokładniej tzw. złożoność obliczeniowa algorytmu jest jego zasadniczym parametrem. Pozwala ona naszkicować linię pomiędzy tym co możliwe a tym co niemożliwe dla danego algorytmu.

Czasami szczególnie interesuje nas wykazanie, że algorytmu nie da się praktycznie wykonać dla danych wejściowych o rozmiarze większym od pewnego $n_0$.

**Analiza i projektowanie algorytmów**

Nasz wykład dotyczy nie tylko różnych typów użytecznych algorytmów (np. algorytmów sortowania, algorytmów tekstowych czy algorytmów grafowych) ale również ogólnie rzecz biorąc: badania, analizy i projektowania algorytmów komputerowych (ang. *design and analysis of computer algorithms*) metodami teoretycznymi.

1. Analiza algorytmu polega na oszacowaniu tzw. złożoności obliczeniowej algorytmu (czasowej i przestrzennej). Krótko mówimy o koszcie algorytmu

2. Projektowanie algorytmu polega na minimalizacji kosztu algorytmu

Algorytm jest realizowany przez komputer lub jak czasem mówimy maszynę. Będziemy zakładać że maszyna na której realizowane są nasze algorytmy jest zawsze maszyną RAM (Random Access Machine).

Jest to nasz punkt odniesienia. Jednocześnie jest to realizacja koncepcji maszyny von Neumanna. Można powiedzieć, że RAM jest najbardziej typową z typowych maszyn.

**Uwaga.** RAM to jednocześnie skrót od Random Access Memory. Nie powinno to mylić Czytelnika.

Cechy maszyny RAM:
- rozkazy wykonywane są sekwencyjnie
- zbiór rozkazów zawiera rozkazy arytmetyczne, logiczne, porównania, skoki

Zakładamy też, że wszystkie rozkazy wykonywane są w 1 jednostce czasu.

# 1.2 Złożoność obliczeniowa-podstawowe pojęcia

**Rozmiar danych wejściowych** *n* dla danego algorytmu (ang. size of the input) jest **liczbą bitów potrzebnych** do reprezentowania danych wejściowych w kodzie NKB (naturalnym kodzie binarnym) przy użyciu odpowiedniego schematu kodowania.

Nieco ogólniej. Dla danego algorytmu rozmiarem danych wejściowych jest liczba znaków z ustalonego alfabetu niezbędnych do reprezentowania tych danych.

**Uwaga1.** Zasadniczo rozmiar *n* danych wejściowych podajemy w bitach. Najczęściej jednak jako rozmiar danych wejściowych przyjmujemy po prostu liczbę pozycji czy dokładniej długość ciągu skończonego stanowiącego dane wejściowe. Tak z reguły będzie np. w problemie sortowania gdzie za rozmiar danych wejściowych wygodnie jest przyjąć długość ciągu sortowanego.

**Uwaga** 2 Sprawa rozmiaru danych w konkretnych przypadkach z reguły nie budzi wątpliwości. Ogólna jednak definicja rozmiaru danych wejściowych jako długości słowa binarnego reprezentującego dane wymaga pewnego komentarza. Można bowiem w sposób sztuczny wydłużać słowo reprezentujące dane. Moglibyśmy np. zastosować do reprezentacji danych kod jedynkowy. Otóż zakładamy, że użyty do reprezentacji danych kod jest maksymalnie oszczędny jeśli chodzi o długość słowa kodowego.

W dalszy ciągu sufit liczby $x \in R$ oznaczamy symbolem $\lceil x \rceil$, a podłogę symbolem $\lfloor x \rfloor$. Przypominamy, podłoga (ang. floor) jest funkcją $\lfloor \cdot \rfloor : R \to Z$ zdefiniowaną wzorem $\lfloor x \rfloor \stackrel{df}{=} \max\{n \in Z; n \le x\}$ a sufit (ang. cealing) jest funkcją $\lceil \cdot \rceil : R \to Z$ zdefiniowaną wzorem $\lceil x \rceil \stackrel{df}{=} \min\{n \in Z; n \ge x\}$.

**Przykłady (rozmiar danych wejściowych)**

1.Liczba bitów w słowie kodowym (kodu NKB) reprezentującym liczbę $m \in N$ jest równa $\lfloor \log_2 m \rfloor + 1$. Za rozmiar $n$ liczby $m$ będziemy więc przyjmować $\lfloor \log_2 m \rfloor + 1$ lub dla uproszczenia niezbyt ściśle $\log_2 m$ lub $\ln m$.

2.Jeśli $f$ jest wielomianem stopnia $\le k$ i każdy współczynnik tego wielomianu jest liczbą naturalną $\le n$, to rozmiar danych reprezentowanych wielomianem $f$ jest równy $(k+1)\cdot(\lfloor \log_2 n \rfloor + 1)$ . Wygodnie jest jednak jako rozmiar danych przyjmować w tym przypadku $(k+1)$.

3.Jeżeli *A* jest macierzą z *k* wierszami i *m* kolumnami o współczynnikach będących liczbami naturalnymi $\le n$, to rozmiar danych reprezentowanych macierzą *A* jest równy $k \cdot m \cdot (\lfloor \log_2 n \rfloor + 1)$. Wygodnie jest jednak jako rozmiar danych przyjmować w tym przypadku $k \cdot m$.

4.Jeśli sortujemy (czyli porządkujemy) ciąg $a_1, a_2, ..., a_k$ i wyrazy tego ciągu są liczbami naturalnymi $\leq n$, to rozmiar bitowy danych wejściowych jest równy $k \cdot (\lfloor \log_2 n \rfloor + 1)$. Wygodnie jest jednak jako rozmiar danych przyjmować w tym przypadku po prostu *k*.

Przyjmujemy na ogół, że wszystkie różne dane wejściowe o ustalonym rozmiarze *n* są jednakowo prawdopodobne.

## Definicja złożoności obliczeniowej i rodzaje złożoności obliczeniowej

**Złożoność obliczeniową** (ang. *computational complexity*) danego algorytmu definiujemy jako zasoby systemu komputerowego niezbędne do realizacji tego algorytmu. Zasobami jakie bierzemy pod uwagę są czas i pamięć. Rozróżniamy więc dwa rodzaje złożoności obliczeniowej **złożoność czasową algorytmu** i **złożoność pamięciową algorytmu.**

Złożoność pamięciowa jest pojęciowo jasna. Mierzymy ją w bajtach B, kilobajtach kB czy megabajtach MB.

Nieco większe kłopoty mamy ze zdefiniowaniem złożoności czasowej (ang. *the running time*),. Czas wykonania algorytmu zależy bowiem i od algorytmu i od systemu komputerowego realizującego ten algorytm i od danych.

Żeby uniezależnić się w ocenie algorytmu od sprzętu realizującego ten algorytm za jednostkę złożoności czasowej przyjmujemy wykonanie jednej operacji dominującej w algorytmie.

Na przykład za taką operację w algorytmach sortowania uważamy porównanie dwóch liczb. Im algorytm sortowania potrzebuje mniej porównań tym lepiej, tym ma mniejszą złożoność czasową.

Ogólnie rzecz biorąc czas wykonania algorytmu dla określonych danych wejściowych jest liczbą elementarnych działań lub kroków wykonywanych przy realizacji algorytmu. Słowo krok może tu oznaczać wykonanie działania na bitach, porównanie dwóch liczb, wykonanie instrukcji maszynowej, okres zegara taktującego procesu, wykonanie mnożenia czy dodawania modulo. Najwygodniej jednak oceniać złożoność czasową algorytmów w języku operacji dominujących.

Złożoność obliczeniowa (czasowa i przestrzenna) jest funkcją rozmiaru danych *n* i na ogół zależy też od samych danych.

W celu precyzyjnego zdefiniowania złożoności czasowej algorytmu wprowadzamy następujące oznaczenia.

Przez $D_n$ oznaczamy zbiór wszystkich zestawów danych wejściowych o rozmiarze *n*. Zbiór $D_n$ jest na ogół zbiorem skończonym choć tak być nie musi.

$t(d)$ - to liczba operacji dominujących (czas – time) podczas realizacji algorytmu dla pewnego zestawu danych wejściowych $d \in D_n$.

$X_n$ - to zmienna losowa o wartościach w zbiorze liczb naturalnych określona na przestrzeni probabilistycznej $(D_n, 2^{D_n}, P_n)$. Rozkład prawdopodobieństwa $P_n$ jest z reguły rozkładem równomiernym na przestrzeni $D_n$. Zmienna losowa $X_n$ przyjmuje dla zdarzenia elementarnego $d \in D_n$ wartość $t(d)$.

$p_{n,k}$ - to prawdopodobieństwo tego, że (przy ustalonym rozmiarze danych $n$ algorytmu) wykonano $k$ operacji dominujących $k \in N \cup \{0\}$. Mamy więc $p_{n,k} = P_n(X_n = k)$.

**Definicja pesymistycznej złożoności czasowej algorytmu.** Dla ustalonego rozmiaru danych $n$ oznaczamy przez $T_w(n)$ pesymistyczną złożoność czasową algorytmu tzn.

$$T_w(n) \stackrel{df}{=} \sup\{t(d); d \in D_n\}$$

(indeks $w$ bo chodzi o najgorszy przypadek „ the worst case")

Pesymistyczna złożoności czasowej algorytmu (ang. *the worst case running time*) jest więc kresem górnym czasów obliczeń dla dowolnych danych wejściowych o ustalonym rozmiarze $n$ . Pesymistyczna złożoności czasowa algorytmu jest rozpatrywana jako funkcja rozmiaru danych wejściowych $n$. Możemy więc napisać: $T_w : N \to N$.

Pesymistyczną złożoność czasową nazywa się również złożonością czasową w najgorszym przypadku.

**Definicja średniej złożoności czasowej algorytmu**

Średnia złożoność czasowa $T_{ave}$ algorytmu lub jak mówimy oczekiwana złożoność czasowa algorytmu (ang. *the average case running time*) jest uśrednionym czasem po wszystkich możliwych danych wejściowych o ustalonym rozmiarze $n$ jest więc to średni czas obliczeń algorytmu.

Średnia złożoność czasowa algorytmu jest rozpatrywana jako funkcja rozmiaru danych wejściowych $n$. Możemy więc napisać: $T_{ave} : N \to R$. Średnią złożoność czasową algorytmu dla danych rozmiaru $n$ oznaczamy przez $T_{ave}(n)$.

$$T_{ave}(n) \stackrel{df}{=} E(X_n) = \sum_{k \in N \cup \{0\}} k \cdot p_{n,k}$$

(indeks *ave* bo chodzi o wartość średnią „ the average")

Warto podkreślić, że wartość $T_{ave}(n)$ zależy od rozkładu $P_n$. Najczęściej przyjmujemy, że wszystkie dane wejściowe o rozmiarze $n$ są jednakowo prawdopodobne tzn., że $P_n$ jest rozkładem równomiernym na przestrzeni $D_n$.

**Miary wrażliwości na dane wejściowe.** Oznaczamy przez $\Delta(n)$ tzw. miarę wrażliwości pesymistycznej

$$\Delta(n) = \sup\{t(d_1) - t(d_2); d_1, d_2 \in D_n\}$$

Przez $\sigma(n)$ oznaczamy średnie odchylenie kwadratowe zmiennej losowej $X_n$ tzn.

$$\sigma(n) = \sqrt{D^2(X_n)} = \sqrt{\sum_{k \in N \cup \{0\}} (k - T_{ave}(n))^2 P_{n,k}}$$

Za pomocą liczb $\Delta(n)$ i $\sigma(n)$ oceniamy wrażliwość algorytmu na dane wejście.

Podamy trzy przykłady ilustrujące pojęcie średniego czasu wykonania algorytmu.

**Przykład 1.** Rozważmy nieskończony ciąg doświadczeń Bernoulliego z prawdopodobieństwem sukcesu $p \in (0,1)$. Dla ustalenia uwagi załóżmy, że rzucamy niesymetryczną monetą do pierwszego wyrzucenia orła. Wyrzucenie reszki to „1" (sukces), orła to „0". Wyrzucenie reszki (czyli 1) odbywa się z prawdopodobieństwem $p \in (0,1)$. Program dokonuje ciągu niezależnych losowań. Algorytm zatrzymuje się gdy wylosuje pierwsze 0. Naszym zadaniem jest ocena złożoności czasowej algorytmu.

---

**Algorytm:** Ciąg prób Bernoulliego z prawdopodobieństwem sukcesu $p \in (0,1)$

**Dane wejściowe**: Prawdopodobieństwo sukcesu $p \in (0,1)$

**Dane wyjściowe**: Długość serii jedynek przed pojawieniem się pierwszego zera.

---

```
procedure ciag_prob_Bernoulliego(p)
begin
m: =0 ;
wynik_rzutu: =1 ;
while  wynik_rzutu=1
begin
if random<p then wynik_rzutu:=1 else wynik_rzutu:=0;   // Funkcja random losuje liczbę z przedziału (0,1)
if  wynik_rzutu =1 then m:=m+1; // z prawdopodobieństwem wyboru 1 równym p ∈ [0,1]
end ;
write(m) ;
end
```

■

Zmienna *m* podaje liczbę pierwszych 1 do pojawienia się pierwszego 0. Jeśli wylosujemy np. ciąg 1110 to *m*=3 i algorytm kończy się po 4 wywołaniach procedury *random*.

Można przyjąć, że operacją dominującą jest losowanie 0 lub 1. Zatem czas wykonania algorytmu jest wprost proporcjonalny do liczby wykonań pętli **while** a więc jednocześnie wprost proporcjonalny do wartości $m + 1$.

Pesymistyczna złożoność czasowa algorytmu jest oczywiście nieskończona tzn. niezależnie od rozmiaru danych $T_w = +\infty$ ale prawdopodobieństwo takiego zdarzenia jest równe 0. Załóżmy, że liczbę wykonań pętli **while** opisuje zmienna losowa *X*. Mamy zatem niezależnie od rozmiaru danych

$$T_{ave} = E(X) = \sum_{k=1}^{\infty} kp^k$$

Obliczmy sumę szeregu liczbowego $\sum_{k=1}^{\infty} kp^k$ .

$$T_{ave} = E(X) = \sum_{k=1}^{\infty} kp^k = p\sum_{k=0}^{\infty}(k+1)p^k = p(\sum_{k=0}^{\infty} x^k)'\Big|_{x=p} = p(\frac{1}{1-x})'\Big|_{x=p} = p(\frac{1}{(1-x)^2})'\Big|_{x=p} = \frac{p}{(1-p)^2}$$

Zatem $T_{ave} = \frac{p}{(1-p)^2}$ co dla symetrycznego rzutu monetą tzn. dla $p = 1/2$ daje $T_{ave} = 2$.

