PODSTAWY INFORMATYKI

14/10/2019

WFAiS UJ, Informatyka Stosowana
I rok studiów, I stopień

Wykład 2a: Struktury danych i algorytmy

Struktury danych i algorytmy

- Typy danych i struktury danych
- Analiza algorytmów
- Sposoby zapisu algorytmów
- Rodzaje algorytmów
- Schematy blokowe i algografy
- Wybór algorytmu

*** Niektóre przykłady z wykł prof. T. Roughgarden, Standford, USA**

Struktury danych i algorytmy

- **Struktury danych to narzędzia do reprezentowania informacji która ma być przetworzona przez program komputerowy,**
- **Algorytmy to przepisy wykonania czynności niezbędnych do jej przetworzenia.**
- **Wybór algorytmu do rozwiązania konkretnego problemu programistycznego pomaga w ustaleniu, jaką strukturę danych należałoby użyć, ale i odwrotnie – wybrana struktura danych ma ogromny wpływ na szczegóły realizacji i efektywności algorytmu.**

Typy danych i struktury danych

- Dane są to „obiekty” którymi manipuluje algorytm.
- Te obiekty to nie tylko dane wejściowe lub wyjściowe (wyniki działania algorytmu), to również obiekty pośrednie tworzone i używane w trakcie działanie algorytmu.
- Dane mogą by różnych typów, do najpospolitszych należą liczby (całkowite, dziesiętne, ułamkowe) i słowa zapisane w rozmaitych alfabetach.

Typy danych i struktury danych

- **Interesują nas sposoby w jaki algorytmy mogą organizować, zapamiętywać i zmieniać zbiory danych oraz „sięgać" do nich.**
  - **Zmienne czyli „pudelka" w których chwilowo przechowujemy jakąś wartość,**
  - **Tablice czyli tabele (macierze), w których to możemy odwoływać się do indeksów,**
  - **Listy i wektory**
  - **Kolejki i stosy,**
  - **Drzewa, czyli hierarchiczna struktura danych,**
  - **Zbiory…. Grafy…. Relacje….**

Typy danych i struktury danych

- W wielu zastosowaniach same struktury danych nie wystarczają.
- Czasami potrzeba bardzo obszernych zasobów danych, stanowiących dla wielu algorytmów potencjalne dane wejściowe, a więc mające ustaloną strukturę i nadające się do odszukiwania i manipulowania nimi. Nazywa się je bazami danych (relacyjne i hierarchiczne).
- Kolejny krok to bazy wiedzy, których elementami są bazy danych, a które zawierają również informacje o związkach pomiędzy danymi.

Algorytmy

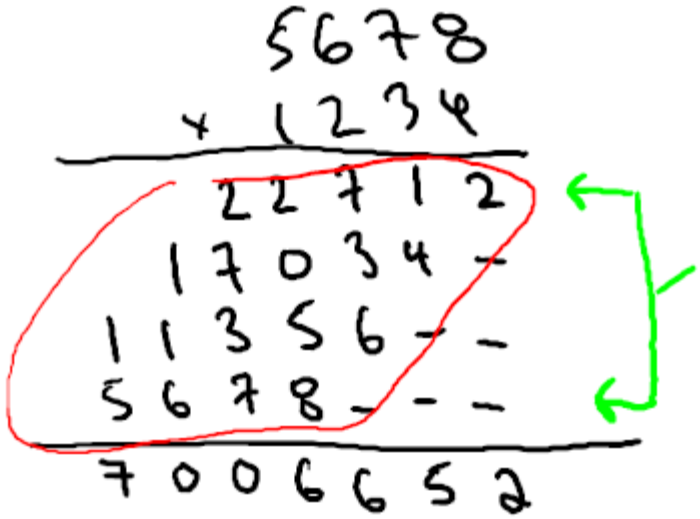
- **Algorytm to „przepis postępowania" prowadzący do rozwiązania konkretnego zadania; zbiór poleceń dotyczących pewnych obiektów (danych) ze wskazaniem kolejności w jakiej mają być wykonane". Jest jednoznaczną i precyzyjną definicją (specyfikacją) kroków które mogą być wykonywane „mechanicznie".**
- **Algorytm odpowiada na pytanie „jak to zrobić" postawione przy formułowaniu zadania. Istota algorytmu polega na rozpisaniu całej procedury na kolejne, możliwie elementarne kroki.**
- **Algorytmiczne myślenie można kształtować niezależnie od programowania komputerów, chociaż każdy program komputerowy jest zapisem jakiegoś algorytmu.**

Analiza algorytmów

- **Analiza algorytmów i powiązanych z nimi struktur danych.**
  - **Znalezienie najlepszych sposobów wykonywania najczęściej spotykanych poleceń,**
    - **musimy nauczyć się podstawowych technik projektowania dobrych algorytmów.**
  - **Zrozumienie w jaki sposób wykorzystywać struktury danych i algorytmy tak, by tworzyć efektywne (szybkie) programy.**

Mnożenie dwóch liczb całkowitych (szkolny algorytm)

- **Wejście: dwie n-cyfrowe liczby x, y**
- **Wyjście: z = x * y**
- **Operacje: mnożenie i dodawanie dwóch 1-dno cyfrowych liczb**



Ilość operacji:
~ n dla każdego wiersza

Ilość wszystkich operacji:
~ n * n = n^2
z dokładnością do stałej

Mnożenie dwóch liczb całkowitych (algorytm Karatsuba)

- **Krok 1: policz** $a \cdot c = 672$
- **Krok 2: policz** $b \cdot d = 2652$
- **Krok 3: policz**

$(a+b)(c+d) = 134 \cdot 46 = 6164$

- **Krok 4: policz** $③ - ② - ① = 2840$
- **Krok 5:**

$$\begin{array}{r} 6720000 \\ 2652 \\ 284000 \\ \hline 7006652 \end{array} = (1234)(5678)$$

Mnożenie dwóch liczb całkowitych (algorytm Karatsuby)

- **Rozpisujemy liczby x, y:**

$$x = 10^{n/2}a + b \qquad y = 10^{n/2}c + d$$

a,b,c,d są liczbami n/2 cyfrowymi

- **Reprezentujemy iloczyn jako**

$$x.y = (10^{n/2}a + b)(10^{n/2}c + d)$$
$$= 10^n ac + 10^{n/2}(ad + bc) + bd$$

- Należy obliczyć:
  - (1) ac (2) bd (3) (a+b) (c+d) = ac+ad+bc+bd
  - Gauss trick: (3) – (1) – (2) = ad + bc

Kilka mnożeń i dodawań liczb n/2 cyfrowych, czy to jest lepszy algorytm?

Algorytm Hornera

- **Załóżmy, że mamy policzyć wartość wielomianu postaci:**

$$f(x) = a_0 x^n + a_1 x^{n-1} + \ldots + a_{n-1} x + a_n$$

dla danych liczb a_0, a_1, ..., a_n, w danym punkcie x_0.

- **Algorytm polegający na bezpośrednim liczeniu ze wzoru wymaga n dodawań i (n-2) potęgowań lub (2n-1) mnożeń co w wyniku daje niedokładności (błąd względny i bezwzględny).**
  - **Warto poszukać innego rozwiązania.**

Algorytm Hornera

Przedstawiamy wielomian w postaci:

$$f(x) = (\ldots((a_0x + a_1)x + a_2)x + \ldots + a_{n-1})x + a_n$$

to otrzymujemy następującą metodę na obliczanie wielomianu:

$$b_0 = a_0$$
$$b_1 = b_0x_0 + a_1$$
$$b_2 = b_1x_0 + a_2$$

$$b_n = b_{n-1}x_0 + a_n$$

gdzie b_i oznacza wartość i-tego nawiasu dla x równego x_0, a b_n szukaną wartość wielomianu.

Algorytm wymaga n dodawań i n mnożeń.

Algorytm Hornera

- Otrzymana metoda to tzw. **Algorytm Hornera** obliczania wartości wielomianu.
- Algorytm ten jest numerycznie poprawny i jest jedynym algorytmem który minimalizuje liczbę dodawań i mnożeń przy obliczaniu wartości wielomianu wg. podanej postaci.