Ten sam algorytm zapisany w Matlabie ma postać:

---

**Algorytm:** Ciąg prób Bernoulliego z prawdopodobieństwem sukcesu $p \in (0,1)$

**Dane wejściowe**: Prawdopodobieństwo sukcesu $p \in (0,1)$

**Dane wyjściowe**: Długość serii jedynek przed pojawieniem się pierwszego zera.

---

```
function wart_1=ciag_prob_Bernoulliego(p)
m=0, wynik_rzutu=1
while wynik_rzutu==1, if rand(1)<p, wynik_rzutu=1, else wynik_rzutu=0, end
    if wynik_rzutu==1, m=m+1, end
end
wart_1=m
```

■

**Przykład 3.** Schemat Hornera. Schemat Hornera (lub algorytm Hornera) służy do obliczania wartości wielomianu $a_n x^n + a_{n-1}x^{n-1} + ... + a_0$ w punkcie $x$. Schemat Hornera sprowadza się do wykorzystania do obliczeń tożsamości:

$$a_n x^n + a_{n-1}x^{n-1} + ... + a_0 = (...((a_n x + a_{n-1})x + a_{n-2})x + ... + a_1)x + a_0$$

---

**Algorytm:** Schemat Hornera obliczania wartości wielomianu $f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$ stopnia $n \geq 1$ dla danego argumentu $x$.

**Dane wejściowe**: Tablica $A[1..(n+1)]$ współczynników wielomianu $f$, $A[n+1] = a_n$, $A[n] = a_{n-1}$, $A[1] = a_0$

**Dane wyjściowe**: Wartość $f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$

---

**procedure** *schemat_Hornera*(*A*,*x*,*n*)

**begin**
*s*: =*A*[*n*+1] ;
**for** *i*:=*n* **downto** 1 **do** *s*: =*s***x* +*A*[*i*] ;
write(*s*)
**end**

Za operację dominującą przyjmujemy w powyższym algorytmie mnożenie. Złożoność czasowa algorytmu zależy tylko od rozmiaru danych wejściowych. Ponieważ mamy w algorytmie dla ustalonego $n$: $n$ obiegów pętli **for** i $n$ mnożeń zatem $T_w(n) = n$ oraz $T_{ave}(n) = n$. Zatem jak mówimy algorytm ma liniową złożoność obliczeniową.

Złożoność przestrzenna algorytmu jest stała (mówimy w takiej sytuacji, że algorytm działa w miejscu: „in place"). W szacowaniu złożoności przestrzennej algorytmu pomijamy miejsce niezbędne do pamiętania danych wejściowych.

**Uwaga.** Schemat Hornera działa poprawnie nie tylko dla wielomianów o współczynnikach rzeczywistych ale również dla wielomianów o współczynnikach w dowolnym ciele.

Schemat Hornera można zapisać w Matlabie następująco:

---

**Algorytm:** Schemat Hornera obliczania wartości wielomianu $f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$ stopnia $n \geq 1$ dla danego argumentu $x$.

**Dane wejściowe**: Tablica $A[1..(n+1)]$ współczynników wielomianu $f$, $A[n+1] = a_n$, $A[n] = a_{n-1}$, $A[1] = a_0$

**Dane wyjściowe**: Wartość $f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$

---

```
function wart_1=schemat_Hornera(A,x,n)
s=A(n+1)
for i=n:-1:1, s=s*x+A(i), end
wart_1=s
```

■

**Przykład 3.** Przeanalizujemy teraz złożoność obliczeniową podanego w dalszym ciągu algorytmu i pokażemy jak analiza czasowej złożoności algorytmu zależy od rozkładu prawdopodobieństwa danych wejściowych.

W danych wejściowych mamy dwie liczby $a, b \in R$, przedział $[a,b]$ i ciąg liczb rzeczywistych $e_0, e_1, ...e_n, e_{n+1}$ taki, że $a = e_0 < e_1 < .. < .e_n < e_{n+1} = b$ zapisany w tablicy $e[0..(n+1)]$ oraz liczbę $x \in [a,b]$. Przedziałowi $[e_i, e_{i+1})$ przyporządkowujemy numer $i$. Algorytm ma podawać jako dane wyjściowe numer przedziału, do którego należy liczba $x \in [a,b]$.

---

**Algorytm:** Podawanie numeru przedziału *i* do którego należy dana wejściowa $x \in [a,b]$

**Dane wejściowe**: Liczby $a, b \in R$, przedział $[a,b]$, ciąg liczb rzeczywistych $e_0, e_1, ...e_n, e_{n+1}$ taki, że $a = e_0 < e_1 < .. < .e_n < e_{n+1} = b$ zapisany w tablicy $e[0..(n+1)]$ oraz $x \in [a,b]$

**Dane wyjściowe**: Numer przedziału, do którego należy liczba $x \in [a,b]$

---

```
procedure numer_przedziału(e,x,n)
begin
i: =0 ;
while x > e[i+1] do i: = i+1;
write(i)
end
```

Ten sam algorytm można zapisać w Matlabie następująco:

---

**Algorytm:** Podawanie numeru przedziału *i* do którego należy dana wejściowa $x \in [a,b]$

**Dane wejściowe**: Liczby $a,b \in R$, przedział $[a,b]$, ciąg liczb rzeczywistych $e_0, e_1, ...e_n, e_{n+1}$ taki, że $a = e_0 < e_1 < .. < .e_n < e_{n+1} = b$ zapisany w tablicy $e[0..(n+1)]$ oraz $x \in [a,b]$

**Dane wyjściowe**: Numer przedziału, do którego należy liczba $x \in [a,b]$

---

```
function wart_1=numer_przedzialu(e,x,n)
i=0
while x>e(i+1), i=i+1, end
wart_1=i
```

Za operację dominującą przyjmujemy porównanie. Niech ilość porównań (jednocześnie liczbę obiegów pętli **while**) opisuje zmienna losowa *X*.
Załóżmy, że prawdopodobieństwo tego, żę dana wejściowa *x* należy do dowolnego z *n*+1 przedziałów $[e_i, e_{i+1})$ jest równe $\frac{1}{n+1}$ wówczas

$$T_{ave}(n) = E(X) = \sum_{k=1}^{n+1} k \frac{1}{n+1} = 1 + \frac{n}{2}$$

Zatem średnia złożoność czasowa algorytmu wynosi $T_{ave}(n) = 1 + \frac{n}{2}$.

Największa liczba porównań jest równa liczbie przedziałów a więc wynosi pesymistyczna złożoność czasowa wynosi $T_w(n) = n + 1$.

Załóżmy teraz, że prawdopodobieństwo tego, żę *x* przyjmuje określoną wartość opisane jest rozkładem równomiernym na przedziale $[a,b]$ wówczas prawdopodobieństwo tego, że *x* należy do przedziału $[e_i, e_{i+1})$ jest równe $p_i = \frac{e_{i+1} - e_i}{b-a}$. Mamy wówczas

$$T_{ave}(n) = E(X) = \sum_{k=1}^{n+1} k p_{i-1} = \frac{1}{e_{n+1} - e_0} \sum_{k=0}^{n} (e_{n+1} - e_j)$$

■

# 1.3 Sposoby opisu algorytmu – język publikacyjny

Algorytm możemy opisać w następujący sposób:

1.w punktach w języku naturalnym

2.za pomocą schematu blokowego (czyli flowdiagramu)

3.w językach programowania np. w asemblerze, w C , w C++ , Pascalu czy w Matlabie

4.w tzw. języku publikacyjnym, czyli niezbyt formalnej odmianie języka programowania

Do w pełni formalnego, precyzyjnego zapisu algorytmów w sposób „zrozumiały" dla komputera służą głównie języki programowania (asembler, C, C++, Pascal, Java itd. ). Wygodniej jest jednak dla naszych celów opisywać algorytmy w tzw. języku publikacyjnym stanowiącym nieco uproszczoną, mniej formalną (ale czytelniejszą dla człowieka) odmianę języka programowania. W praktyce używany często do opisu algorytmów nieco „odformalizowanego" Pascala, C lub C++.

Język publikacyjny nazywany jest też pseudokodem. Pseudokod używany w skrypcie do opisu algorytmów to „mniej rygorystyczny Pascal". Staramy się jednak zachować szczegóły składni Pascala. Do opisów algorytmów używamy w dalszym ciągu również języka Matlab.

**Przykład 1.** Opis algorytmu w punktach w języku naturalnym.

---

**Algorytm:** **Przepis na robienie budyniu czekoladowego**

DANE WEJŚCIOWE: budyń czekoladowy w postaci proszku w torebce, ½ litra mleka, 3 łyżki cukru, łyżeczka masła

DANE WYJŚCIOWE: gotowy ciepły budyń

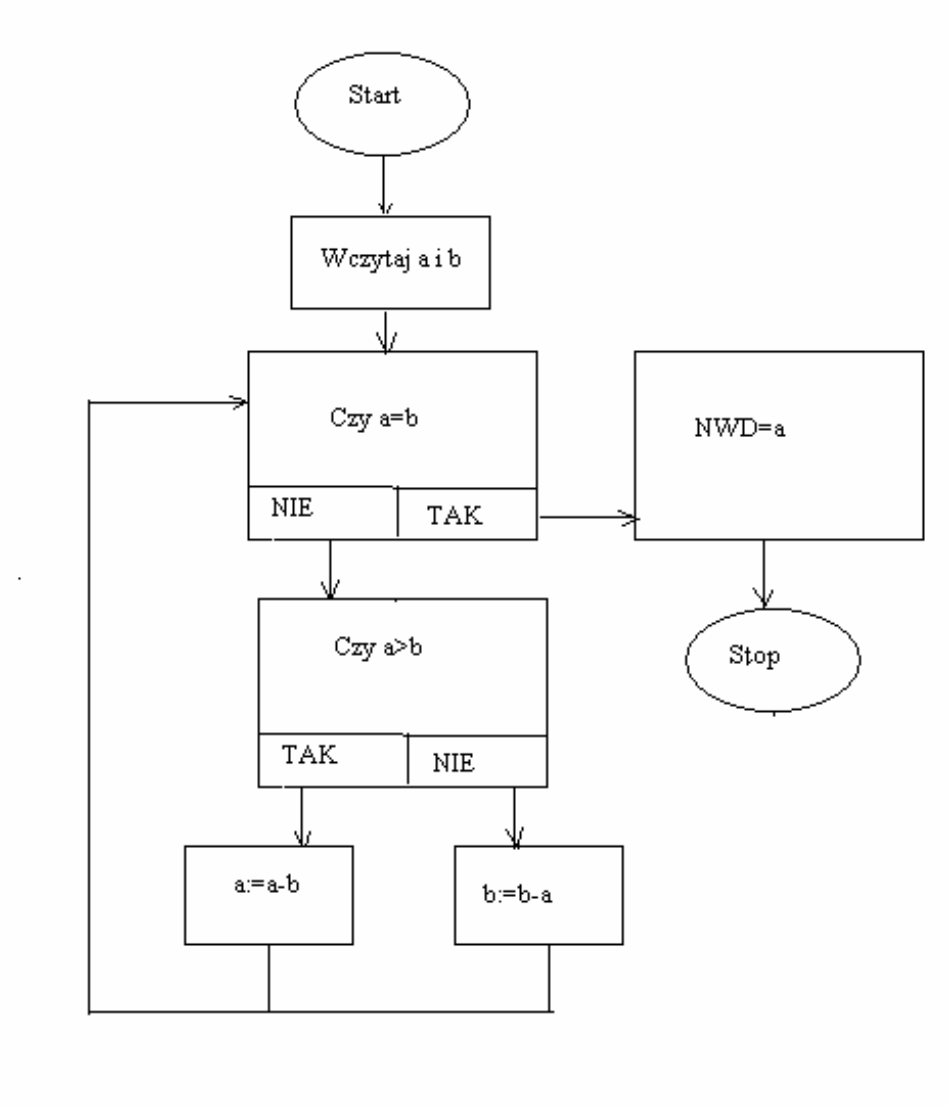
---

1. z pół litra mleka odlać pół szklanki
2. resztę mleka zagotować dodając 3 łyżki cukru i łyżeczkę masła
3. wsypać do szklanki zawartość torebki i dobrze wymieszać
4. miksturę otrzymaną w punkcie 3 wlać do zagotowanego mleka
5. gotować przez chwilę stale mieszając aż do momentu zgęstnienia (reguła stopu)

■

**Przykład 2** Opis algorytmu za pomocą schematu blokowego. Niech $a,b \in N$. Opiszemy schematem blokowym najprostszą wersję algorytmu Euklidesa wykorzystującą z działań arytmetycznych tylko odejmowanie. Klasyczny algorytm Euklidesa służy jak pamiętamy do obliczania $d = NWD(a,b)$ metodą dzielenia z resztą.

Algorytm opisywany oblicza $d = NWD(a,b)$ i oparty jest na równościach $NWD(a,b) = NWD(a,b-a) = NWD(b,a-b)$




Rys.1 Schemat blokowy algorytmu obliczania NWD z odejmowaniem jako jedynym działaniem arytmetycznym

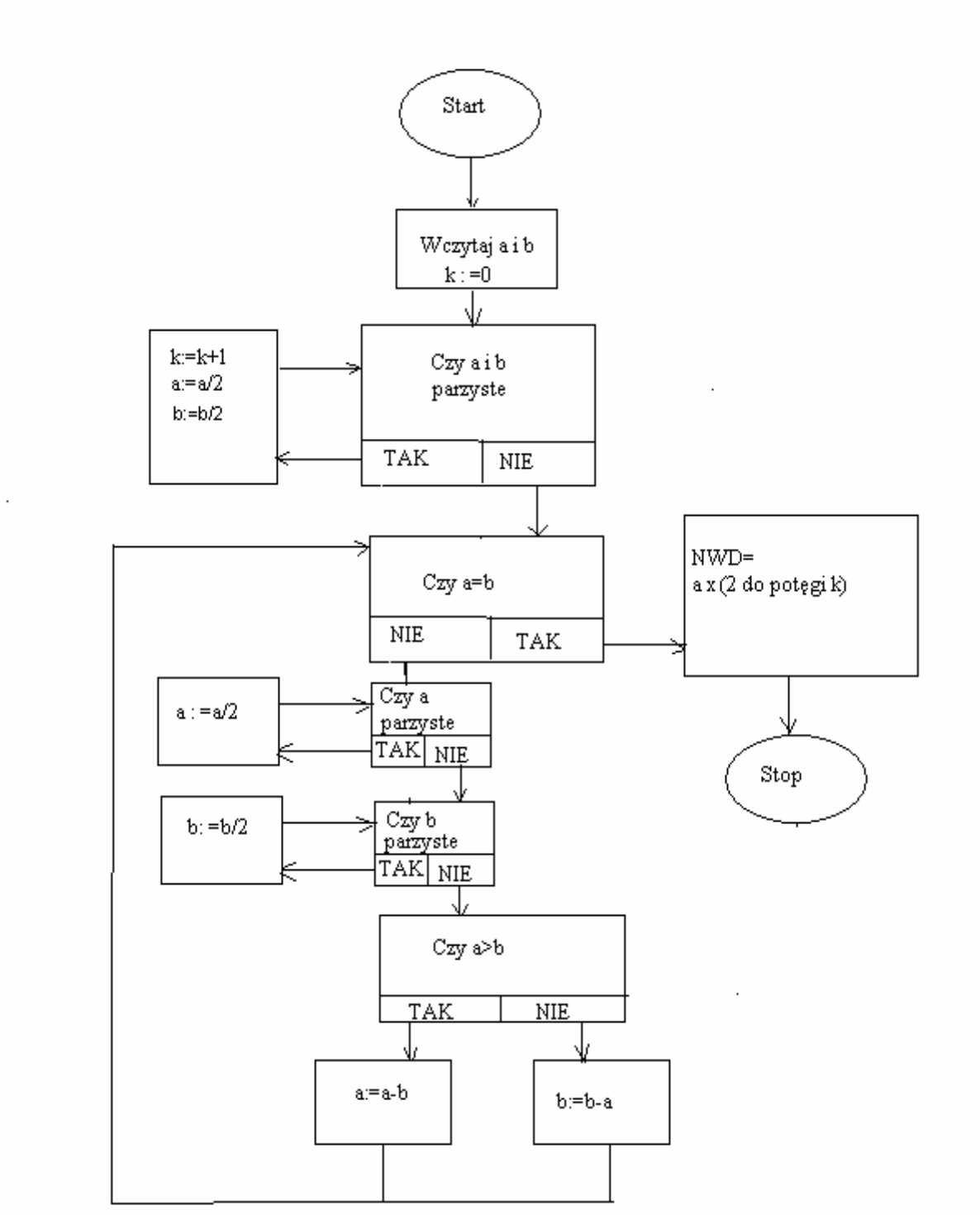
■

**Przykład 3** Podamy jeszcze jeden przykład opisu algorytmu za pomocą schematu blokowego. Niech $a,b \in N$. Algorytm opisywany tzw. algorytm Steina (nazywany też binarnym algorytmem Euklidesa) oblicza $d = NWD(a,b)$ i oparty jest na następujących prostych do dowodu faktach.

Jeśli $a = 2^k c$, $b = 2^k d$ gdzie $a,b,c,d,k \in N$ to $NWD(a,b) = 2^k NWD(c,d)$

Jeśli $a \in N$ jest liczbą parzystą a $b \in N$ nieparzystą to $NWD(a,b) = NWD(a/2,b)$

Jeśli $a,b \in N$ $NWD(a,b) = NWD(a,b-a) = NWD(b,a-b)$




Rys. 2 Binarny algorytm Euklidesa (algorytm Steina)

■

**Przykład 4** Klasyczny algorytm Euklidesa zapisany za pomocą publikacyjnego Pascala.

W poniższym algorytmie wykorzystujemy zależność

$$NWD(a,b) = NWD(b, a \bmod b)$$

gdzie $a$ jest nieujemną liczbą całkowitą a $b$ jest dodatnią liczbą całkowitą. W publikacyjnym a więc nieco nieformalnym Pascalu algorytm Euklidesa można zapisać następująco:

---

**Algorytm Algorytm Euklidesa dla liczb całkowitych nieujemnych**

Dane WEJŚCIOWE: dwie liczby całkowite $a, b \geq 0$
Dane WYJŚCIOWE: największy wspólny dzielnik $a$ i $b$ czyli $NWD(a,b)$

---

**procedure** $Euklid(a,b)$

**begin**
**if** $b = 0$ **then** $NWD(a,b) := a$ **else** $Euklid(b, a \bmod b)$

**end**

Podobnie już w pełni formalnie można zapisać algorytm Euklidesa w Matlabie:

---

**Algorytm Algorytm Euklidesa dla liczb całkowitych nieujemnych**

Dane WEJŚCIOWE: dwie liczby całkowite $a, b \geq 0$

Dane WYJŚCIOWE: największy wspólny dzielnik $a$ i $b$ czyli $NWD(a,b)$

---

```
function NWD=Euclid(a,b)
% funkcja Euclid(a,b) oblicza NWD(a,b) za pomoca algorytmu
% Euklidesa, a, b sa liczbami calkowitymi nieujemnymi
if b==0 NWD=a, else Euclid(b,mod(a,b)), end
```

Klasyczny algorytm Euklidesa działa poprawnie nie tylko dla pierścienia liczb całkowitych $Z$ ale również po pewnych modyfikacjach dla pierścienia wielomianów nad dowolnym ciałem. Ogólnie rzecz biorąc algorytm Euklidesa działa poprawnie dla tzw. pierścieni euklidesowych.

Opisany algorytm działa rekurencyjnie tzn. odwołując się do siebie.
■

**Przykład 5.** Algorytm sortowania bąbelkowego tzw. bubblesort zapisany za pomocą publikacyjnego Pascala.

---

**Algorytm: Sortowanie bąbelkowe (bubblesort)**

Dane WEJŚCIOWE: ciąg sortowany $A[1] = a_1, A[2] = a_2, ..., A[n] = a_n$

Dane WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le ... \le A[n] = a_{\pi(n)}$

---

**procedure** *bubblesort* (*A*,*n*)

**begin**
**for** $j$ =*n*-1 **downto** 1 **do**
**for** $i$ =1 **to** $j$ **do**
**if** $A[i+1] < A[i]$ **then** zamień miejscami $A[i]$ z $A[i+1]$
**end**

Można udowodnić, że procedura *bubblesort* sortuje czyli porządkuje elementy tablicy $A$ w porządku niemalejącym.

Algorytm sortowania bąbelkowego zapisać można w Matlabie następująco:

---

**Algorytm: Sortowanie bąbelkowe (bubblesort)**

Dane WEJŚCIOWE: ciąg sortowany $A[1] = a_1, A[2] = a_2, ..., A[n] = a_n$

Dane WYJŚCIOWE: ciąg posortowany $A[1] = a_{\pi(1)} \le A[2] = a_{\pi(2)} \le ... \le A[n] = a_{\pi(n)}$

---

```
function B=bubblesort(A,n)
for j=n-1:-1:1,
for i=1:1:j
   if A(i+1)<A(i), c=A(i), A(i)=A(i+1), A(i+1)=c, end

end
end
B=A
```

■

Trzy instrukcje Pascala służące do organizacji pętli, tzw. instrukcje iteracyjne będą w dalszym ciągu szczególnie ważne więc przypomnimy je krótko używając do opisu ich składni notacji Backusa-Naura (w skrócie notacji BNF).

<instrukcja iteracyjna> : := <instrukcja **for**> |<instrukcja **while**>|<instrukcja **repeat**>

**1.Instrukcja „for”.** Instrukcję „**for**” zapisujemy w notacji Backusa-Naura tak:

<instrukcja „for”> : : = **for** <zmienna sterująca> := <lista „for”> **do** <instrukcja>

<lista „for”> : : = <wartość początkowa> **to** <wartość końcowa>|
|<wartość początkowa> **downto** <wartość końcowa>

<zmienna sterująca> : : = identyfikator
<wartość początkowa> : : = <wyrażenie>
<wartość końcowa> : := <wyrażenie>

**Przykład.**

**for** $i$ : = 1 **to** $m$ **do** $A[i] := 0$

**for** $j$ : = 5 **downto** -5 **do** $a := a + j$

**2.Instrukcja „while”.** Instrukcję **„while”** definiujemy w notacji Backusa-Naura tak:

<instrukcja while> : : = **while** <wyrażenie> **do** <instrukcja>

Dopóki wyrażenie ma wartość logiczną **true** wykonywana jest instrukcja po **do**. Gdy wyrażenie stanie się równe **false** wychodzimy z pętli.

**Przykład**

**while** a>b **do** a : =a-b

**3.Instrukcja „repeat”.** Instrukcję „**repeat**” zapisujemy w notacji Backusa-Naura tak:

<instrukcja repeat> : : = **repeat** <instrukcja> {;<instrukcja>} **until** <wyrażenie>

Instrukcja „**repeat**” służy do powtarzania ciągów instrukcji ze sprawdzaniem warunku wyjścia z pętli na końcu. Wykonanie ciągu instrukcji zapisanego między symbolami **repeat** i **until** powtarza się co najmniej raz ale tak długo dopóki warunek po **until** nie zostanie spełniony.

**Przykład**

**repeat** a: =a/2 **until** a>100

Ponieważ w dalszym ciągu będziemy również używać do opisu algorytmów również języka MATLAB podajemy poniżej zasadnicze instrukcje MATLABA.

## Instrukcja warunkowa if

Instrukcja warunkowa czyli instrukcja **if** ma w najprostszej wersji następującą składnię:

**if** *wyrażenie_ warunkowe*

*ciąg instrukcji*

**end**

a w przypadku ogólnym mamy składnię następującą:

**if** *wyrażenie_ warunkowe*_1

*ciąg instrukcji*_1

**elseif** *wyrażenie_ warunkowe*_2
*ciąg instrukcji*_1

**elseif** *wyrażenie_ warunkowe*_3
*ciąg instrukcji*_1
…

**elseif** *wyrażenie_ warunkowe*_(*n* -1)

*ciąg instrukcji*_1

**else** *wyrażenie_ warunkowe_n*

*ciąg instrukcji*_1

**end**

Przypominamy, że wartość wyrażenia warunkowego jest w Matlabie macierzą.

## Instrukcja pętli for

Instrukcja pętli **for** ma następującą składnię:

**for** *zmienna_iterowana* = *macierz wartości*

*ciąg instrukcji*

**end**

**Przykład**

```
% program umieszczony w m-pliku i wykonany generuje liczby od 1 do 101
A=1
for i=1:100
A=A+1
end
```

## Instrukcja pętli while

Instrukcja pętli **while** ma następującą składnię:

**while** *wyrażenie_ warunkowe*

*ciąg instrukcji*

**end**

Wykonanie instrukcji **while** powoduje wykonanie ciągu instrukcji do momentu do którego wyrażenie warunkowe ma wszystkie elementy niezerowe

# 1.4 Zapisy asymptotyczne

Często trudno dokładnie podać czas obliczeń algorytmu, wygodnie jest wtedy oszacować czas obliczeń wprowadzając pojęcie **asymptotycznego czasu obliczeń** lub **asymptotycznej złożoności obliczeniowej**.

Badając ustalony algorytm chcemy ustalić, jak zachowuje się pesymistyczny czas obliczeń $T_w(n)$ i średni czas obliczeń $T_{ave}(n)$ w funkcji rozmiaru danych wejściowych $n$ dla $n$ rosnących do $\infty$.

Zapisy asymptotyczne to inaczej rzędy wielkości funkcji (ang. *asymptotic notation* lub *order notation*)

Krótkie, przejrzyste wprowadzenie w zagadnienia rzędów wielkości funkcji można znaleźć np. w pracach R.Neapolitana i K.Naimipour'a „Podstawy algorytmów z przykładami w C++" i podręczniku T.H. Cormen'a C.E.Leiserson'a R.L.Rivest'a i C.Stein'a „Wprowadzenie do algorytmów".

**NOTACJA O (notacja O duże).**

Niech $f, g : N \to R$. Piszemy $f(n) = O(g(n))$ wtedy i tylko wtedy z definicji, gdy istnieje taka stała rzeczywista $c > 0$ i liczba naturalna $n_0$, że $0 \le f(n) \le cg(n)$ dla każdego $n \ge n_0$.

Dokładniej, formalnie symbol $O(g(n))$ oznacza z definicji zbiór funkcji $f : N \to R$ zdefiniowany następująco

$\Theta(g(n)) = \{ f \in R^N$; istnieje taka stala $c > 0$ i $n_0 \in N$, ze $0 \le f(n) \le cg(n)$ dla kazdego $n > n_0 \}$

Powinniśmy więc pisać $f(n) \in O(g(n))$ zwyczajowo jednak będziemy używać zapisu $f(n) = O(g(n))$ (chociaż coraz częściej bywa jednak stosowany precyzyjniejszy zapis $f(n) \in O(g(n))$).

Sens notacji O to „asymptotyczne ograniczenie górne", sposób szacowania od góry wartości funkcji $f$ przez wartości funkcji $g$. Zauważmy jeszcze, że jeśli spełniona jest relacja $f(n) = O(g(n))$ to z definicji notacji „duże O" wynika, że funkcje $f, g : N \to R$ są jak mówimy asymptotycznie nieujemne tzn. istnieje taka liczba naturalna $n_0$, że dla każdego $n \ge n_0$ mamy $0 \le f(n)$ oraz $0 \le g(n)$.

**Przykład.** Jeśli dla każdego $n \in N$ mamy $f(n) = n$ i $g(n) = n^2$ to przyjmując $c = 1$ mamy dla każdego $n \in N : f(n) = n \le 1 \cdot n^2 = g(n)$ co zapisujemy jako $n = O(n^2)$. Dla $c = 1$ mamy również $n \le 1 \cdot n^k$ dla każdego ustalonego $k \in N$ zatem możemy napisać nieco ogólniej $n = O(n^k)$ dla każdego $k \in N$. ■

**NOTACJA $\Omega$ (notacja omega duże)**

Niech $f, g : N \to R$. Piszemy $f(n) = \Omega(g(n))$ wtedy i tylko wtedy z definicji, gdy istnieje taka stała $c > 0$ i liczba naturalna $n_0$, że $0 \le cg(n) \le f(n)$ dla każdego $n \ge n_0$.

Dokładniej, formalnie symbol $\Omega(g(n))$ oznacza z definicji zbiór funkcji $f : N \to R$ zdefiniowany następująco

$\Omega(g(n)) = \{ f \in R^N$; istnieje taka stala $c > 0$ i $n_0 \in N$, ze $0 \le cg(n) \le f(n)$ dla kazdego $n > n_0 \}$

Powinniśmy więc pisać $f(n) \in \Omega(g(n))$ zwyczajowo jednak używamy zapisu $f(n) = \Omega(g(n))$.

Sens notacji $\Omega$ to „asymptotyczne ograniczenie dolne", sposób szacowania od dołu wartości funkcji $f$ przez wartości funkcji $g$. Zauważmy jeszcze, że jeśli spełniona jest relacja $f(n) = \Omega(g(n))$ to z definicji notacji „duże $\Omega$" wynika, że funkcje $f, g : N \to R$ są asymptotycznie nieujemne tzn. istnieje taka liczba naturalna $n_0$, że dla każdego $n \ge n_0$ mamy $0 \le f(n)$ oraz $0 \le g(n)$.

**Przykład.** Jeśli $a, b, c \in R$ oraz $a > 0$ to $an^2 + bn + c = \Omega(n^2)$

■

**NOTACJA $\Theta$ (notacja teta duże)**

Niech $f, g : N \to R$, Piszemy $f(n) = \Theta(g(n))$ wtedy i tylko wtedy z definicji, gdy istnieją takie stałe $c_1 > 0$ i $c_2 > 0$ i liczba naturalna $n_0$ taka, że $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ dla każdego $n \ge n_0$.

Dokładniej, formalnie symbol $\Theta(g(n))$ oznacza z definicji zbiór funkcji $f : N \to R$ zdefiniowany następująco

$\Theta(g(n)) = \{ f \in R^N$; istnieje taka stala $c_1 > 0$ i stala $c_1 >$ i $n_0 \in N$, ze $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ dla kazdego $n > n_0 \}$

Powinniśmy więc pisać $f(n) \in \Theta(g(n))$ zwyczajowo jednak używamy zapisu $f(n) = \Theta(g(n))$. Zauważmy jeszcze, że jeśli spełniona jest relacja $f(n) = \Theta(g(n))$ to z definicji notacji „duże $\Theta$" wynika, że funkcje $f, g : N \to R$ są asymptotycznie nieujemne tzn. istnieje taka liczba naturalna $n_0$, że dla każdego $n \ge n_0$ mamy $0 \le f(n)$ oraz $0 \le g(n)$.