Sposoby zapisu algorytmu

- Najprostszy sposób zapisu to zapis słowny
  - Pozwala określić kierunek działań i odpowiedzieć na pytanie, czy zagadnienie jest możliwe do rozwiązania.
- Bardziej konkretny zapis to lista kroków
  - Staramy się zapisać kolejne operacje w postaci kolejnych kroków które należy wykonać.
- Bardzo wygodny zapis to zapis graficzny
  - schematy blokowe i grafy.
- Bardziej zaawansowana forma to zapis przy pomocy uproszczonego kodu języka programowania tzw. pseudo-kod

Opis słowny

Przykład: dodanie dwóch liczb

- **Sformułowanie zadania:**
  oblicz sumę dwóch liczb naturalnych: a,b. Wynik oznacz przez S.
- **Dane wejściowe: dwie liczby a i b**
- **Cel obliczeń: obliczenie sumy S = a + b**
- **Dodatkowe ograniczenia: sprawdzenie warunku dla danych wejściowych np. czy a, b są naturalne.**

Lista kroków

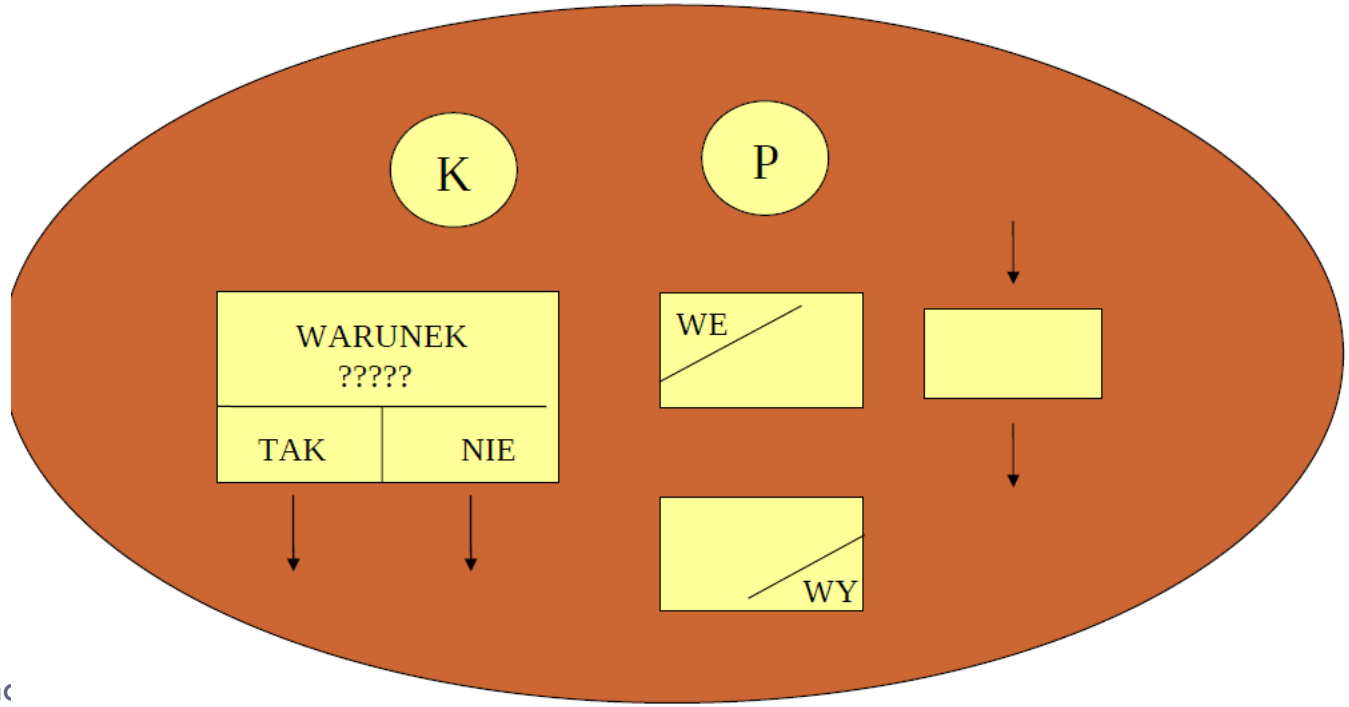
- **Zapis algorytmu przy pomocy listy kroków:**
  - **sformułowanie zagadnienia (zadanie algorytmu),**
  - **określenie zbioru danych potrzebnych do rozwiązania zagadnienia (określenie czy zbiór danych jest właściwy),**
  - **określenie przewidywanego wyniku (wyników): co chcemy otrzymać i jakie mogą być warianty rozwiązania,**
  - **zapis kolejnych ponumerowanych kroków, które należy wykonać, aby przejść od punktu początkowego do końcowego.**

Algorytm - przykład

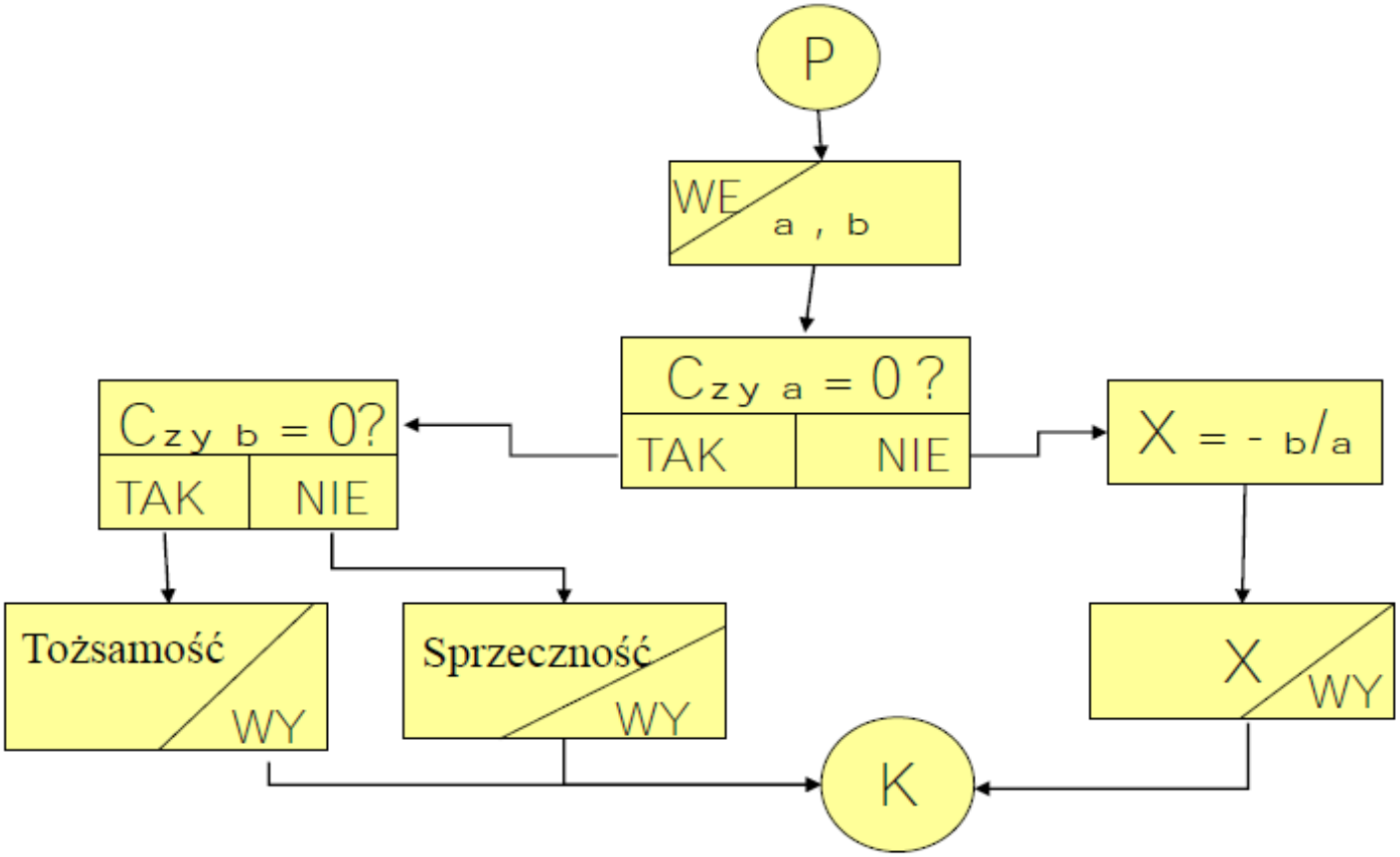
- **Sformułowanie zadania**
  Znajdź rozwiązanie równania liniowego postaci $a \cdot x + b = 0$.
  Wynikiem jest wartość liczbowa lub stwierdzenie dlaczego nie ma jednoznacznego rozwiązania.
- **Dane wejściowe**
  Dwie liczby rzeczywiste a i b
- **Cel obliczeń (co ma być wynikiem)**
  Obliczenie wartości x lub stwierdzenie, że równanie nie ma jednoznacznego rozwiązania.
  - **gdy $a = 0$ to sprawdź czy $b = 0$, jeśli tak to równanie sprzeczne lub tożsamościowe**
  - **gdy $a \neq 0$ to oblicz $x = -b/a$**

Schematy blokowe i algografy

Schemat blokowy to sposób zapisu algorytmu prezentujący kolejne kroki (instrukcje) które należy wykonać w celu osiągnięcia postawionego celu. Wykorzystuje pewnie zbiór figur geometrycznych reprezentujących pewne kategorie operacji na danych oraz połączenia które wskazują kierunek ich przetwarzania i możliwe alternatywne przejścia.



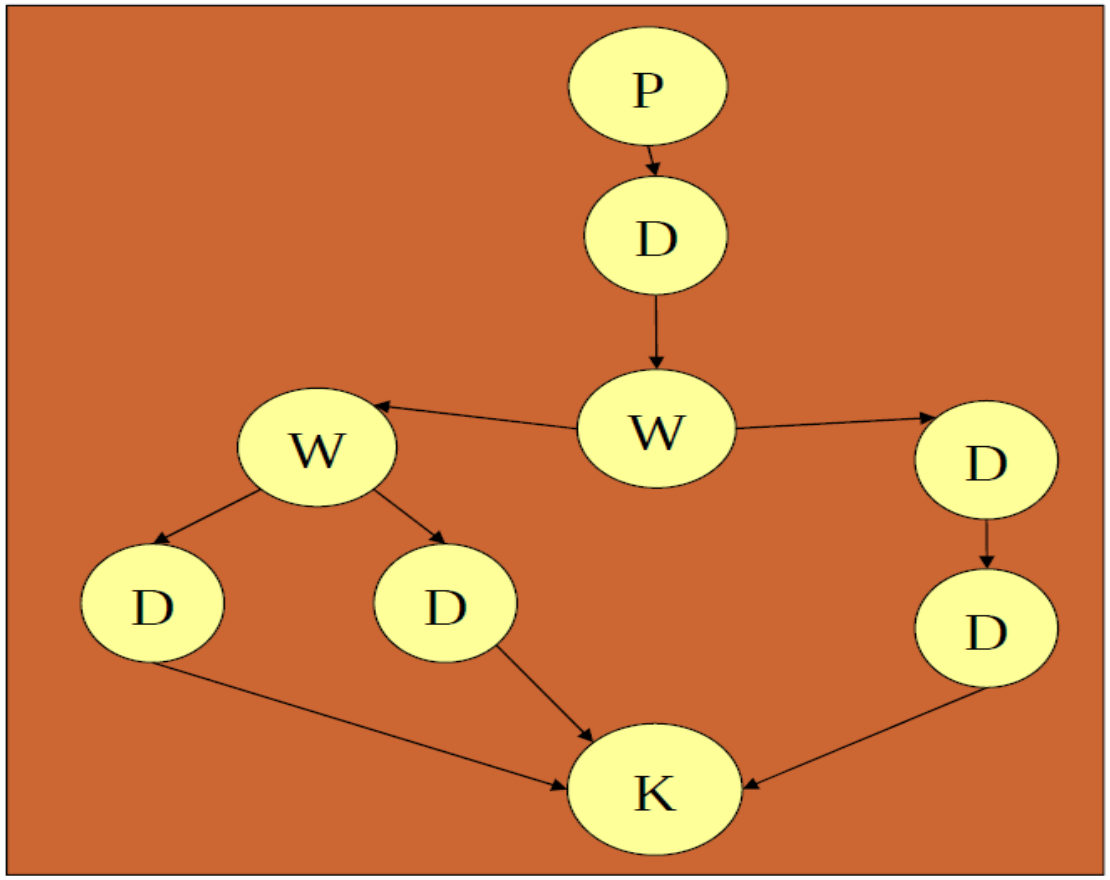
Schemat blokowy rozwiązania równania liniowego



Grafy

- **Graf symbolizuje przepływ informacji.**
- **Graf składa się z węzłów i ścieżek.**
- **W przypadku algorytmów graf można wykorzystać aby w uproszczonej formie zilustrować ilość różnych dróg prowadzących do określonego w zadaniu celu.**
- **Graf pozwala wykryć drogi, które nie prowadzą do punktu końcowego, których to poprawny algorytm nie powinien posiadać.**

Graf algorytmu rozwiązania równania liniowego



- P – początek
- K – koniec
- D – działanie
- W – warunek

Grafy

- **Jeżeli w grafie znajduje się ścieżka, która nie doprowadza do węzła końcowego, to mamy do czynienia z niepoprawnym grafem.**
- **W programie przygotowanym na podstawie takiego grafu, mamy do czynienia z przerwaniem próby działania i komunikatem o zaistnieniu jakiegoś błędu w działaniu.**
- **Węzeł grafu może mieć dwa wejścia jeżeli ilustruje pętle. Wtedy liczba ścieżek początek-koniec może być nieskończona, gdyż nieznana jest liczba obiegów pętli.**

Schemat blokowy czy graf ?

- **Graf to tylko schemat kontrolny służący do sprawdzenia algorytmu.**
  - **Brak informacji o wykonywanych operacjach**
- **Schemat blokowy służy jako podstawa do tworzenia programów.**

Rodzaje algorytmów

Algorytm liniowy:

- **Ma postać ciągu kroków których jest liniowa ilość (np. stała albo proporcjonalna do liczby danych) które muszą zostać bezwarunkowo wykonane jeden po drugim.**
- **Algorytm taki nie zawiera żadnych warunków ani rozgałęzień: zaczyna się od podania zestawu danych, następnie wykonywane są kolejne kroki wykonawcze, aż dochodzimy do wyniku.**

Rodzaje algorytmów

Algorytm z rozgałęzieniem:

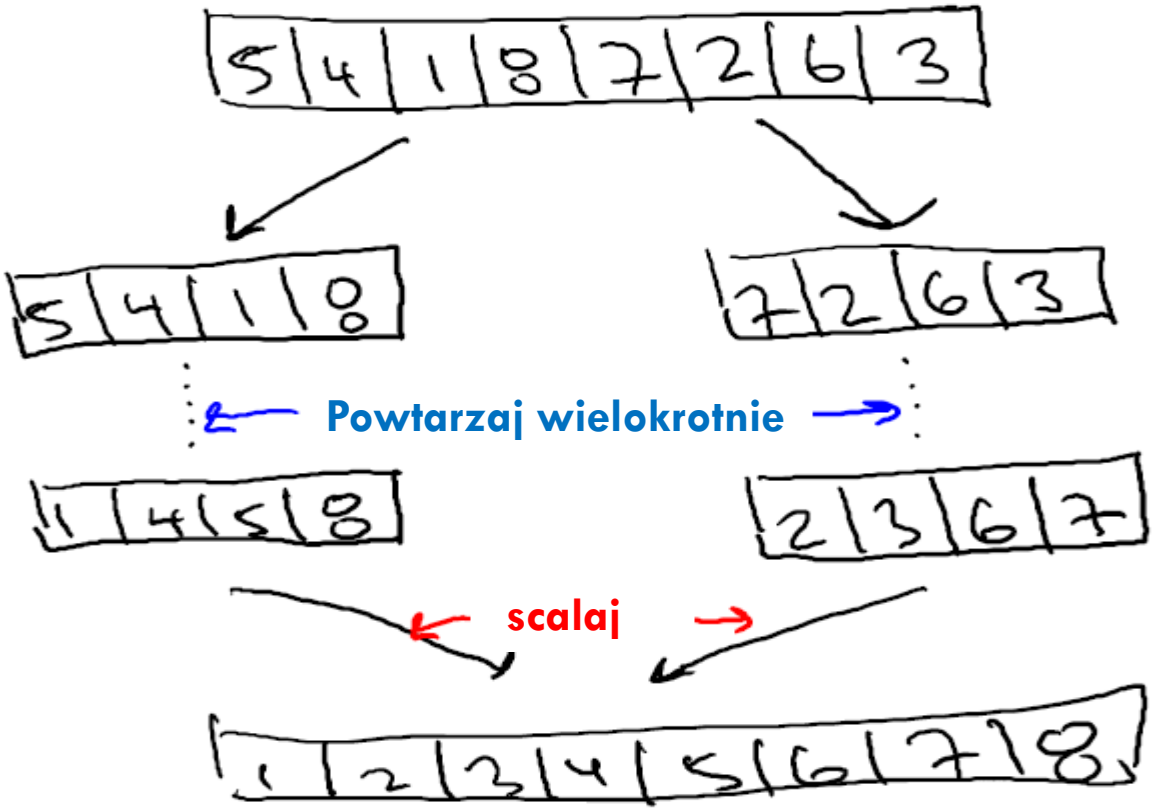
- Większość algorytmów zawiera rozgałęzienia będące efektem sprawdzania warunków. Wyrażenia warunkowe umożliwiają wykonanie zadania dla wielu wariantów danych i rozważanie różnych przypadków.
- Powtarzanie różnych działań ma dwojaką postać:
  - liczba powtórzeń jest z góry określona (przed rozpoczęciem cyklu), alg. najczęściej związany z działaniami na tablicach,
  - liczba powtórzeń jest nieznana (zależy od spełnienia pewnego warunku), alg. najczęściej związany z obliczeniami typu iteracyjnego.