**Przykład** $3n^3 + 90n^2 - 5n + 646 = \Theta(n^3)$. Ogólnie mamy dla wielomianu

$$a_k n^k + a_{k-1} n^{k-1} + ... a_1 n + a_0 = \Theta\left(n^k\right)$$

**Fakt** Niech funkcje $f, g : N \to R$ będą asymptotycznie nieujemne oraz $g(n) \neq 0$ dla każdego $n \in N$. Jeśli $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ i $c > 0$ to $f(n) = \Theta\left(g(n)\right)$.

**Dowód.**1.Jeśli $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ i $c > 0$ to z definicji granicy dla każdego $\varepsilon > 0$ istnieje takie $n_0 \in N$, że dla każdego $n > n_0$ mamy $c - \varepsilon \leq \frac{f(n)}{g(n)} \leq c + \varepsilon$ oraz $g(n) > 0$. Mnożąc obie strony tej nierówności przez g(*n*) dostajemy.

$$(c - \varepsilon) g(n) \leq f(n) \leq (c + \varepsilon) g(n)$$

Ponieważ $\varepsilon > 0$ może być dowolnie małe np. możemy przyjąć $\varepsilon = c/2$ i wówczas mamy:

$$\frac{c}{2} g(n) \leq f(n) \leq \frac{3c}{2} g(n)$$

a więc rzeczywiście $f(n) = \Theta\left(g(n)\right)$ ■

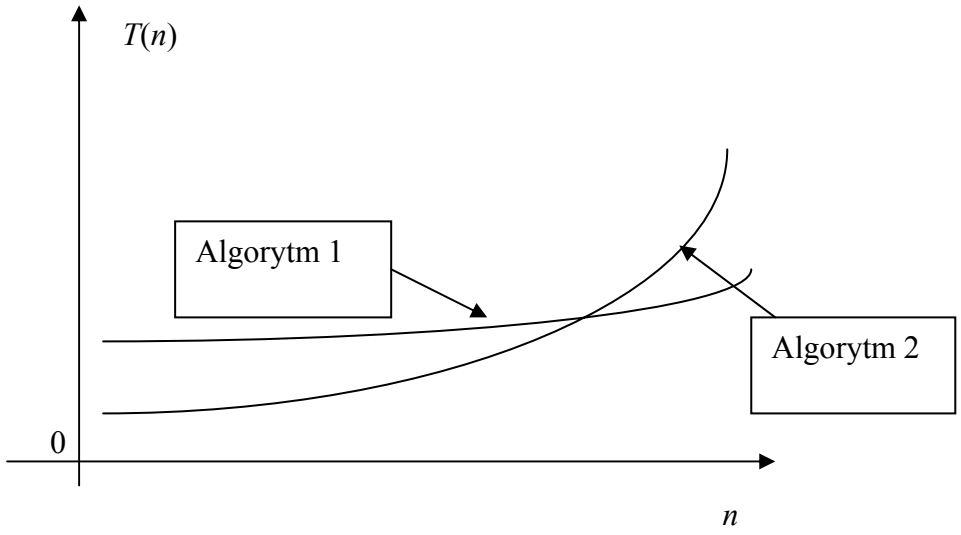
**Fakt** Niech funkcje $f, g : N \to R$ będą asymptotycznie nieujemne oraz dla każdego $n \in N$ mamy $g(n) \neq 0$ oraz $f(n) \neq 0$ . Jeśli $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ i $c > 0$ to $f(n) = \Theta\left(g(n)\right)$ i $g(n) = \Theta\left(f(n)\right)$.

**Dowód** 1. Dowód faktu, że $f(n) = \Theta\left(g(n)\right)$ jest identyczny jak faktu poprzedniego.

2. Ponieważ jeśli $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ i $c > 0$ to przy przyjętych założeniach $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{1}{c}$ oraz $\frac{1}{c} > 0$ rozumując analogicznie jak w p. 1. dostajemy, że $g(n) = \Theta\left(f(n)\right)$.■

**Uwaga.** Z tego, że $g(n) = \Theta\left(f(n)\right)$ i $f(n) = \Theta\left(g(n)\right)$ nie wynika, że istnieje granica $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$. Istotnie niech na przykład $f(n) = 4$ dla *n* parzystych i $f(n) = 1$ dla *n* nieparzystych oraz $g(n) = 2$ dla *n* parzystych i $g(n) = 3$ dla *n* nieparzystych. Oczywiście mamy $g(n) = \Theta\left(f(n)\right)$ oraz $f(n) = \Theta\left(g(n)\right)$ ale $\frac{f(n)}{g(n)} = 2$ dla *n* parzystych oraz $\frac{f(n)}{g(n)} = 1/3$ dla *n* nieparzystych a więc ciąg $(a_n)_{n=1}^{\infty}$, gdzie $a_n = \frac{f(n)}{g(n)}$ nie ma granicy .

**Uwaga.** Warto zwrócić uwagę, że jeśli czasowa złożoność obliczeniowa jakiegoś algorytmu 1 wynosi $\Theta(n^2)$ a algorytmu 2 $\Theta(n^3)$ to asymptotycznie (tzn. dla $n \geq n_0$ dla pewnego $n_0 \in N$) algorytm 1 jest lepszy niż algorytm 2. Jednak dla małych *n* algorytm 2 może okazać się lepszy. Sytuację dobrze ilustruje rys. 1. 1.




Rys.1.1 Dla małych *n* algorytm 2 o większej czasowej złożoności obliczeniowej może okazać się lepszy . Algorytm lepszy asymptotycznie nie zawsze jest lepszy dla małych *n*

**NOTACJA *o* (notacja o małe)**

Niech $g, f : N \to R$. Piszemy $f(n) = o(g(n))$ wtedy i tylko wtedy (z definicji) gdy dla każdej stałej $c > 0$, istnieje taka liczba naturalna $n_0 > 0$, że $0 \leq f(n) < cg(n)$ dla każdego $n \geq n_0$.

Dokładniej, formalnie symbol $o(g(n))$ oznacza z definicji zbiór funkcji $f : N \to R$ zdefiniowany następująco

$$o(g(n)) = \{ f \in R^N ; \text{dla kazdej stalej } c > 0 \text{ istnieje } n_0 \in N, \text{ze } 0 \leq f(n) \leq cg(n) \text{ dla kazdego } n > n_0 \}$$

Powinniśmy więc pisać $f(n) \in o(g(n))$ zwyczajowo jednak używamy najczęściej zapisu $f(n) = o(g(n))$.

**Przykład 1.** $2n = o(n^2)$ ale $2n^2 \neq o(n^2)$. ■

**Przykład 2.** Dla każdego $a_k, a_{k-1}, ..., a_1, a_0 \in R$ oraz ustalonego $k \in N$ mamy

$a_k n^k + a_{k-1} n^{k-1} + ... + a_1 n + a_0 = o(n^{k+1})$

■

**Fakt** Jeśli dla każdego $n \in N$ mamy $g(n) \neq 0$ oraz $f : N \to R$ i $g : N \to R$ są asymptotycznie dodatnie to

$$f(n) = o(g(n)) \quad \text{wtedy i tylko wtedy gdy} \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 .$$

**Dowód.**1.Jeśli $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ to z definicji granicy i asymptotycznej dodatniości mamy, że dla każdego $\varepsilon > 0$ istnieje takie $n_0 \in N$, że dla każdego $n > n_0$ mamy $0 < \frac{f(n)}{g(n)} \leq \varepsilon$ oraz $g(n) > 0$. Mnożąc obie strony tej nierówności przez g($n$) dostajemy dla każdego $n > n_0$

$$0 < f(n) \leq \varepsilon \cdot g(n)$$

Ponieważ $\varepsilon > 0$ mogło być dowolne więc $f(n) = o(g(n))$.
2. Implikacja w przeciwnym kierunku wynika bezpośrednio z definicji granicy.■

**NOTACJA $\omega$ (notacja $\omega$ małe)**
Niech $f, g : N \to R$. Piszemy $f(n) = \omega(g(n))$ wtedy i tylko wtedy (z definicji) gdy dla każdej stałej $c > 0$, gdy istnieje liczba naturalna $n_0$ taka, że $0 \leq c \cdot g(n) < f(n)$ dla każdego $n \geq n_0$.

Dokładniej, formalnie symbol $\omega(g(n))$ oznacza z definicji zbiór funkcji $f : N \to R$ zdefiniowany następująco

$$\omega(g(n)) = \{ f \in R^N ; \text{dla każdej stałej } c > 0 \text{ istnieje takie } n_0 \in N, \text{ze } 0 \leq cg(n) \leq f(n) \text{ dla kazdego } n > n_0 \}$$

Powinniśmy więc pisać $f(n) \in \omega(g(n))$ zwyczajowo jednak używamy zapisu $f(n) = \omega(g(n))$.

**Fakt** Jeśli dla każdego $n \in N$ mamy $g(n) \neq 0$ oraz funkcja $g : N \to R$ jest asymptotycznie dodatnia to

$$f(n) = \omega(g(n)) \quad \text{wtedy i tylko wtedy gdy} \quad \lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty .$$

**Dowód.** 1. Jeśli $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ to z definicji granicy i asymptotycznej dodatniości mamy, że dla każdego $c > 0$ istnieje takie $n_0 \in N$, że dla każdego $n > n_0$ mamy $0 < c < \frac{f(n)}{g(n)}$ oraz $g(n) > 0$. Mnożąc obie strony tej nierówności przez g($n$) dostajemy, że dla każdego $n > n_0$

$$0 < c \cdot g(n) \le f(n)$$

Ponieważ $c > 0$ mogło być dowolne więc $f(n) = \omega(g(n))$.
2. Implikacja w przeciwnym kierunku. Jeśli $f(n) = \omega(g(n))$ to zgodnie z definicją notacji „małe $\omega$" dla każdego $c > 0$ istnieje takie $n_0' \in N$, że dla każdego $n > n_0'$ mamy

$$0 \le c \cdot g(n) \le f(n)$$

Ponieważ z założenia funkcja $g : N \to R$ jest asymptotycznie dodatnia to istnieje takie $n_0'' \in N$, że dla każdego $n > n_0''$ mamy $g(n) > 0$.

Zatem dla każdego $c > 0$ istnieje takie $n_0 = \max(n_0'', n_0')$, że $0 \le c \cdot g(n) \le f(n)$ i $g(n) > 0$. Dzieląc teraz obie strony pierwszej nierówności przez $g(n)$ dostajemy $0 \le c < \frac{f(n)}{g(n)}$ stwierdzamy więc, że dla każdego $c > 0$ istnieje takie $n_0 \in N$, że dla każdego $n > n_0$ mamy $0 \le c < \frac{f(n)}{g(n)}$ a więc $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$

■

**Fakt (własności notacji asymptotycznych, własności rzędu funkcji)**

Zakładamy, że $f, g, h, l : N \to R$

1. $f(n) = O(g(n))$ wtedy i tylko wtedy, gdy $g(n) = \Omega(f(n))$
2. $f(n) = \Theta(g(n))$ wtedy i tylko wtedy, gdy $f(n) = O(g(n))$ i $f(n) = \Omega(g(n))$
3. Jeśli $f(n) = O(h(n))$ oraz $g(n) = O(h(n))$ to $(f + g)(n) = 0(h(n))$
4. Jeśli $f(n) = O(h(n))$ oraz $g(n) = O(l(n))$ to $f(n) \cdot g(n) = O(h(n) \cdot l(n))$
5. Zwrotność $f(n) = O(f(n))$
6. Przechodniość: jeśli $f(n) = O(g(n))$ oraz $g(n) = O(h(n))$ to $f(n) = O(h(n))$

**Fakt (oszacowania za pomocą zapisów asymptotycznych dla typowych funkcji)**

1.Jeśli funkcją jest funkcja wielomianową, $f(n)$ jest wielomianem stopnia $k$ z wyrazem dodatnim przy najwyższej potędze, to $f(n) = \Theta(n^k)$

2.Dla dowolnego $c > 0$ mamy $\log_c n = \Theta(\ln n)$

3. $\lg(n!) = \Theta(n \lg n)$

4.Ze wzoru Stirlinga dla $n!$ : dla każdego $n \geq 0$ mamy

$$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leq n! \leq \sqrt{2\pi n}\left(\frac{n}{e}\right)^{(n+\frac{1}{12\cdot n})}$$

Wynika stąd, że

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

$$n! = 0\left(n^n\right)$$

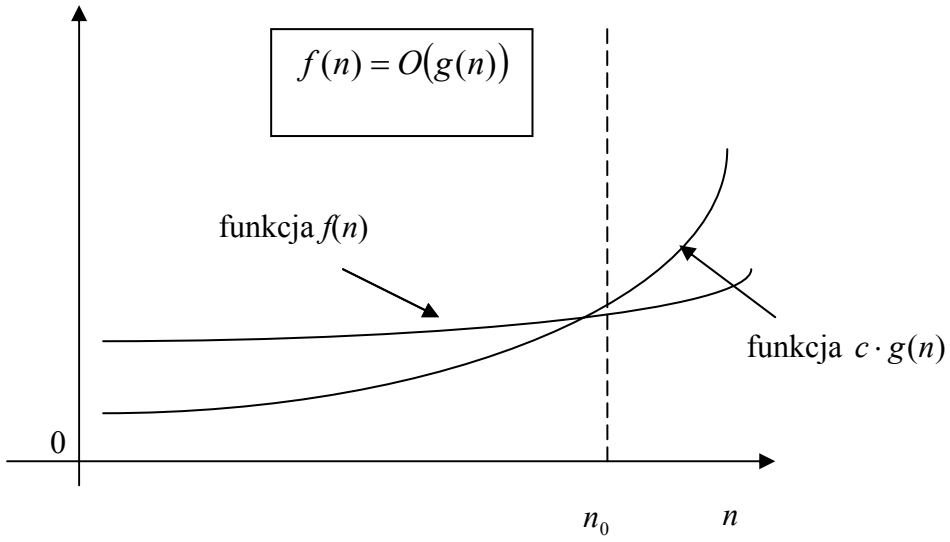
$$n! = \Omega\left(2^n\right)$$

**Przykład.** Lista funkcji o rosnącej szybkości wzrostu. Niech $c$ i $\varepsilon$ będą takimi stałymi, że $0 < \varepsilon < 1 < c$.

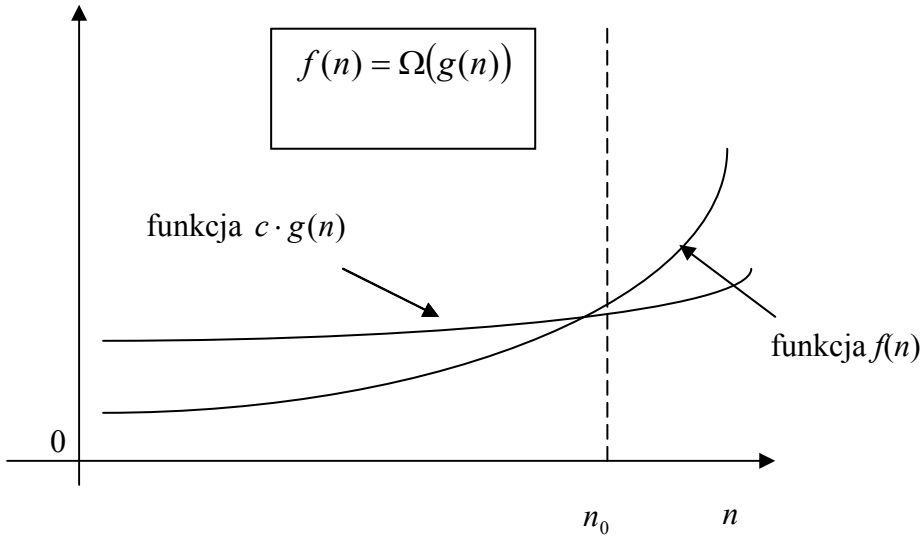
$$1 < \ln\ln n < \ln n < \exp\left(\sqrt{\ln n \ln\ln n}\right) < n^{\varepsilon} < n^{c} < n^{\ln n} < c^{n} < n^{n} < c^{c^{n}}$$

**Uwaga.** Często analizując algorytmy mówimy o kategoriach złożoności (ang. *complexity categories*). Najczęściej w praktyce mamy do czynienia z następującymi kategoriami złożoności

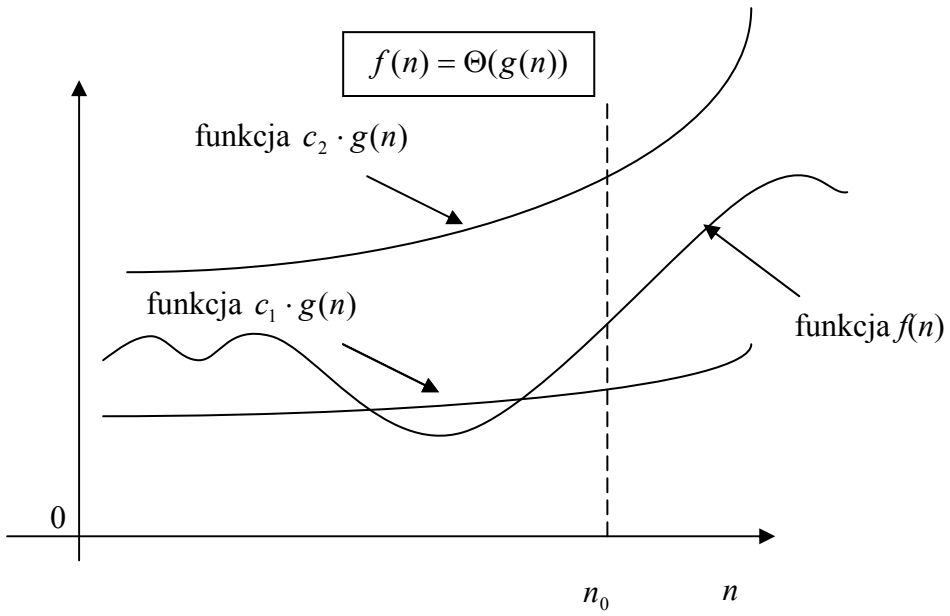
$$\Theta(\ln n) \qquad \Theta(n) \qquad \Theta(n \ln n) \qquad \Theta(n^2) \qquad \Theta(n^3) \qquad \Theta(2^n)$$




Rys 1. Ilustracja oszacowania asymptotycznego „duże O” $f(n) = O(g(n))$.




Rys 2. Ilustracja oszacowania asymptotycznego „duża $\Omega$” $f(n) = \Omega(g(n))$.




Rys 3. Ilustracja oszacowania asymptotycznego „duże $\Theta$” $f(n) = \Theta\big(g(n)\big)$.

# 1.5 Elementarne struktury danych

Struktury danych mówiąc najprościej to obiekty, na których operuje algorytm. Istotne są przy tym związane ze strukturą danych typowe realizowane na niej operacje oraz sposób implementacji struktury danych w komputerze.

Charakterystyczne dla implementacji struktur danych jest wykorzystywanie do definiowania wprowadzanej struktury danych struktur prostszych. Z reguły każdy język programowania ma zaimplementowany pewien prosty podstawowy zestaw struktur danych, który może służyć do definiowania struktur danych bardziej złożonych.

Struktury danych możemy podzielić na statyczne których rozmiar (czyli zapotrzebowanie na pamięć) nie ulega zmianie w czasie i dynamiczne, które zmieniają (lub mogą zmieniać) swój rozmiar w czasie wykonania algorytmu. Przykładem statycznej struktury danych jest tablica lub rekord, dynamicznej mawiana w dalszym ciągu lista.

Struktury danych omawiane w tym podrozdziale to dynamiczne struktury danych, których rozmiar może ulegać zmianie podczas realizacji algorytmu.

Warto może w tym miejscu przypomnieć określenie Wirtha: program to algorytm + struktura danych

**1. Lista.** Lista (lub lista uporządkowana) jest jedną z podstawowych, najczęściej używanych struktur danych. Z matematycznego punktu widzenia lista to skończony ciąg $a_1, a_2, ..., a_n$ elementów pewnego zbioru $U$ (często ten zbiór nazywamy uniwersum). Lista tym jednak różni się od ciągu, że z listą wiążemy pewien zestaw typowych operacji na listach.

Listę będziemy oznaczać tak $q = [a_1, a_2, ..., a_n]$, gdzie $q$ jest nazwą listy. Liczbę $n \in N$, ($|q| = n$) nazywamy, długością lub rozmiarem listy. Elementy $a_1$ i $a_n$ nazywamy końcami listy $q = [a_1, a_2, ..., a_n]$ ($a_1$ jest lewym końcem listy a $a_n$ prawym końcem listy $q = [a_1, a_2, ..., a_n]$). Szczególnym przypadkiem listy jest lista pusta oznaczana symbolem $q = [\ ]$.

Niech $q = [a_1, a_2, ..., a_n]$ i $r = [b_1, b_2, ..., b_m]$ będą listami oraz $i, j \in <1, n>$. Podstawowymi operacjami na listach są:

1. dostęp do dowolnego elementu listy: $q[i] = b_i$

2. branie podlisty: $q[i..j] = [a_i, a_{i+1}, ..., a_j]$

3. złożenie czyli konkatenacja list $q \,\&\, r = [a_1, a_2, ..., a_n, b_1, b_2, ..., b_m]$

4.wstawianie elementu $x$ na pozycję $i+1$ (czyli za element $a_i$) w liście $q=[a_1,a_2,...,a_n]$. Sprowadza się to do utworzenia listy $q[1..i]\&[x]\&q[(i+1)..|q|]$. Oczywiście taka lista ma długość $|q|+1$.

5. usunięcie $i$-tego elementu z listy $q=[a_1,a_2,...,a_n]$ (gdzie $i\in<1,n>$) to utworzenie listy $r=[a_1,a_2,...,a_{i-1},a_{i+1},...,a_n]=q[1..(i-1)]\&q[(i+1)..n]$. Oczywiście taka lista ma długość $|q|-1$.

Jak widać operacje 4 i 5 można realizować za pomocą 3 pierwszych operacji.

Listy wygodnie jest używać w specjalny sposób operując tylko na jej końcach. Definiujemy 6 specjalnych operacji działających tylko na końcach listy: lewym albo prawym .

3 operacje dotyczące lewego końca listy

1.pobieranie (dostęp do) lewego końca listy $q=[a_1,a_2,...,a_n]$(nie modyfikujemy listy)

$front(q)\stackrel{df}{=}q[1]$

2. wstawianie elementu $x$ na lewy koniec listy $q=[a_1,a_2,...,a_n]$: $push(q,x)\stackrel{df}{=}[x]\&q$

3.usunięcie lewego końca listy z listy $q=[a_1,a_2,...,a_n]$: $pop(q)\stackrel{df}{=}q[2..|q|]$

3 operacje dotyczące prawego końca listy

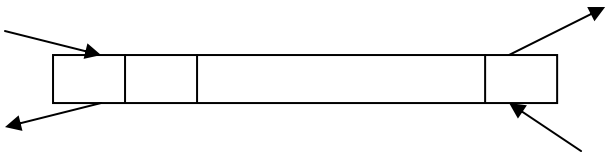
1.pobieranie (dostęp do) prawego końca listy $q=[a_1,a_2,...,a_n]$(nie modyfikujemy listy)

$rear(q)\stackrel{df}{=}q[|q|]$

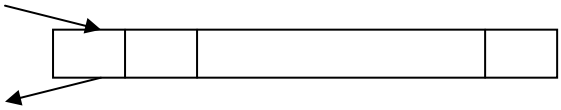
2. wstawianie elementu $x$ na prawy koniec listy $q=[a_1,a_2,...,a_n]$: $inject(q,x)\stackrel{df}{=}q\&[x]$

3.usunięcie prawego końca listy z listy $q=[a_1,a_2,...,a_n]$: $eject(q)\stackrel{df}{=}q[1..(|q|-1)]$

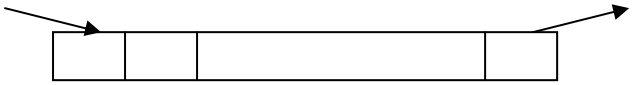
Listę na której można wykonywać 6 powyższych operacji nazywamy kolejką podwójną lub kolejką dwustronną.



Rys. 1.5 Kolejka podwójna



Rys. 1.6 Stos



Rys. 1.7 Kolejka

Jeśli na liście możemy wykonywać tylko 3 pierwsze operacje $front(q) \overset{df}{=} q[1]$ $push(q,x) \overset{df}{=} [x] \,\&\, q$, $pop(q) \overset{df}{=} q[2..|q|]$ to taką listę nazywamy stosem.

Jeśli na liście $q = [a_1, a_2, ..., a_n]$ możemy wykonywać tylko operacje $front(q) \overset{df}{=} q[1]$, $pop(q) \overset{df}{=} q[2..|q|]$ i $eject(q) \overset{df}{=} q[1..(|q|-1)]$ to taką listę nazywamy kolejką.

Istnieją 2 typowe sposoby implementacji listy $q = [a_1, a_2, ..., a_n]$:

- tablicowa

-dowiązaniowa

Ponieważ struktura dowiązaniową nie powinna być pusta dodajemy na początku listy element pusty nazywany głową lub wartownikiem listy. Każdy węzeł struktury dowiązaniowej wskazuje na element następny listy (jeśli taki istnieje).

**2. Zbiór.** Struktura danych o nazwie zbiór to odpowiednik pojęcia zbioru skończonego w sensie teorii mnogości. Liczbę elementów w zbiorze $S$ oznaczamy symbolem $|S|$ i nazywamy rozmiarem zbioru $S$.

Ze zbiorem $S$ jako strukturą danych wiążemy kilka podstawowych operacji

1. wstawianie elementu $x$ do zbioru $S$ $insert(x,S)$ ; $S := S \cup \{x\}$

2 usuwanie elementu $x$ ze zbioru $S$ $delete(x,S)$ ; $S := S - \{x\}$

3. sprawdzanie czy element $x$ jest elementem zbioru $S$ ; $member(x,S)$

4. znalezienie najmniejszego elementu w zbiorze $S$ ; $\min(S)$ (zakładamy, że S jest wyposażony w pewien liniowy porządek $\leq$)

5. usuwanie najmniejszego elementu zbioru $S$ ; $delete\_\min(S)$ ; $S := S - \{\min(S)\}$

6. Obliczanie sumy i iloczynu zbiorów.

Implementacje zbioru mogą być różne. Stosowane najczęściej to:

- implementacja za pomocą funkcji charakterystycznej zbioru. Zbiór uniwersum $U$ utożsamiamy ze zbiorem indeksów tablicy i zbiór S utożsamiamy z tablicą

$$C[x] = \begin{cases} 1 \text{ jesli } x \in S \\ 0 \text{ jesli } x \notin S \end{cases}$$

- implementacja listowa

**3. Kolejka priorytetowa.** Kolejka priorytetowa to jedna z ważniejszych struktur danych. Mówiąc najprościej kolejka priorytetowa to multizbiór wyposażony w pewne specjalne operacje. Multizbiór to zbiór, w którym elementy mogą się powtarzać.

Kolejka priorytetowa tym różni się od listy, że w kolejce priorytetowej (tak jak w przypadku zbioru) elementy nie są ustawione w ciąg.

Operacje definiowane dla kolejki priorytetowej to:

1. $construct(q,S)$ - utworzenie multizbioru $S = \{a_1, a_2, ..., a_n\}$ z danej listy $q = [a_1, a_2, ..., a_n]$

2. $insert(x,S)$ - dodawanie elementu $x$ do multizbioru $S=\{a_1,a_2,...,a_n\}$ czyli $S:=S\cup\{x\}$

3. $delete\_\max(q,S)$ - usunięcie z multizbioru $S=\{a_1,a_2,...,a_n\}$ największego elementu

4. $delete\_\min(q,S)$ - usunięcie z multizbioru $S=\{a_1,a_2,...,a_n\}$ najmniejszego elementu

Kolejkę priorytetową można zaimplementować za pomocą

- Listy nieuporządkowanej

- Listy uporządkowanej (czyli listy). Z kolei listę można reprezentować np. za pomocą metody dowiązaniowej.

- Kopca (ang heap) por. rozdział o algorytmach sortowania

**4. Graf.** Z matematycznego punktu widzenia graf to uporządkowana para zbiorów $G=(V,E)$, gdzie $V$ jest dowolnym zbiorem skończonym, którego elementy nazywamy wierzchołkami lub węzłami grafu. Jeśli zbiór $E$ jest zbiorem uporządkowanych par wierzchołków to taki graf nazywamy skierowanym. Jeśli zbiór $E$ jest podzbiorem zbioru wszystkich dwuelementowych podzbiorów zbioru $V$ to taki graf nazywamy niekierowanym.
Elementy zbioru $E$ nazywamy krawędziami.
Przez rozmiar grafu rozumiemy sumę dwu liczb $n=|V|$ i $m=|E|$

Graf implementujemy z reguły za pomocą

- list tzw. list sąsiedztwa

Dla każdego wierzchołka $x\in V$ definiujemy listę $L[x]$ wierzchołków połączonych z wierzchołkiem $x\in V$.

- Macierzy sąsiedztwa $A$ (inaczej macierzy incydencji) definiowanej następująco:

$$A[x,y]=\begin{cases}1 \text{ dla } (x,y)\in E\\ 0 \text{ dla } (x,y)\notin E\end{cases}$$

**Drzewo** to dowolny niezorientowany graf spójny i acykliczny. Graf spójny to graf taki, ż każde 2 jego wierzchołki są połączone ścieżką utworzoną z krawędzi grafu. Acykliczność grafu oznacza, że w grafie nie ma cykli czyli zamkniętych ścieżek.

**Drzewo z korzeniem** to drzewo z wyróżnionym jednym wierzchołkiem. Mówiąc drzewo mamy z reguły na myśli drzewo z korzeniem.

Szczególnym przypadkiem drzewa z korzeniem jest **drzewo binarne,** które charakteryzuje się tym, że każdy wierzchołek drzewa ma co najwyżej dwóch potomków

Użytecznych ważnych struktur danych jest dosyć dużo. Inne ważne struktury danych to m.in. drzewa czarno-białe, B-drzewa, zrównoważone drzewa poszukiwań, kopce, kopce Fibonaciego i kopce dwumianowe.

# 1.6 Rekurencja i metody projektowania algorytmów

Rekurencja polega na tym, że rozwiązanie danego problemu (z rozmiarem danych wejściowych $n$) wyraża się za pomocą rozwiązań tego samego problemu dla danych o mniejszych rozmiarach. Rekurencja nazywana jest czasami rekursją.