Algorytmy: „dziel i zwyciężaj”

- **Metoda: „dziel i zwyciężaj” :**
  - **Dzielimy problem na mniejsze części tej samej postaci co pierwotny.**
  - **Teraz te pod-problemy dzielimy dalej na coraz mniejsze, używając tej samej metody, aż rozmiar problemu stanie się tak mały, że rozwiązanie będzie oczywiste lub będzie można użyć jakiejś innej efektywnej metody rozwiązania.**
  - **Rozwiązania wszystkich pod-problemów muszą być połączone w celu utworzenia rozwiązania całego problemu.**
- **Ten typ algorytmów zazwyczaj jest implementowany z zastosowaniem technik rekurencyjnych.**

Algorytmy: „dziel i zwyciężaj”

- **Mamy posortować tablicę liczb, zakładamy że są różn**



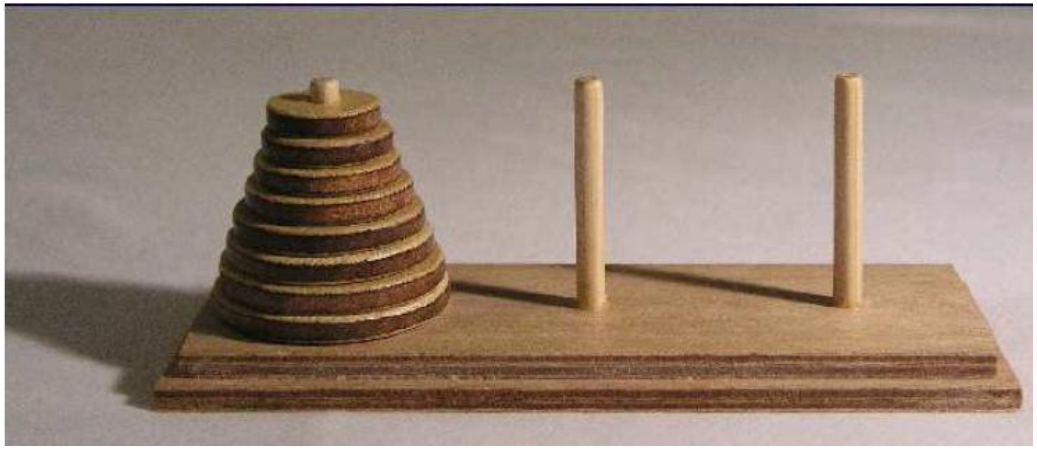
Algorytmy: „dziel i zwyciężaj”

- **Jak znaleźć minimum ciągu liczb?**
  - **Dzielimy ciąg na dwie części, znajdujemy minimum w każdej z nich, bierzemy minimum z obu liczb jako minimum ciągu.**

- **Jak sortować ciąg liczb?**
  - **Dzielimy na dwie części, każdą osobno sortujemy a następnie łączymy dwa uporządkowane ciągi (scalamy).**

Algorytmy inne: wieże Hanoi

- **Wieże Hanoi**



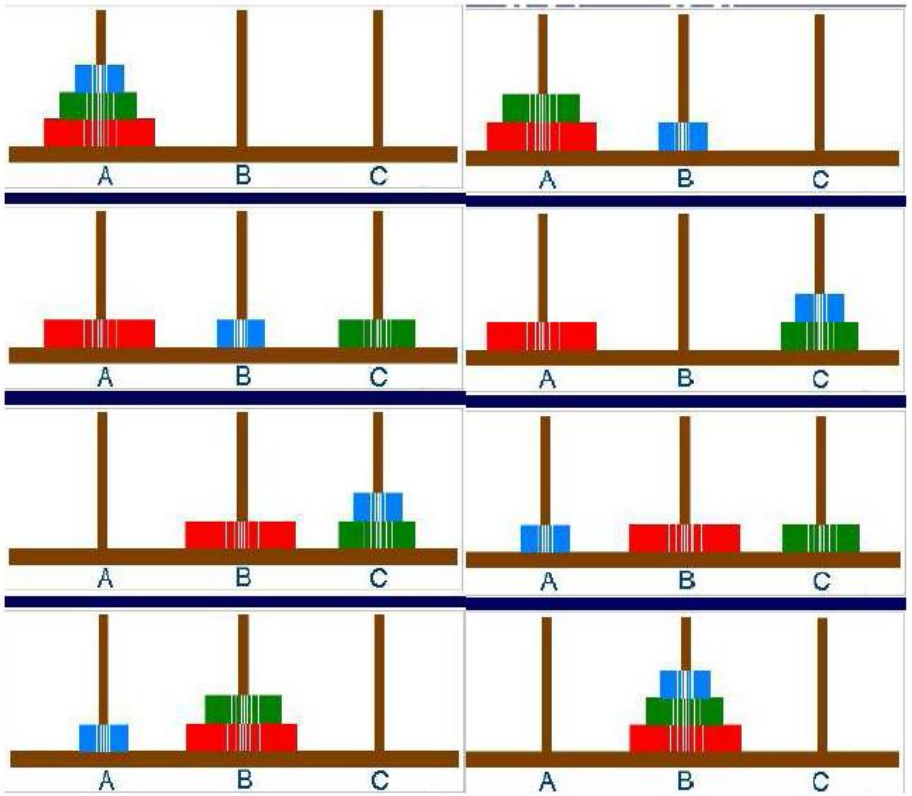
Zadanie polega na przeniesieniu wieży z krążków na inny pręt za zachowaniem następujących reguł.

- **jednorazowo można przenosić tylko 1 krążek**
- **dopuszczalne jest umieszczanie tylko mniejszego krążka na większym.**

Ilość operacji wzrasta bardzo szybko z ilością krążków, 2^n dla n- krążków. Dla n=64: 18.5 tryliona operacji.

Algorytm inne: wieże Hanoi

- **W celu przeniesienia n krążków z A do B należy**
  - **Przenieść n-1 krążków z A do C**
  - **Przenieść n-ty krążek z A do B**
  - **Przenieść n-1 krążków z C do B**



Algorytmy oparte na programowaniu dynamicznym

- **Można stosować wówczas, kiedy problem daje się podzielić na wiele pod-problemów, których rozwiązania są możliwe do zapamiętania w jedno-, dwu- lub wielowymiarowej tablicy w taki sposób że w pewnej określonej kolejności można je wszystkie (a więc i cały problem) efektywnie rozwiązać.**

Jak obliczać ciąg Fibonacciego?

$$F(i) = \begin{cases} 1 & \text{jeśli } i = 1 \\ 1 & \text{jeśli } i = 2 \\ F(i-2)+F(i-1) & \text{jeśli } i > 2 \end{cases}$$

- **Aby obliczyć F(n), wartość F(k), gdzie k<n musimy wyliczyć F(n-k) razy.**
- **Liczba obliczeń rośnie wykładniczo.**
- **Korzystnie jest więc zachować (zapamiętać w tablicy) wyniki wcześniejszych obliczeń (F(k)).**

Jak obliczać liczbę kombinacji?

- **Liczba kombinacji (podzbiorów) r-elementowych ze zbioru n-elementowego oznaczana ($\frac{n}{r}$), dana jest wzorem:**

$$\binom{n}{r} = n!/(r!\,(n-r)!)$$

Możemy użyć wzorów:

$$\binom{n}{r} = 1, \text{ jeśli } r = 0 \text{ lub } n = r$$

$$\binom{n}{r} = \binom{n-1}{r-1} + \binom{n-1}{r} \text{ dla } 0 < r < n$$

Obliczamy rząd po rzędzie w **trójkącie Pascala**

n \ r	0	1	2	3	4
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1

Algorytmy z powrotami

- Często możemy zdefiniować jakiś problem jako poszukiwanie rozwiązania wśród wielu możliwych przypadków.
- Dane:
  - Pewna przestrzeń stanów, przy czym stan jest to sytuacją stanowiąca rozwiązanie problemu albo mogąca prowadzić do rozwiązania
  - Sposób przechodzenia z jednego stanu do drugiego.
  - Mogą istnieć stany które nie prowadzą do rozwiązania.

Przykładami tego typu algorytmów są gry.

Algorytmy z powrotami

Metoda powrotów

- **Wymaga zapamiętania wszystkich wykonanych ruchów czy też wszystkich odwiedzonych stanów aby możliwe było cofanie się (powroty).**
- **Stanów mogą być tysiące lub miliony więc bezpośrednie zastosowanie metody powrotów, mogące doprowadzić do odwiedzenia wszystkich stanów, może być zbyt kosztowne.**
- **Inteligentny wybór następnego posunięcia, tzw. funkcja oceniająca, może znacznie poprawić efektywność algorytmu.**
  - **Np. aby uniknąć przeglądania nieistotnych fragmentów przestrzeni stanów.**

Wybór algorytmu

- **Regułą jest że należy implementować algorytmy najprostsze, które wykonują określone zadanie.**
- **Prosty algorytm to**
  - **łatwiejsza implementacja, czytelniejszy kod**
  - **łatwość testowania**
  - **łatwość pisania dokumentacji,….**
- **Jeśli program ma działać wielokrotnie, jego wydajność i wykorzystywany algorytm stają się bardzo ważne.**
- **Błędy zaokrągleń, powstające przy reprezentacji liczb, a także przy wykonywaniu działań na nich rozwinęły się w samodzielna dziedzinę tzw. analiza numeryczna.**

Wybór algorytmu

- **Istnieją również inne zasoby, które należy niekiedy oszczędnie wykorzystywać w pisanych programach:**
  - **ilość przestrzeni pamięciowej wykorzystywanej przez zmienne**
  - **generowane przez program obciążenie sieci komputerowej**
  - **ilość danych odczytywanych i zapisywanych na dysku**
  - **mniej obliczeń to lepsza dokładność numeryczna (zaokrąglenia)**