W matematyce mechanizm rekurencyjny stosowany jest dosyć często do definiowania lub opisywania algorytmów. Użycie opisu rekurencyjnego w przypadku algorytmu pozwala z reguły na przejrzysty zwarty opis funkcji lub procedury.

Klasycznym przykładem (problemem) podawanym przy okazji omawiania pojęcia rekurencji są tzw. wieże Hanoi. Wieże Hanoi bardzo dobrze ilustrują skuteczność i prostotę rozumowania rekurencyjnego.

**Przykład 1.** (wieże Hanoi)
Mamy 3 pręty 1,2,3 (por. rys. 1.1). Na pręt 1 nałożonych jest $n$ krążków o malejącej średnicy. Zadanie polega na przeniesieniu krążków z pręta 1 na jeden prętów 2 lub 3 tak by były końcu ułożone tak jak na pręcie 1 (tzn. wg. zasady mniejszy na większym) . Wolno w każdym elementarnym ruchu przekładać górny krążek na dowolny pręt jeśli jest pusty lub z zachowaniem zasady, że krążek mniejszy można położyć na większym ale nigdy odwrotnie. Pokażemy stosując rekurencję , że zadanie ma rozwiązanie dla dowolnego $n \in N$ . Widać, że dla $n = 1,2,3$ zadanie jest trywialne i ma rozwiązanie.

Załóżmy, że problem ma rozwiązanie dla $n$-1 krążków wówczas ma rozwiązanie dla $n$ krążków. Istotnie korzystając z metody rozwiązania dla dla $n$-1 krążków przerzucimy n-1 krążków na pręt nr 2 a następnie przerzucimy krążek największy na pręt nr. 3 i ponownie skorzystamy z metody dla $n$-1 krążków przestawiając $n$-1 krążków na pręt nr 3.

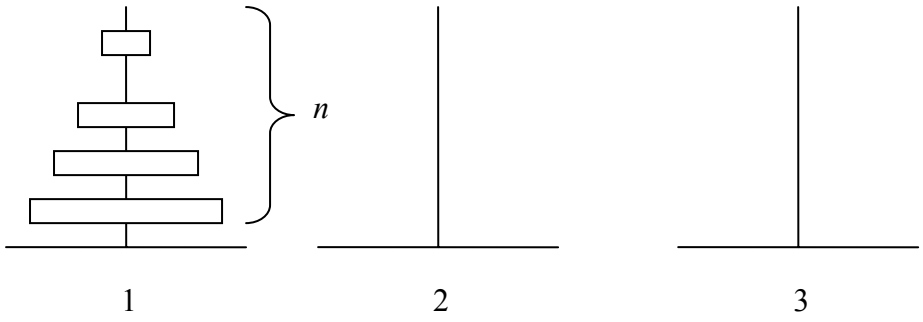
Łatwo można pokazać, że liczba $T(n)$ niezbędnych elementarnych kroków (przełożeń krążków) wyrażona jest za pomocą tzw. równania rekurencyjnego

$$T(0) = 0$$

$$T(n) = 2T(n-1) + 1 \quad \text{dla} \quad n > 0$$

Rozwiązaniem tego równania jest $T(n) = 2^n - 1$. Uważny Czytelnik łatwo sprawdzi, że jeśli $n = 64$ i na jedno przełożenie krążka poświęcimy $1\,\mu s$ to na realizację całego algorytmu przełożenia wieży poświęcimy ponad pół miliona lat.

Wniosek płynący z powyższego przykładu jest więc taki, że rekurencja wspaniale upraszcza rozumowanie i opis algorytmu rozwiązania problemu ale bywa kłopotliwa w realizacji. ■




Rys. 1.1 Wieże Hanoi

**Przykład 2.** (Obliczanie silni $n!$ dla danego $n$.).
Dobrym przykładem ilustrującym użycie rekurencji w definicji jest definicja silni. Można silnię *n!* definiować i obliczać metodą rekurencyjną i iteracyjną. Pokażemy obie metody. Definicja rekurencyjna silni jest następująca.

$$n! = \begin{cases} 1 & \text{dla } n = 0 \\ n \cdot (n-1)! & \text{dla } n > 0 \end{cases}$$

Odwołujemy się więc w definicji $n!$ do definicji $(n-1)!$. W definicji $(n-1)!$ do definicji $(n-2)!$ itd. Algorytm obliczający silnię rekurencyjnie jest więc następujący.

---

**Algorytm:** Obliczanie wartości silni $n!$ metodą rekurencyjną

Dane WEJŚCIOWE: $n \in N$

Dane WYJŚCIOWE: wartość silni $n!$

---

**function** *silnia*(*n*: **integer**): **integer**;

**begin**
**if** $n = 0$ **then** $silnia := 1$ **else** $silnia := n * silnia(n-1)$
**end** *silnia*;

Powyższy program można zapisać w Matlabie następująco:

---

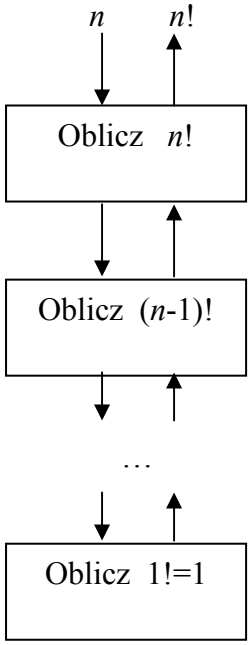
**Algorytm:** Obliczanie wartości silni $n!$ metodą rekurencyjną

Dane WEJŚCIOWE: $n \in N$

Dane WYJŚCIOWE: wartość silni $n!$

---

```
function wart_1=silnia(n)
if n==0 wart_1=1, else wart_1=n*silnia(n-1), end
```




Rys. 1.2 Sposób obliczania silni $n!$ metodą rekurencyjną

---

**Algorytm:** Obliczanie wartości silni $n!$ metodą iteracyjną

Dane WEJŚCIOWE: $n \in N$

Dane WYJŚCIOWE: wartość silni $n!$

---

**function** *silnia*(*n*: **integer**): **integer**;

**begin**
$k := 1;$
**for** $i := 1$ **to** $n$ **do** $k := k * i;$
$silnia := k$
**end** *silnia*;

Powyższy algorytm można zapisać w Matlabie następująco

---

**Algorytm:** Obliczanie wartości silni $n!$ metodą iteracyjną

Dane WEJŚCIOWE: $n \in N$

Dane WYJŚCIOWE: wartość silni $n!$

---

```
function wart_1=silnia_iter(n)
k=1
for i=1:n, k=k*i, end
wart_1=k
```

Rozwiązanie iteracyjne okazuje się w przypadku obliczania silni znacznie szybsze.

■

W przypadku problemów, które można rozwiązywać iteracyjnie lub rekurencyjnie decyzja o wyborze algorytmu (por. definicja silni) powinna być starannie przemyślana. Nie zawsze bowiem rozwiązanie rekurencyjne prowadzi do rozwiązania efektywnego czasami wręcz użycie rekurencji prowadzi wyraźnie do obniżenia efektywności algorytmu.

Praktycznie we wszystkich współczesnych językach programowania dopuszczalne są procedury i funkcje definiowane rekurencyjnie. Procedurę, która bezpośrednio lub pośrednio wywołuje sama siebie nazywamy procedurą rekurencyjną. W treści procedury zdefiniowanej rekurencyjnie występuje odwołanie do tej samej procedury tzn. do procedury definiowanej.

**Przykład 3** (obliczanie *n*-tego wyrazu ciągu Fibonacciego)
Ciąg Fibonacciego jest określony następującym wzorem $F_0 = 0$, $F_1 = 1$ oraz

$$F_n = F_{n-1} + F_{n-2} \text{ dla każdego } n \geq 2$$

Podamy algorytm obliczający funkcję *Fib*(*n*) (której wartością jest *n*-ty wyraz ciągu Fibonacciego)

a) w sposób rekurencyjny

b) w sposób nierekurencyjny (iteracyjny)

---

**Algorytm:** Rekurencyjna wersja algorytmu obliczania *n*-tego wyrazu ciągu Fibonacciego

Dane WEJŚCIOWE: $n \in N \cup \{0\}$
Dane WYJŚCIOWE: $F_n$ *n*-ty wyraz ciągu Fibonacciego

---

**function** *fibonacci*(*n* : **integer**): **integer**

**begin**
**if** $n \leq 1$ **then** *fibonacci* : = *n* **else** *fibonacci* : = *fibonacci*(*n*-1) + *fibonacci*(*n*-2)

**end** *fibonacci*;

Powyższy algorytm można zapisać w Matlabie następująco

---

**Algorytm:** Rekurencyjna wersja algorytmu obliczania *n*-tego wyrazu ciągu Fibonacciego

Dane WEJŚCIOWE: $n \in N \cup \{0\}$

Dane WYJŚCIOWE: $F_n$ *n*-ty wyraz ciągu Fibonacciego

---

```
function wart_1=fibonacci_rekur(n)
if n<=1, wart_1=n, else wart_1=fibonacci_rekur(n-1) + fibonacci_rekur(n-2), end
```

---

**Algorytm:** Iteracyjna wersja algorytmu obliczania *n*-tego wyrazu ciągu Fibonacciego

Dane WEJŚCIOWE: $n \in N \cup \{0\}$

Dane WYJŚCIOWE: $F_n$ *n*-ty wyraz ciągu Fibonacciego

---

```
function fibonacci(n : integer): integer

begin
var fib1, fib2, fib3, i : integer;
fib1 : =0 ; fib2 : =1 ;

if   n ≤ 1 then fibonacci : = n else  begin
                                      for  i : = 2  to n do begin
                                                            fib3 : = fib1+ fib2;
                                                            fib1 : = fib2 ;
                                                            fib2 : = fib3 ;
                                                            end ;
                                       fibonacci : =fib3

                                      end;
end fibonacci;
```

W Matlabie powyższy algorytm można zapisać następująco:

---

**Algorytm:** Iteracyjna wersja algorytmu obliczania *n*-tego wyrazu ciągu Fibonacciego

Dane WEJŚCIOWE: $n \in N \cup \{0\}$

Dane WYJŚCIOWE: $F_n$ *n*-ty wyraz ciągu Fibonacciego

---

```
function wart_1=fibonacci_iter(n)
fib1=0; fib2=1;
if n<=1, wart_1=n, else
   for i=2:n, fib3=fib1+fib2; fib1=fib2; fib2=fib3; end
   wart_1=fib3
end
```

Zauważmy, że w przypadku opisanego wyżej algorytmu rekurencyjnego wielokrotnie odwołujemy się rekurencyjnie do tej samej procedury z tymi samymi parametrami (a więc obliczamy to samo) co daje w wyniku algorytm o niskiej efektywności w porównaniu z algorytmem iteracyjnym.

■

**Przykład 3** (obliczanie wartości $\binom{n}{k}$ metodą rekurencyjną)

Przeanalizujemy jeszcze jeden algorytm rekurencyjny obliczający wartość $\binom{n}{k} \overset{df}{=} \frac{n!}{k!\cdot(n-k)!}$ gdzie $n,k \in N$, $k \le n$. Bezpośrednie obliczenie $\binom{n}{k}$ z definicji może być kłopotliwe. Łatwo możemy jednak obliczać $\binom{n}{k}$ za pomocą wzoru rekurencyjnego $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$. Z tego wzoru korzystamy też tworząc tzw. trójkąt Pascala. Odpowiedni algorytm obliczania wartości $\binom{n}{k}$ jest następujący.

---

**Algorytm:** Obliczanie wartości współczynnika $\binom{n}{k}$ metodą rekurencyjną

Dane WEJŚCIOWE: $n \in N$ , $k \in <0,n>$

Dane WYJŚCIOWE: wartość współczynnika $\binom{n}{k}$

---

**function** *n_po_k*(*n*,*k* : **integer**): **integer**

**begin**

**if** (*k*=0 or *k*=*n*) **then** *n_po_k*: = 1 **else** *n_po_k* : = *n_po_k*(*n*-1,*k*-1) + *n_po_k*(*n*-1,*k*)

**end** *n_po_k*;

W Matlabie powyższy algorytm można zapisać następująco:

---

**Algorytm:** Obliczanie wartości współczynnika $\binom{n}{k}$ metodą rekurencyjną

Dane WEJŚCIOWE: $n \in N$ , $k \in <0,n>$

Dane WYJŚCIOWE: wartość współczynnika $\binom{n}{k}$

---

**function** wart_1=n_po_k_rekur(n,k)
**if** (k==0|k==n), wart_1=1; **else** wart_1=n_po_k_rekur(n-1,k-1) + n_po_k_rekur(n-1,k), **end**

Zauważmy, że w powyższym algorytmie liczba $T(n,k)$ wykonań procedury *n_po_k* jest równa

$$T(n,k) = 2\binom{n}{k} - 1 \qquad (*)$$

Dowód tego faktu można przeprowadzić przez indukcję względem $n$. Dla $n = 1$ i każdego $k \in <0,n>$ jak łatwo sprawdzić wzór jest oczywiście prawdziwy. Jeśli wzór jest prawdziwy dla pewnego $n-1$ oraz każdego $k \in <0,n-1>$ to
1.Jeśli $k = 0$ lub $k = n$ to $T(n,k) = 1$ i wzór (*) jest oczywiście prawdziwy.
2. Jeśli $k \neq 0$ i $k \neq n$ to korzystając z założenia indukcyjnego i faktu, że

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

mamy:

$$T(n,k) = T(n-1,k-1) + T(n-1,k) + 1 = 2\binom{n-1}{k-1} - 1 + 2\binom{n-1}{k} - 1 + 1 = 2\binom{n}{k} - 1$$

Ostatecznie więc stwierdzamy, że wzór (*) jest prawdziwy dla każdego $n \in N$ i $k \in <0,n>$.■

**Przykład 4** (obliczanie wartości $\binom{n}{k}$ metodą programowania dynamicznego)

Przeanalizujemy jeszcze jeden algorytm obliczający wartość $\binom{n}{k} \overset{df}{=} \frac{n!}{k!\cdot(n-k)!}$ gdzie $n,k \in N$, $k \leq n$. W algorytmie tym budujemy fragment trójkąta Pascala wykorzystując wzór $\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$. Odpowiedni algorytm obliczania wartości $\binom{n}{k}$ jest następujący.