Wybór algorytmu

- **Zrozumiałość i efektywność: to są często sprzeczne cele. Typowa jest sytuacja w której programy efektywne dla dużej ilości danych są trudniejsze do napisania/zrozumienia.**
  - **Np. sortowanie przez wybieranie (łatwy, nieefektywny dla dużej ilości danych) i sortowanie przez „dzielenie i scalanie" (trudniejszy, dużo efektywniejszy).**
- **Zrozumiałość to pojęcie względne, natomiast efektywność można obiektywnie zmierzyć: testy wzorcowe, analiza złożoności obliczeń.**

Efektywność algorytmu

- **Czas działania:**
  - **Oznaczamy przez funkcje T(n) liczbę jednostek czasu, które zajmuje wykonanie programu lub algorytmu w przypadku problemu o rozmiarze n.**
  - **Funkcje te nazywamy czasem działania. Dość często czas działania zależy od konkretnych danych wejściowych, nie tylko ich rozmiaru. W takim przypadku, funkcje T(n) definiuje się jako najmniej korzystny przypadek z punktu widzenia kosztów czasowych. Inną wyznaczaną wielkością jest też czas średni, czyli średni dla różnych danych wejściowych.**

Testy wzorcowe

- Podczas porównywania dwóch lub więcej programów zaprojektowanych do wykonywania tego samego zadania, opracowujemy niewielki zbiór typowych danych wejściowych które mogą posłużyć jako dane wzorcowe (ang. benchmark).
- Powinny być one reprezentatywne i zakłada się że program dobrze działający dla danych wzorcowych będzie też dobrze działał dla wszystkich innych danych.
- Np. test wzorcowy umożliwiający porównanie algorytmów sortujących może opierać się na jednym małym zbiorze danych, np. zbiór pierwszych 20 cyfr liczby ∏; jednym średnim, np. zbiór kodów pocztowych województwa krakowskiego; oraz na dużym zbiorze takim jak zbiór numerów telefonów z obszaru Krakowa i okolic.
- Przydatne jest też sprawdzenie jak algorytm działa dla ciągu już posortowanego (często działają kiepsko).

Uwagi końcowe

- **Na wybór najlepszego algorytmu dla tworzonego programu wpływa wiele czynników, najważniejsze to:**
  - **poprawność (zwraca zawsze poprawny wynik)**
  - **prostota,**
  - **łatwość implementacji**
  - **efektywność**

Pytania do wykładu

1) **Co to jest algorytm i jakie znasz sposoby jego zapisu?**
2) **Scharakteryzuj, na czym polegają następujące typy algorytmów:**
   - **liniowy**
   - **z rozgałęzieniem**
   - **z powrotami**
   - **„dziel i zwyciężaj”**
   - **zachłanny**
   - **oparty o programowanie dynamiczne**
3) **Według jakich kryteriów efektywność algorytmu?**
4) **W jaki sposób badamy czas działania algorytmu?**

Wykład 2b: Złożoność obliczeniowa

Złożoność obliczeniowa algorytmów

- **Notacja *„wielkie 0”***
- **Notacja Ω i Θ**
- ***Algorytm Hornera***
- ***Przykłady rzędów złożoności***
- ***Klasy złożoności algorytmów***
- ***Funkcje niewspółmierne***
- ***Analiza czasu działania algorytmu***
  - ***Instrukcje proste; instrukcje warunkowe; bloki instrukcji***
- ***Efektywność algorytmu***

Złożoność obliczeniowa

- **Złożoność obliczeniowa:**
  - **Jest to miara służąca do porównywania efektywności algorytmów.**
  - **Mamy dwa kryteria efektywności:**
    - **Czas,**
    - **Pamięć**
- Do oceny efektywności stosujemy jednostki logiczne wyrażające związek miedzy rozmiarem danych N (wielkość pliku lub tablicy) a ilością czasu T potrzebną na ich przetworzenie.

Złożoność asymptotyczna

- Funkcja wyrażająca zależność miedzy N a T jest zwykle bardzo skomplikowana, a jej obliczenie ma znaczenie jedynie w odniesieniu do dużych rozmiarów danych
- Przybliżona miara efektywności to tzw. **złożoność asymptotyczna**.

Które człony są ważne?

Funkcja: $f(n) = n^2 + 100n + \log_{10} n + 1000$

n	f(n)	n^2	100•n	$\log_{10} n$	1000
1	1 101	0.1%	9%	0.0%	91%
10	2 101	4.8%	48%	0.05%	48%
100	21 002	48%	48%	0.001%	4.8%
10^3	1 101 003	91%	9%	0.0003%	0.09%
10^4		99%	1%	0.0%	0.001%
10^5		99.9%	0.1%	0.0%	0.0000%

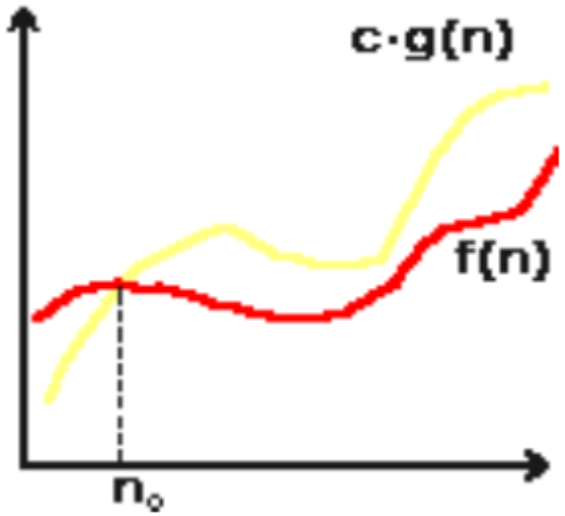
- n – rozmiar danych,
- f(n) – ilość wykonywanych operacji

Dla dużych wartości n funkcja rośnie jak n^2, pozostałe składniki mogą być zaniedbane.

Notacja „wielkie O ”

Definicja:

f(n) jest O(g(n)), jeśli istnieją liczby dodatnie c i n_0 takie że:
$f(n) < c \bullet g(n)$ dla wszystkich $n \geq n_0$.



Przykład:

- **$f(n) = n^2 + 100n + \log_{10} n + 1000$**

możemy przybliżyć jako:

$f(n) \approx n^2 + 100n + O(\log_{10} n)$

albo jako:

$f(n) \approx O(n^2)$

- **Notacja „wielkie O ” ma kilka pozytywnych własności które możemy wykorzystać przy szacowaniu efektywności algorytmów.**

Własności notacji „*wielkie O*”

- **Własność 1** (przechodniość):
  - **Jeśli f(n) jest *O(g(n))* i g(n) jest *O(h(n))*, to f(n) jest *O(h(n))***
- **Własność 2:**
  - **Jeśli f(n) jest *O(h(n))* i g(n) jest *O(h(n))*, to f(n)+g(n) jest *O(h(n))***
- **Własność 3:**
  - **Funkcja an^k jest $O(n^k)$**
- **Własność 4:**
  - **Funkcja n^k jest $O(n^{k+j})$ dla dowolnego dodatniego j**

Własności notacji „*wielkie O*”

- **Z tych wszystkich własności wynika, że dowolny wielomian jest „*wielkie O*” dla n podniesionego do najwyższej w nim potęgi, czyli :**
  $f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0$ jest $O(n^k)$

  (jest też oczywiście $O(n^{k+j})$ dla dowolnego dodatniego j)

Własności notacji „wielkie O”

- **Własność 5:**
  - **Jeśli f(n) = c g(n), to f(n) jest $O(g(n))$**
- **Własność 6:**
  - **Funkcja $\log_a n$ jest $O(\log_b n)$ dla dowolnych a i b większych niż 1**
- **Własność 7:**
  - **$\log_a n$ jest $O(\log_2 n)$ dla dowolnego dodatniego a**

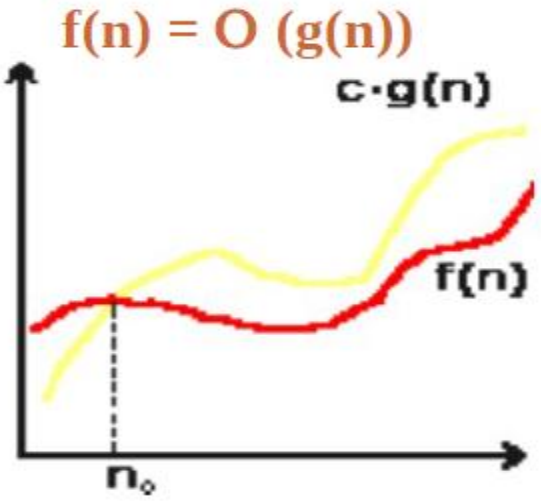
Własności notacji „*wielkie O*”

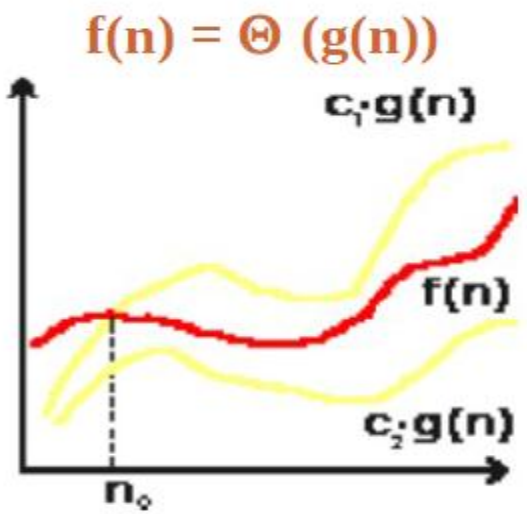
- Jedną z najważniejszych funkcji przy ocenianiu efektywności algorytmów jest **funkcja logarytmiczna**.
- Jeżeli można wykazać że złożoność algorytmu jest rzędu logarytmicznego, algorytm można traktować jako bardzo dobry.
- Istnieje wiele funkcji lepszych w tym sensie niż logarytmiczna, jednak zaledwie kilka spośród nich, jak $O(\log_2 \log_2 n)$ czy $O(1)$ ma praktyczne znaczenie.