---

**Algorytm:** Obliczanie wartości współczynnika $\binom{n}{k}$ metodą programowania dynamicznego

Dane WEJŚCIOWE: $n \in N$ , $k \in < 0, n >$

Dane WYJŚCIOWE: wartość współczynnika $\binom{n}{k}$

---

**function** *n_po_k*(*n*,*k* : **integer**): **integer**

**begin**
**var**
$i, j$ **: integer**
$A[0..n, 0..k]$ **array of integer**

**for** $i := 0$ **to** *n* **do**
**for** $j := 0$ **to** min(*i*,*k*) **do**

**if** (*j*=0 **or** *j*=*i*) **then** $A[i, j] := 1$ **else** $A[i, j] := A[i - 1, j - 1] + A[i - 1, j]$;

*n_po_k* : =*A*[*n*,*k*]

**end** *n_po_k*;

Powyższy algorytm można zapisać też w Matlabie następująco

---

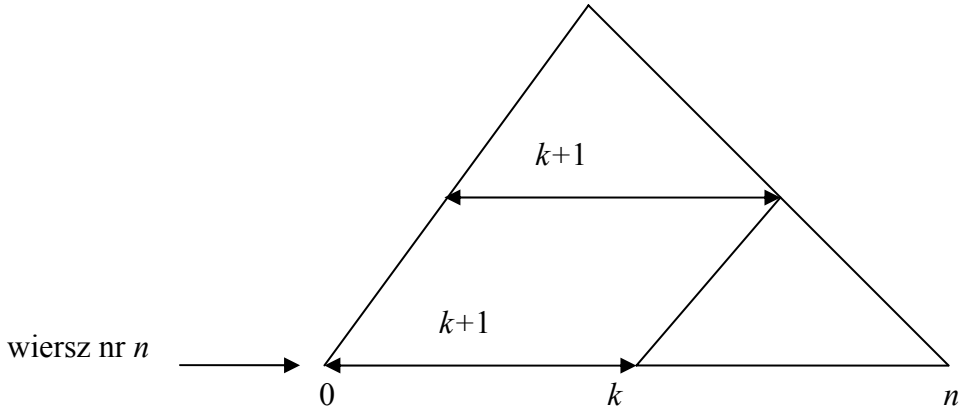
**Algorytm:** Obliczanie wartości współczynnika $\binom{n}{k}$ metodą programowania dynamicznego

Dane WEJŚCIOWE: $n \in N$ , $k \in < 0, n >$

Dane WYJŚCIOWE: wartość współczynnika $\binom{n}{k}$

---

```
function wynik_1=n_po_k(n,k)
for i=1:n+1,
   for j=1:min(i,k+1),
      if (j==1|j==i), A(i,j)=1, else A(i,j)=A(i-1,j-1)+A(i-1,j), end
   end
end
wynik_1=A(n+1,k+1)
```




Rys. 1.2 Algorytm obliczania współczynnika $\binom{n}{k}$ wykorzystujący metodę programowania dynamicznego tworzy fragment trójkąta Pascala

Powyższy algorytm skonstruowany metodą programowania dynamicznego okazuje się znacznie efektywniejszy od algorytmu skonstruowanego metodą dziel i zwyciężaj.
■

Jak zobaczymy w rozdziale o algorytmach sortowania często problem sortowania daje się rozwiązać rekurencyjnie np. algorytm sortowania przez scalanie, to również algorytm rekurencyjny.

Omówimy teraz krótko 2 metody konstrukcji algorytmu metodę dziel i zwyciężaj i metodę programowania dynamicznego.

**Metoda dziel i zwyciężaj (ang. divide-and-conquer)**

Istota metody „dziel i zwyciężaj" polega na tym, że w celu rozwiązania danego problemu dzielimy ten problem na podproblemy, dla których najpierw znajdujemy rozwiązania a następnie za pomocą tych rozwiązań znajdujemy rozwiązanie całego problemu. Jest to bardzo naturalna koncepcja.

Metoda ta zwłaszcza stosowana rekurencyjnie prowadzi często do efektywnych algorytmów rozwiązujących dany problem, których podproblemy maja taką samą postać, ale są mniejszego rozmiaru.

Zastosowanie rekurencji prowadzi do bardziej zrozumiałych i zwięzłych opisów algorytmów, choć często ich czas wykonania jest dłuższy niż odpowiednich algorytmów iteracyjnych**.**

## Metoda programowania dynamicznego

Programowanie dynamiczne (ang. dynamic programming) to technika projektowania algorytmów polegająca na tym ,że problem zostaje podzielonyna mniejsze ale zaczynamy obliczenia od rozwiązywania najprostszych problemów składając z nich rozwiązanie problemu bardziej złożonego itd. Programowanie dynamiczne jest więc metodą wstępującą (ang. bottom-up)

Zaprojektowanie algorytmu zgodne z metodą programowania dynamicznego polega na:

1.Określeniu własności rekurencyjnej opisującej obliczenia

2. Rozwiązaniu problemu zgodnie z metodą wstępującą tzn. bottom-up.

## Algorytmy zachłanne (ang. greedy algorithms)

Algorytmy zachłanne służą do rozwiązywania problemów optymalizacyjnych. Istota rzeczy polega na tym, że algorytm zachłanny dokonuje zawsze wyboru, który lokalnie czyli w danej chwili jest oceniany jako najkorzystniejszy.

Dokonywany lokalnie najkorzystniejszy wybór ma w założeniu prowadzić do znalezienia globalnego optymalnego rozwiązania jednak algorytmy zachłanne nie zawsze prowadzą do optymalnych rozwiązań. Często jednak rozwiązania suboptymalne są całkowicie dla wielu problemów wystarczające.

Typowym problemem dającym się rozwiązać za pomocą algorytmu zachłannego jest tzw. „problem wyboru zajęć".

## Algorytmy aproksymacyjne (ang. aproximation algorithms)

Załóżmy, że rozwiązujemy problem optymalizacyjny. Z każdym dopuszczalnym rozwiązaniem problemu wiążemy pewien koszt *c*. Chcielibyśmy znaleźć rozwiązanie optymalne tzn. (zależnie od problemu) o największym lub największym koszcie.

Co zrobić jeśli problem optymalizacyjny jest obliczeniowo trudny? Możemy postąpić tak:

1. ograniczyć się do małych rozmiarów danych (do małego *n*)

2.nałożyć dodatkowe ograniczenia na problem co niekiedy znakomicie upraszcza algorytm

3. wykorzystać algorytm aproksymacyjny mający mniejszą złożoność obliczeniową niż algorytm znajdujący rozwiązanie optymalne ale dostarczający rozwiązania na ogół gorszego co do kosztu ale z kontrolowanym wskaźnikiem pogorszenia się jakości rozwiązania.

Algorytm aproksymacyjny rozwiązujący problem optymalizacyjny na ogół nie znajduje rozwiązania optymalnego tzn. rozwiązania o koszcie $c_{opt}$ ale suboptymalne o pewnym koszcie $c$. W algorytmach aproksymacyjnych staramy się jednak kontrolować różnicę pomiędzy kosztem rozwiązania optymalnego $c_{opt}$ a kosztem rozwiązania suboptymalnego $c$. Prowadzi to do pojęcia algorytmu $\rho(n)$ aproksymacyjnego.

Definicja algorytmu $\rho(n)$ aproksymacyjnego jest następująca. Jeśli potrafimy znaleźć funkcję $\rho : N \to R^+$ taką, że

$$\max(\frac{c_{opt}}{c}, \frac{c}{c_{opt}}) \le \rho(n)$$

gdzie $n$ jest rozmiarem danych wejściowych to algorytm aproksymacyjny nazywamy algorytmem $\rho(n)$ aproksymacyjnym. Z powyższej definicji wynika, że $1 \le \rho(n)$.

**Przykład** (algorytm aproksymacyjny rozwiązujący problem licznościowego pokrycia wierzchołkowego grafu nieskierowanego)

Niech będzie dany graf nieskierowany $G = (V, E)$. Pokryciem wierzchołkowym grafu nieskierowanego $G = (V, E)$ jest taki podzbiór $V' \subseteq V$, że jeśli $(u, v)$ jest krawędzią grafu $G$ to $u \in V'$ lub $v \in V'$. Rozmiar pokrycia wierzchołkowego to liczba należących do niego wierzchołków. Problem pokrycia wierzchołkowego (licznościowy) nazywany często problemem VERTEX-COVER polega na znalezieniu w danym grafie nieskierowanym pokrycia wierzchołkowego minimalnego rozmiaru.

Algorytm aproksymacyjny rozwiązujący problem licznościowego pokrycia wierzchołkowego jest następujący:

---

**Algorytm:** Algorytm aproksymacyjny rozwiązujący problem pokrycia wierzchołkowego grafu nieskierowanego $G = (V, E)$.

Dane WEJŚCIOWE: graf nieskierowany $G = (V, E)$

Dane WYJŚCIOWE: podzbiór $V' \subseteq V$ będący pokryciem wierzchołkowym

---

**function** *approx-vertex-cover*($V$,$E$ : **set**): **set**

**begin**

**var** $C$ , $E'$ **: set;**

$C := \varnothing$ ; wyzerowanie zbioru $C$

$E' := E$ ; pod $E'$ podstawiamy zbiór gałęzi grafu $G = (V, E)$

**while** $E' \neq \varnothing$ **do begin**

wybierz dowolną krawędź $(u, v) \in E'$;

$C := C \cup \{u, v\}$;

usuń z $E$' wszystkie krawędzie incydentne z $u$ lub $v$ ;

**end**

*approx-vertex-cover* := $C$

**end** *approx-vertex-cover*;

Powyższy algorytm można zapisać w Matlabie następująco:

---

**Algorytm:** Algorytm aproksymacyjny rozwiązujący problem pokrycia wierzchołkowego grafu nieskierowanego $G = (V, E)$ przy czym zbiór wierzchołków utożsamiamy z liczbami naturalnymi 1,2,…,*n*.

Dane WEJŚCIOWE: graf nieskierowany $G = (V, E)$ a ściślej zbiór gałęzi $E$ dany w postaci dwuwierszowej macierzy, której kolumny utożsamiamy z gałęziami grafu.

Dane WYJŚCIOWE: podzbiór $V' \subseteq V$ będący pokryciem wierzchołkowym zdefiniowany za pomocą wektora $C$ ($C(i)$=1 wtedy itylko wtedy gdy wierzchołek $i$ należy do pokrycia wierzchołkowego).

---

```
function Wynik=approx_vertex_cover(E)
% zerujemy macierz wierszowa C reprezentujaca podzbior zbioru wierzcholkow
wymiar=size(E)
for i=1:2*wymiar(2), C(i)=0; end
E1=E
while any(any(E1))~=0,
  for i=1:wymiar(2),
  if E1(1,i)~=0, u=E1(1,i), v=E1(2,i),E1(1,i)=0; E1(2,i)=0; C(u)=1; C(v)=1; end
  end
end
Wynik=C
```

**Fakt** Algorytm Approx-Vertex-Cover jest algorytmem 2 aproksymacyjnym z wielomianową złożonością czasową

**Dowód** por. [5].■

# 1.7 Równania rekurencyjne

Często analizowany algorytm zawiera rekurencyjne wywołania samego siebie wówczas dobrym, naturalnym narzędziem do obliczania lub szacowania czasu wykonania algorytmu są tzw. równania rekurencyjne.

Równanie rekurencyjne to równanie funkcyjne, w którym niewiadomą jest funkcja $T : N \to R$ (lub $T : N \cup \{0\} \to R$) a samo równanie wyraża związek pomiędzy wyrazami ciągu $T : N \to R$. Nas będą głównie interesowały funkcje $T : N \to N$, gdzie $T(n)$ jest złożonością czasową algorytmu.

Szczególną postacią równań rekurencyjnych są równania rekurencyjne liniowe o postaci

$$a_0 T(n) + a_1 T(n-1) + ... + a_k T(n-k)) = f(n) \text{ dla } n \geq k$$

gdzie $k \in N \cup \{0\}$, $a_0, a_1, ..., a_k \in R$ a $f : N \to R$ jest ustalonym ciągiem. Rozwiązaniem tego równania jest ciąg $T : N \to R$ spełniający to równanie. Liniowe równanie rekurencyjne nazywa się też liniowym równaniem różnicowym.

Metodą rozwiązywania liniowych równań rekurencyjnych jest metoda funkcji tworzących lub przekształcenie *Z*. Liniowe równania rekurencyjne służą np. do opisu liniowych stacjonarnych układów dyskretnych a więc w szczególności liniowych filtrów cyfrowych.

**Przykład 1** (równanie rekurencyjne definiujące ciąg Fibonacciego)
Ciąg Fibonacciego opisywany jest równaniem rekurencyjnym liniowym

$$T(n) = T(n-1) + T(n-2),$$

gdzie $T(0) = 0$. Można wykazać, że rozwiązaniem tego równania jest ciąg zadany wzorem:

$$T(n) = \frac{1}{\sqrt{5}} \left[ \left( \frac{1+\sqrt{5}}{2} \right)^{n+1} - \left( \frac{1-\sqrt{5}}{2} \right)^{n+1} \right]$$

■

**Przykład 2** (równanie rekurencyjne na pesymistyczną złożoność czasową algorytmu sortowania *mergesort*)
Równanie rekurencyjne na pesymistyczną złożoność czasową algorytmu *mergesort* ma postać $T(1) = \Theta(1)$, $T(n) = 2T(\lfloor n/2 \rfloor) + \Theta(n)$.

Można pokazać, że rozwiązaniem tego równania jest $T(n) = \Theta(n \lg n)$. ■

**Twierdzenie** (twierdzenie o rekurencji uniwersalnej)
Niech $a,b \in R$, $a \geq 1$, $b > 1$ będą stałymi a $f : N \cup \{0\} \to R$ dowolną ustaloną funkcją. Rozważmy następujące równania rekurencyjne dla funkcji $T : N \cup \{0\} \to R$

$$T(n) = aT(\lfloor n/b \rfloor) + f(n) \qquad \text{lub} \qquad T(n) = aT(\lceil n/b \rceil) + f(n)$$

(co równie zapisujemy czasem niezbyt poprawnie w postaci $T(n) = aT(n/b) + f(n)$).
Rozwiązanie $T : N \cup \{0\} \to R$ powyższego równania może być oszacowane asymptotycznie następująco.

Jeśli $f(n) = O(n^{\log_b a - \varepsilon})$ dla pewnej stałej $\varepsilon > 0$ to $T(n) = \Theta(n^{\log_b a})$.

Jeśli $f(n) = \Theta(n^{\log_b a})$ to $T(n) = \Theta(n^{\log_b a} \lg n)$.

Jeśli $f(n) = \Omega(n^{\log_b a + \varepsilon})$ dla pewnej stałej $\varepsilon > 0$ oraz dla pewnej stałej $c \in R$, $0 \leq c < 1$ istnieje takie $n_1 \in N$, że dla każdego $n > n_1$ mamy

$$a \cdot f(\lfloor n/b \rfloor) \leq c \cdot f(n) \text{ to } T(n) = \Theta(f(n)).$$

**Dowód** por [1]■

Funkcja $f : N \cup \{0\} \to R$ opisuje na ogół w analizie algorytmów koszt dzielenia rozwiązywanego problemu na podproblemy oraz łączenia wyników.