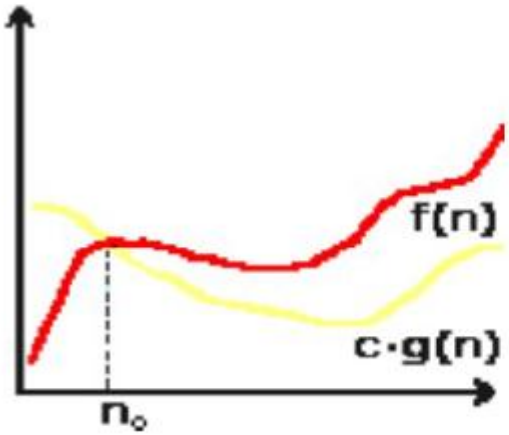
Notacja Ω i Θ

- Notacja *„wielkie O”* odnosi się do górnych ograniczeń funkcji. Istnieje symetryczna definicja dotycząca dolnych ograniczeń
- Definicja
  - f(n) jest Ω(g(n)), jeśli istnieją liczby dodatnie c i n_0 takie że, $f(n) \geq c\, g(n)$ dla wszystkich $n \geq n_0$.
- Równoważność
  - f(n) jest Ω(g(n)) wtedy i tylko wtedy, gdy g(n) jest *O(f(n))*
- Definicja
  - f(n) jest Θ(g(n)), jeśli istnieją takie liczby dodatnie c_1, c_2 i n_0 takie że, $c_1 g(n) \leq f(n) \leq c_2 g(n)$ dla wszystkich $n \geq n_0$.

Notacja O, Ω i Θ







$f(n) = \Omega(g(n))$

O czym należy pamiętać

- Celem wprowadzonych wcześniej sposobów zapisu (notacji) jest porównanie efektywności rozmaitych algorytmów zaprojektowanych do rozwiązania tego samego problemu.
- Jeżeli będziemy stosować tylko notacje „wielkie O” do reprezentowania złożoności algorytmów, to niektóre z nich możemy zdyskwalifikować zbyt pochopnie.

Pamiętaj o dużych stałych

Przykład:

- **Załóżmy, że mamy dwa algorytmy rozwiązujące pewien problem, wykonywana przez nie liczba operacji to odpowiednio $10^8 n$ i $10n^2$. Pierwsza funkcja jest O(n), druga $O(n^2)$.**
- **Opierając się na informacji dostarczonej przez notacje *„wielkie O”* odrzucilibyśmy drugi algorytm ponieważ funkcja kosztu rośnie zbyt szybko.**
- **To prawda … ale dopiero dla odpowiednio dużych n, ponieważ dla $n<10^7$ drugi algorytm wykonuje mniej operacji niż pierwszy.**
- **Istotna jest więc też stała (10^8), która w tym przypadku jest zbyt duża aby notacja była znacząca.**

Przykłady rzędów złożoności

- Algorytmy można klasyfikować ze względu na złożoność czasową lub pamięciową.
- W związku z tym wyróżniamy wiele klas algorytmów.
  - Algorytm stały: czas wykonania pozostaje taki sam niezależnie od ilości przetwarzanych elementów.
  - Algorytm kwadratowy: czas wykonania wynosi $O(n^2)$.
  - Algorytm logarytmiczny: czas wykonania wynosi $O(\log n)$.
  - itd …
- Analiza złożoności algorytmów jest niezmiernie istotna i nie można jej lekceważyć argumentując potencjalną szybkością obliczeń komputera. Nie sposób jej przecenić szczególnie zastanawiając się nad doborem struktury danych.

Najczęstsze złożoności

- **log(n)** – złożoność logarytmiczna
- **n** – złożoność liniowa
- **n log(n)** – złożoność liniowo-logarytmiczna
- $\mathbf{n^k}$ – złożoność wielomianowa
- $\mathbf{2^n}$ – złożoność wykładnicza
- **n!** – złożoność wykładnicza ponieważ $n! > 2^n$ już dla $n=4$.

Klasy złożoności algorytmów

- **Czasy wykonania logarytmów na komputerze działającym z szybkością 1 instrukcja / µs.**

klasa	złożoność	liczba operacji i czas wykonania			
	n	**10**		**10^3**	
stały	$O(1)$	1	1µs	1	1 µs
logarytmiczny	$O(\log n)$	3.32	3µs	9.97	10 µs
liniowy	$O(n)$	10	10µs	10^3	1ms
kwadratowy	$O(n^2)$	10^2	100µs	10^6	1s
wykładniczy	$O(2^n)$	1024	10ms	10^{301}	>> 10^{16} lat

Funkcje niewspółmierne

- **Bardzo wygodna jest możliwość porównywania dowolnych funkcji f(n) i g(n) za pomocą notacji *„wielkie O”***
  - **albo f(n) = *O(g(n))***
  - **albo g(n) = *O(f(n))***

  Albo jedno i drugie czyli f(n) = *Θ(g(n))*.

- **Istnieją pary funkcji niewspółmiernych (ang. incommensurate), z których żadne nie jest „wielkim O” dla drugiej.**

Funkcje niewspółmierne

- **Przykład:**

  Rozważmy funkcję f(n)=n dla nieparzystych n oraz $f(n)=n^2$ dla parzystych n.

  - **Oznacza to, że f(1)=1, f(2)=4, f(3)=3, f(4)=16, f(5)=5 itd…**
  - **Podobnie, niech $g(n)=n^2$ dla nieparzystych n oraz g(n)=n dla parzystych n.**
  - **W takim przypadku, funkcja f(n) nie może być O(g(n)) ze względu na parzyste argumenty n, analogicznie g(n) nie może być O(f(n)) ze względu na nieparzyste elementy n.**

  Obie funkcje mogą być ograniczone jako $O(n^2)$.

Analiza czasu działania programu

- **Mając do dyspozycji definicję *„wielkie O”* oraz własności (1)-(7) będziemy mogli, wg. kilku prostych zasad, skutecznie analizować czasy działania większości programów spotykanych w praktyce.**
- **Efektywność algorytmów ocenia się przez szacowanie ilości czasu i pamięci potrzebnych do wykonania zadania, dla którego algorytm został zaprojektowany.**
- **Najczęściej jesteśmy zainteresowani złożonością czasową, mierzoną zazwyczaj liczbą przypisań i porównań realizowanych podczas wykonywania programu.**
- **Bardzo często interesuje nas tylko złożoność asymptotyczna, czyli czas działania dla dużej ilości analizowanych zmiennych.**

Czas działania instrukcji prostych

- **Przyjmujemy zasadę że czas działania pewnych prosty operacji na danych wynosi O(1), czyli jest niezależny od rozmiaru danych wejściowych.**
  - **Operacje arytmetyczne, np. (+), (-)**
  - **Operacje logiczne (&&)**
  - **Operacje porównania (<=)**
  - **Operacje dostępu do struktur danych, np. indeksowanie tablic (A[i])**
  - **Proste przypisania, np. kopiowanie wartości do zmiennej.**
  - **Wywołania funkcji bibliotecznych, np. scanf lub printf**
- **Każdą z tych operacji można wykonać za pomocą pewnej (niewielkiej) liczby rozkazów maszynowych.**