**Przykład 3** Rozważmy równanie rekurencyjne postaci

$$T(n) = 27 \cdot T(\lfloor n/3 \rfloor) + n$$

Rekurencję tę możemy rozwiązać za pomocą twierdzenia o rekurencji uniwersalnej. Mamy (korzystając z oznaczeń wprowadzonych w tym twierdzeniu): $a = 27$, $b = 3$, $f(n) = n$, zatem $n^{\log_b a} = O(n^{\log_3 27}) = \Theta(n^3)$. Ponieważ $f(n) = O(n^{\log_b a - \varepsilon}) = O(n^{\log_3 27 - \varepsilon})$ dla $\varepsilon = 2$ zatem

stosując twierdzenie o rekurencji uniwersalnej stwierdzamy ostatecznie, że $T(n) = \Theta(n^3)$.■

**Przykład 4** Rozważmy równanie rekurencyjne postaci

$$T(n) = T(\lfloor 2n/3 \rfloor) + 1$$

Rekurencję możemy rozwiązać za pomocą twierdzenia o rekurencji uniwersalnej. Mamy $a = 1, b = 3/2,$ $f(n) = 1,$ $n^{\log_b a} = O(n^{\log_{3/2} 1}) = n^0 = 1.$ Zatem stosując twierdzenie o rekurencji uniwersalnej stwierdzamy, że rozwiązaniem równania jest $T(n) = \Theta(\lg n)$ .■

Można wykazać następujące (podobne do twierdzenia o rekurencji uniwersalnej) twierdzenie o rozwiązaniu asymptotycznym równania rekurencyjnego

$$T(n) = \sum_{i=1}^{k} a_i T(\lfloor n/b_i \rfloor) + f(n)$$

gdzie $k \in N$, $a_i, b_i \in R$, $a_i > 0,$ $b_i \geq 2$ dla każdego $i = 1,2,...,k$ oraz $\sum_{i=1}^{k} a_i \geq 1$. Ponadto zakładamy, że funkcja $f : N \cup \{0\} \to R$ jest dodatnia, ograniczona, niemalejąca i ma taką własność: dla każdej stałej $c \in R, c > 1$ istnieją stałe $n_0 \in N$ i $d \in R,\ d > 0$ takie, że $f(\lfloor n/c \rfloor) \geq df(n)$ dla każdego $n \geq n_0$.

Z innych metod rozwiązywania równań rekurencyjnych warto wspomnieć jeszcze o dwóch metodach:

1. metodzie podstawiania, która sprowadza się do zgadnięcia rozwiązania i następnie dowiedzenia przez indukcję, że rozwiązanie jest poprawne

2.metodzie drzewa rekursji

Obie te metody omawiane są szczegółowo w pracy [1].

# 1.8. Algorytmy probabilistyczne.

Ogólnie rzecz biorąc, dzielimy algorytmy na deterministyczne i probabilistyczne. Do tej pory omawialiśmy wyłącznie algorytmy deterministyczne.

Algorytm nazywamy probabilistycznym jeśli jego zachowanie się jest określone nie tylko przez dane wejściowe lecz także przez wartości losowe uzyskiwane z generatora liczb losowych.

Najprostszy, choć mało praktyczny, generator liczb losowych to rzut monetą generujący 1 bit losowy czyli wartość ze zbioru $\{0,1\}$ lub rzut kostką do gry generujący realizację zmiennej losowej o wartościach w zbiorze $\{1,2,3,4,5,6\}$.

W praktyce często korzysta się z generatorów liczb pseudolosowych. Generator liczb pseudolosowych jest odpowiednim algorytmem deterministycznym zwracającym liczby, które ze statystycznego punktu widzenia „zachowują się losowo" tzn. , że generowane sekwencje „liczb losowych" spełniają odpowiednie testy statystyczne.

Wszystkie ważniejsze środowiska programistyczne wyposażone są w odpowiednie generatory liczb losowych. W niektórych współczesnych mikroprocesorach np. w Pentium 4 generatory liczb losowych są tzw. fizycznymi generatorami liczb losowych i realizowane są układowo wewnątrz mikroprocesora.

Algorytmy probabilistyczne (ang. randomized algorithms) są bardzo ważną kategorią algorytmów. W wielu problemach okazują się najefektywniejsze, choć dane wyjściowe typowego algorytmu probabilistycznego obarczone są na ogół pewnym szczególnym rodzajem niepewności. Algorytmy probabilistyczne nazywamy też **algorytmami randomizowanymi**.

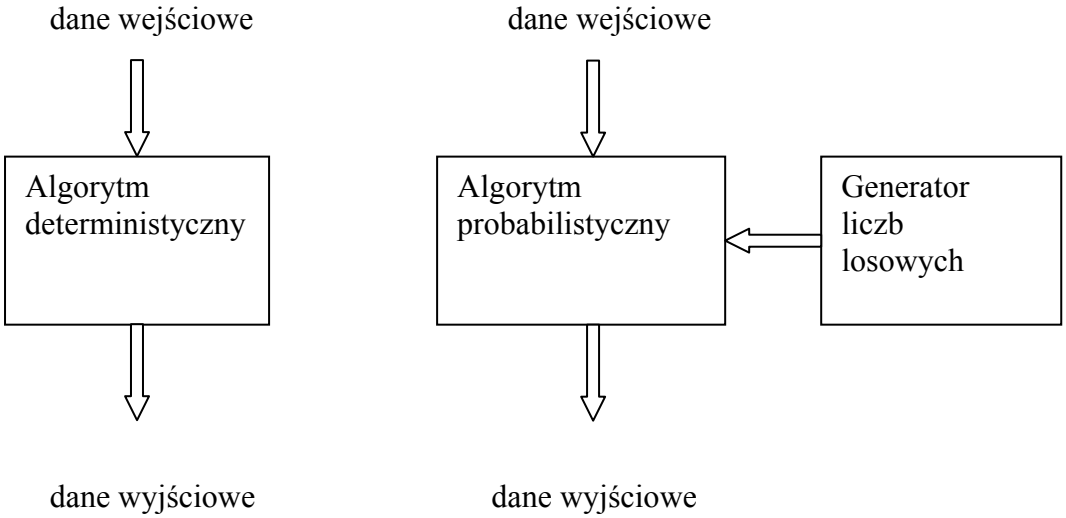
Typowa odpowiedź (tzn. dane wyjściowe algorytmu probabilistycznego) np.algorytmu testowania pierwszości liczby $n \in N$ jest taka:

„$n$ na pewno nie jest liczbą pierwszą" lub

„ $n$ najprawdopodobniej jest liczbą pierwszą"

ale prawdopodobieństwo pomyłki (jeśli przyjmiemy, że $n$ jest liczbą pierwszą a w rzeczywistości nie jest) jest $\leq 2^{-m}$ gdzie $m \in N$ jest pewnym dowolnie wybranym, ale też ustalonym parametrem algorytmu.

Z punktu widzenia ustalania prawdy absolutnej, odpowiedzi takie są oczywiście niewystarczające, natomiast informacja, że prawdopodobieństwo pomyłki jest bardzo małe np. mniejsze od $2^{-1000}$ w praktyce całkowicie nam wystarcza.




Rys. 2.1 Podział algorytmów na deterministyczne i losowe.

**Przykład 1.** Niech $(X_i)_i^{\infty}$ będzie ciągiem niezależnych zmiennych losowych o rozkładzie jednostajnym na przedziale $[a,b] \subset R$ określonych na przestrzeni probabilistycznej $(\Omega, \mathfrak{M}, P)$. Zakładamy, że $f:[a,b] \to R$ jest funkcją ciągłą na odcinku $[a,b]$. Z mocnego prawa wielkich liczb wynika, że *P* prawie wszędzie zachodzi zbieżność:

$$\frac{1}{N_0}\sum_{i=1}^{N_0} f(X_i) \to \int_a^b f(x)dx$$

W szczególności np. mamy $\frac{1}{N_0}\sum_{i=1}^{N_0} \sin(X_i) \to \int_a^b \sin x dx$. Algorytm wykorzystujący powyższą metodę obliczania całki oznaczonej musi być oczywiście wyposażony w generator liczb losowych generujący realizacje zmiennej losowej o rozkładzie jednostajnym na przedziale $[a,b]$. Opisany w przykładzie algorytm obliczania wartości całki oznaczonej jest przykładem tzw. metody Monte Carlo.

W ocenie jak bardzo możemy ufać obliczonej w algorytmie wartości

$$Y(\omega) \stackrel{df}{=} \frac{1}{N_0}\sum_{i=1}^{N_0} f(X_i(\omega)),$$

(gdzie $\omega \in \Omega$ jest zdarzeniem elementarnym) możemy skorzystać np. z następującej nierówności Czebyszewa.

Jeśli zmienna losowa jest ma wariancję (a więc $Y \in L^2(\Omega, \mathfrak{M}, P)$ ) to

$$P(|Y - E(Y)| \geq \varepsilon) \leq \frac{D^2(Y)}{\varepsilon^2}$$

Zauważmy, że z uwagi na niezależność ciągu zmiennych losowych $(X_i)_i^\infty$ również ciąg $(f(X_i))_i^\infty$ jest ciągiem niezależnych zmiennych losowych i mamy:

$$D^2(Y) = \frac{1}{N_0^2}\sum_{i=1}^{N_0} D^2(f(X_i)) = \frac{D^2(f(X_i))}{N_0}$$

Korzystając teraz z nierówności Czebyszewa mamy

$$P(|Y - E(Y)| \geq \varepsilon) = P\left(\left|Y - \int_a^b f(x)dx\right| \geq \varepsilon\right) \leq \frac{D^2(f(X_i))}{N_0\varepsilon^2}$$

■

**Przykład 2.** Algorytm Millera-Rabina. Algorytm Millera – Rabina służy do sprawdzania pierwszości liczby naturalnej. Zasada działania algorytmu jest następująca.

Mamy pewna liczbę naturalną, *n*>2 nieparzystą chcielibyśmy sprawdzić czy jest to liczba pierwsza. Sprawdzamy tezę małego twierdzenia Fermata $a^{n-1} \equiv 1 \pmod n$ dla dowolnej ustalonej liczby całkowitej *a* takiej, że $NWD(a,n) = 1$. Jeśli *n* jest pierwsza to kongruencja $a^{n-1} \equiv 1 (\mathrm{mod}\, n)$ oczywiście musi być spełniona. Jeśli okazuje się, że kongruencja $a^{n-1} \equiv 1 (\mathrm{mod}\, n)$ nie jest spełniona to na pewno liczba *n* nie jest pierwsza i wyprowadzamy odpowiedź: „*n* na pewno nie jest liczbą pierwszą”.

Może się jednak zdarzyć, że kongruencja $a^{n-1} \equiv 1 \pmod n$ zachodzi dla liczby *n* złożonej. Mało tego, istnieje nieskończenie wiele takich liczb naturalnych złożonych *n*, że dla każdego naturalnego *a* takiego, że $NWD(a,n) = 1$ mamy $a^{n-1} \equiv 1 (\mathrm{mod}\, n)$. Są to tzw. liczby Carmichaela (czyt. karmajkla). Nasz test pierwszości oparty wyłącznie o sprawdzaniu tezy małego twierdzenia Fermata jest więc zawodny bowiem niektóre liczby złożone całkiem dobrze „udają liczby pierwsze”.

Modyfikujemy więc nasz algorytm tak. Jeśli $a^{n-1} \equiv 1 \pmod n$ to dzielimy *n*-1 przez 2 (niech $m \overset{ozn}{=} (n-1)/2$ ) i obliczamy $a^m \pmod n$. Jeśli nie zachodzi $a^m \equiv 1 \pmod n$ lub $a^m \equiv -1 \pmod n$ to liczba *n* na pewno jest złożona.

Jeśli $a^m \equiv -1 \pmod{n}$ lub $a^m \equiv 1 \pmod{n}$ przy nieparzystym $m$ to stwierdzamy, że nie potrafimy dać odpowiedzi na pytanie o pierwszość liczby $n$. Jeśli $a^m \equiv 1 \pmod{n}$ oraz $m$ jest liczbą parzystą to podstawiamy $m := (n-1)/2$ i postępujemy jak poprzednio.

Można wykazać, że dla co najmniej ¾ wszystkich podstaw $a \in <2, n-1>$ powyższa metoda prowadzi w przypadku gdy $n$ jest złożona do udzielenia odpowiedzi na postawione pytanie o pierwszość $n$.

Postępujemy więc następująco. Losujemy $a \in <2, n-1>$ i obliczamy $a^m \pmod{n}$. Jeśli nie uzyskamy odpowiedzi, że $n$ jest złożona dokonujemy ponownie nowego losowania $a \in <2, n-1>$ itd. Jeśli w $r$ losowaniach ciągle nie uzyskamy odpowiedzi to albo mamy bardzo mało szczęścia i ciągle trafiamy na „złe", nie dające rozstrzygnięcia podstawy albo $n$ jest liczbą pierwszą.

■

# Zadania

**Zadanie 1**
Podać definicję notacji "O duże" i notacji "$\Theta$" i notacji "$\Omega$". Pokazać, że

$$\frac{1}{100}n^2 - 100n = \Theta(n^2).$$

**Zadanie 2**
Odpowiedzieć czy: a) $10^{n+1} = O(2^n)$, b) $2^{n+5} = O(2^n)$, c) $2^{4n} = O(2^n)$

**Zadanie 3**
Pokazać, że dla dowolnych stałych $a \in R, b \in R,\ b > 0$ mamy $(a+a)^b = \Theta(n^b)$.

**Zadanie 4**
Pokazać, że $o(g(n)) \cap \omega(g(n)) = \varnothing$

**Zadanie 5**
Pokazać, że $n! = \omega(2^n)$ oraz $n! = o(n^n)$.

**Zadanie 6**
Rozważmy zbiór wszystkich funkcji $f : N \to R$ asymptotycznie dodatnich. Pokazać, że w tym zbiorze relacje $O$, $\Theta, \Omega, o$, $\omega$ są przechodnie. Czy asymptotyczna dodatniość jest istotna dla wszystkich tych relacji.

**Zadanie 7**
Rozwiązać metodą transformaty $Z$ równanie rekurencyjne liniowe

$$T(n) = T(n-1) + T(n-2),$$

gdzie $T(0) = 0$ definiujące ciąg Fibonacciego. Wykazać, że rozwiązaniem tego równania jest ciąg $T : N \cup \{0\} \to N$ zadany wzorem:

$$T(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right]$$

**Zadanie 8**
Stosując twierdzenie o rekurencji uniwersalnej oszacować asymptotycznie rozwiązanie równania rekurencyjnego $T(n) = 9T(\lfloor n/3 \rfloor) + n$.

**Zadanie 9**
Stosując twierdzenie o rekurencji uniwersalnej oszacować asymptotycznie rozwiązanie równania rekurencyjnego $T(n) = 4T(\lfloor n/2 \rfloor) + n^2$.

**Zadanie 10**
Ocenić złożoność obliczeniową algorytmu mnożenia macierzy $A \cdot B$, gdzie $A$ i $B$ są zespolonymi macierzami $n$x$n$. Zakładamy, że algorytm oblicza iloczyn macierzy bezpośrednio w oparciu o definicję iloczynu macierzy.

**Zadanie 11**
Porównać złożoność obliczeniową rekurencyjnej i iteracyjnej wersji algorytmu obliczania

a) silni $n!$

b) $n$-tego wyrazu ciągu Fibonacciego

# Literatura

[1] T. H. Cormen, C.E. Leiserson, R.L. Rivest, C.Stein ; Wprowadzenie do algorytmów; WNT, Warszawa 2004.

[2] R. Neapolitan i K.Naimpour; Podstawy algorytmów z przykładami w C++; Hellion 2004.

[3] D.E.Knuth; Sztuka programowania - tom 1,2, 3; WNT, Warszawa 2003

[3] L.Banachowski, K.Diks, W.Rytter; Algorytmy i struktury danych; WNT, Warszawa 1996.

[4] A.Aho, J.E.Hopcroft, J.D.Ullman; Projektowanie i analiza algorytmów; Helion, 2003.

[5] V.V.Vazirani; Algorytmy aproksymacyjne; WNT, Warszawa 2004.

[6] M.Karoński, H.J.Promel; Lectures on Approximation and Randomized Algorithms; Advanced Topics in Mathematics, PWN, Warszawa 1999.