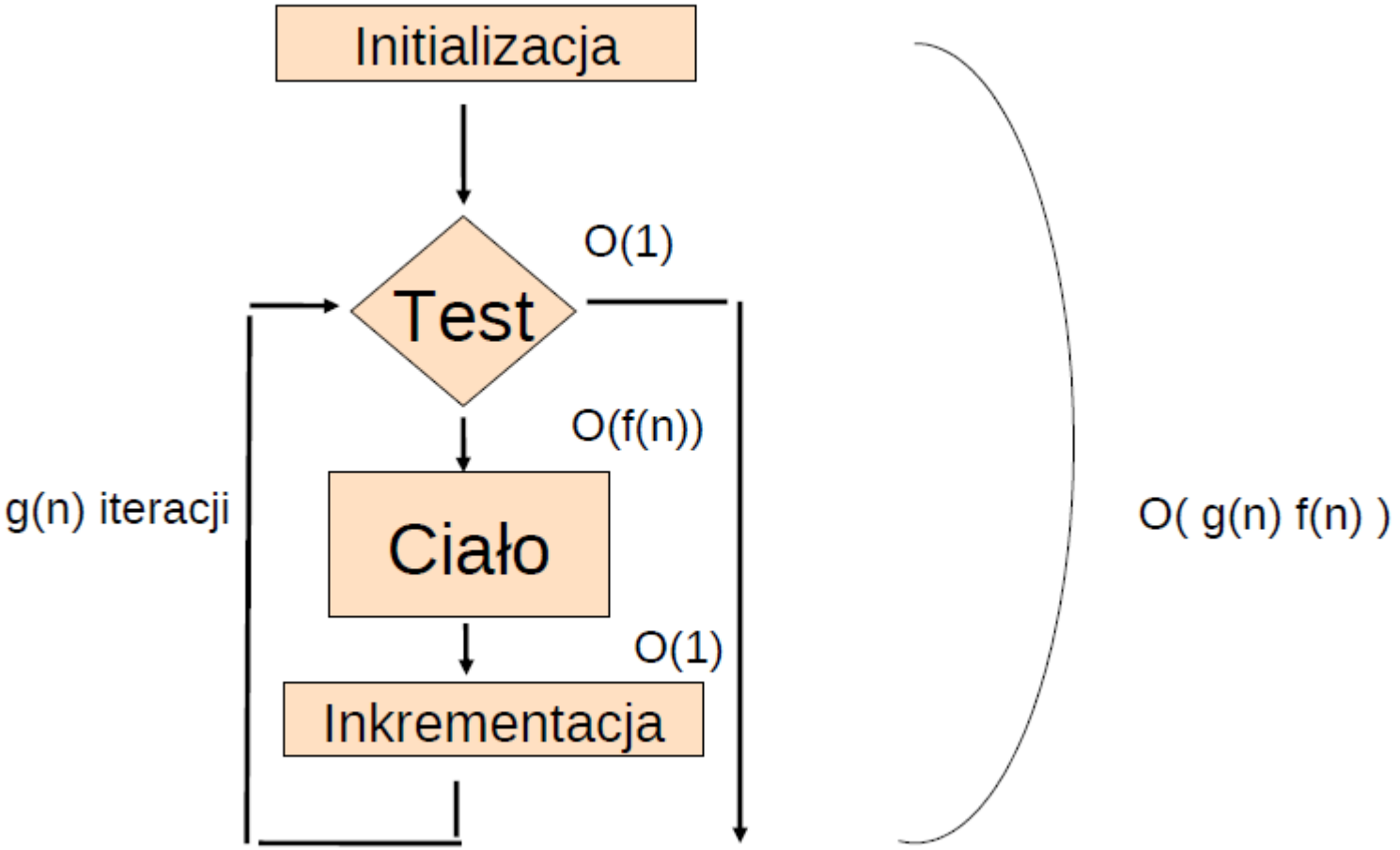
Czas działania pętli „for”

- **Przykład 1: Prosta pętla**

```
for (i=sum=0; i<n; i++) sum+=a[i];
```

- **Powyższa pętla powtarza się n razy, podczas każdego jej przebiegu realizuje dwa przypisania:**
  - **aktualizujące zmienną „sum”**
  - **zmianę wartości zmiennej „i”**
- **Mamy zatem 2n przypisań podczas całego wykonania pętli.**
- **Złożoność asymptotyczna algorytmu jest O(n).**

Czas działania pętli „for”



Czas działania pętli „for”

- **Przykład 2:** **Pętla zagnieżdżona**

```
for  (i=0; i<n; i++)    {
        for (j=1, sum=a[0]; j<=i; j++)
                sum+=a[j];  }
```

- **Pętla zewnętrzna powtarza się n razy, a w każdej jej iteracji wykonuje się wewnętrzna pętla oraz instrukcja przypisania wartości zmiennym „i”, „ j”, „sum”.**
- **Pętla wewnętrzna wykonuje się „i” razy dla każdego i ϵ {1, …,n-1}, a na każdą iteracje przypadają dwa przypisania: jedno dla „sum”, jedno dla „j”.**
- **Mamy zatem: $1+3n+2(1+2+...+n-1) = 1+3n+n(n-1) = O(n)+O(n^2) = O(n^2)$ przypisań wykonywanych w całym programie.**
- **Złożoność asymptotyczna algorytmu jest $O(n^2)$. Pętle zagnieżdżone mają zwykle większą złożoność niż pojedyncze, jednak nie musi tak być zawsze.**

Czas działania pętli „for”

- **Przykład 3: Znajdź najdłuższą podtablicę zawierającą liczby uporządkowane rosnąco.**

```
for (i=0; len=1; i<n-1; i++) {
        for (i1=i2=k=i; k<n-1 && a[k]<a[k+1]; k++,i2++);
            if(len < i2-i1+1)  len=i2-i1+1; }
```

- **Jeśli liczby w tablicy są uporządkowane malejąco, to pętla zewnętrzna wykonuje się n-1 razy, a w każdym jej przebiegu pętla wewnętrzna wykona się tylko 1-raz.**
  Złożoność asymptotyczna algorytmu jest więc O(n).
- **Jeśli liczby w tablicy są uporządkowane rosnąco, to pętla zewnętrzna wykonuje się n-1 razy, a w każdym jej przebiegu pętla wewnętrzna wykona się i-razy dla i ϵ {1,...,n-1}.**
  Złożoność asymptotyczna algorytmu jest więc $O(n^2)$.

Czas działania pętli „for”

- Z reguły dane nie są uporządkowane i ocena złożoności algorytmu jest rzeczą niełatwą ale bardzo istotną.
- Staramy się wyznaczy złożoność
  - w **„przypadku optymistycznym”,**
  - w **„przypadku pesymistycznym”**
  - oraz w **„przypadku średnim”**
- Często posługujemy się przybliżeniami opartymi o notacje ***„wielkie O, Ω i Θ”*** .

Czas działania instrukcji warunkowych

Instrukcje warunkową **if-else** zapisuje się w postaci:

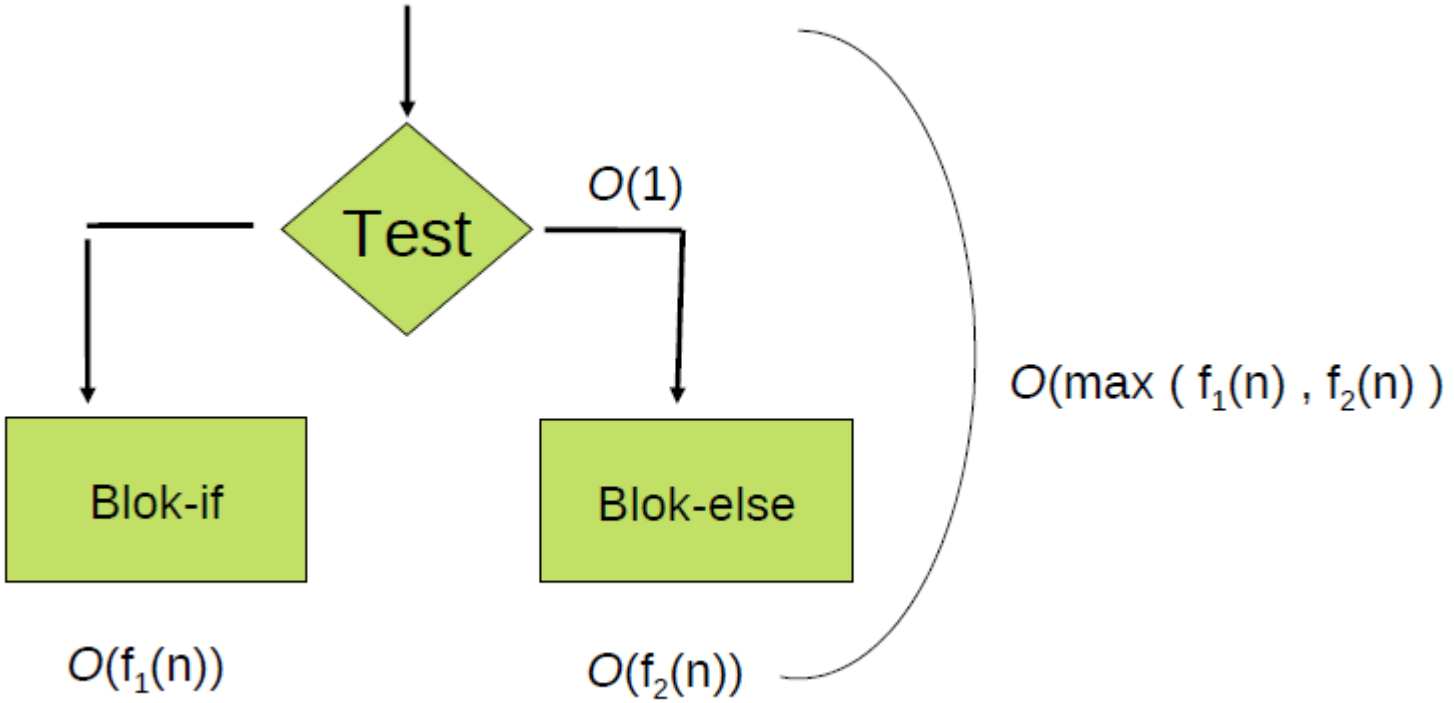
```
if (<warunek>)
        <blok-if>
else
        <blok-else>
```

Gdzie

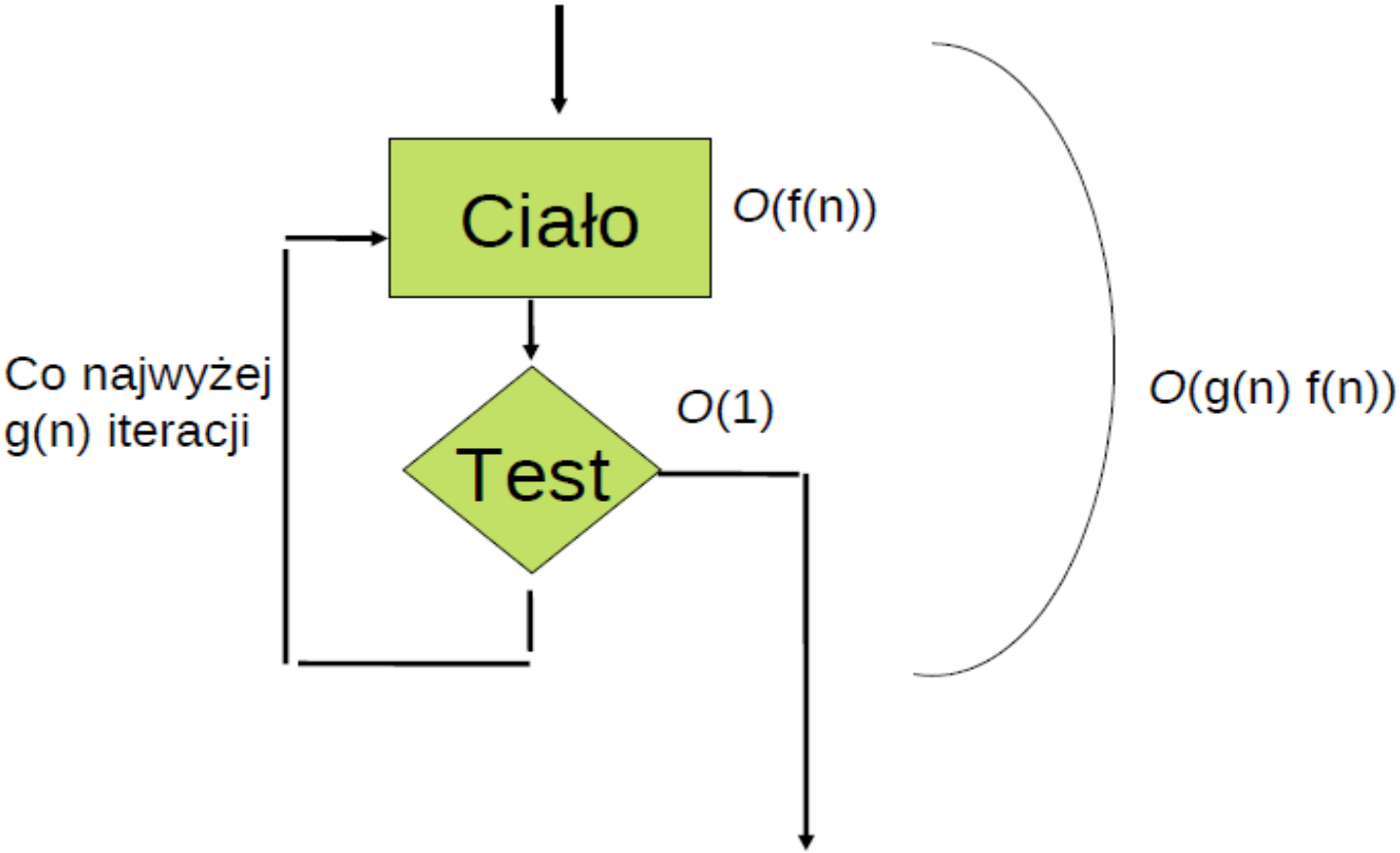
- **<warunek>** jest wyrażeniem które trzeba obliczyć. Warunek niezależnie od tego jak skomplikowany wymaga wykonania stałej liczby operacji (więc czasu **O(1)**) chyba ze zawiera wywołanie funkcji, .
- **<blok-if>** zawiera instrukcje wykonywane tylko w przypadku gdy warunek jest prawdziwy, czas działania **f(n)**.
- **<blok-else>** wykonywany jest tylko w przypadku gdy warunek jest fałszywy, czas działania **g(n)**.

Czas działania instrukcji warunkowej należy zapisać jako ***O(max (f(n), g(n)))***

Czas działania instrukcji „if”



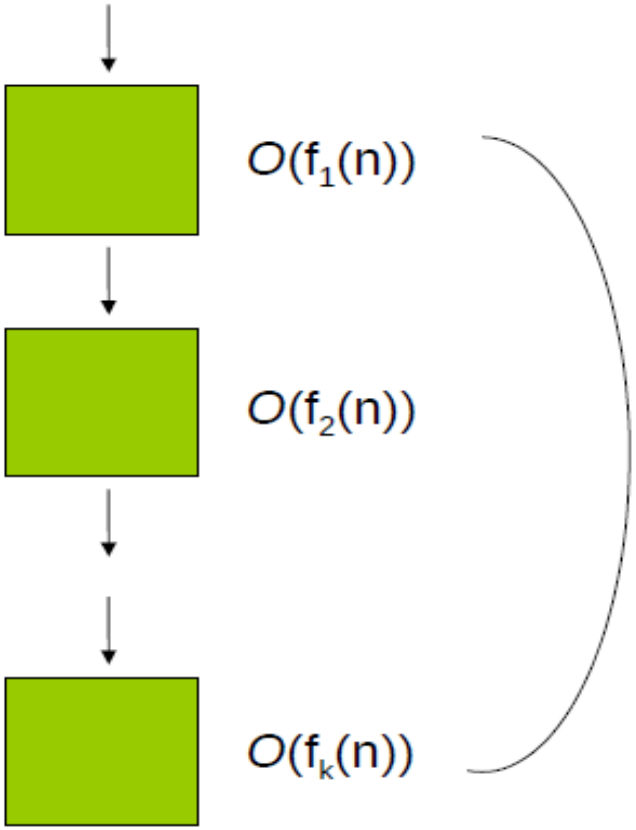
Czas działania instrukcji „do while”



Czas działania bloków

- Sekwencja instrukcji przypisań, odczytów i zapisów, z których każda wymaga czasu ***O(1)***, potrzebuje do swojego wykonania łącznego czasu ***O(1)***.
- Pojawiają się również instrukcje złożone, jak instrukcje warunkowe i pętle.
  - **Sekwencję** prostych i złożonych **instrukcji** nazywa się **blokiem**.
- Czas działania bloku obliczymy sumując górne ograniczenia czasów wykonania poszczególnych instrukcji, które należą do tego bloku.

Czas działania bloku instrukcji

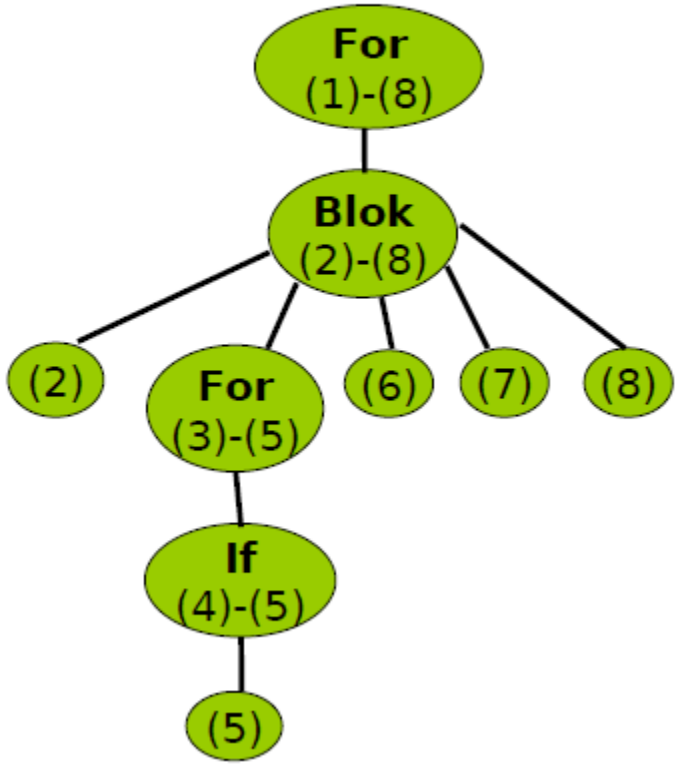


$O(f_1(n)+f_2(n)+...+ f_k(n))$
lub
$O($ największa $f_i(n))$

Przykład: „sortowanie przez wybieranie”


```
(1) for (i=0; i< n-1; i++ ) {
(2)   small = i;
(3)   for (j=i+1; j<n; j++ )
(4)         if( A[j] < A[small] )
(5)         small =j;
(6)   temp = A[small];
(7)   A[small] = A[i];
(8)   A[i] = temp; }
```

Drzewo reprezentujące grupowanie instrukcji

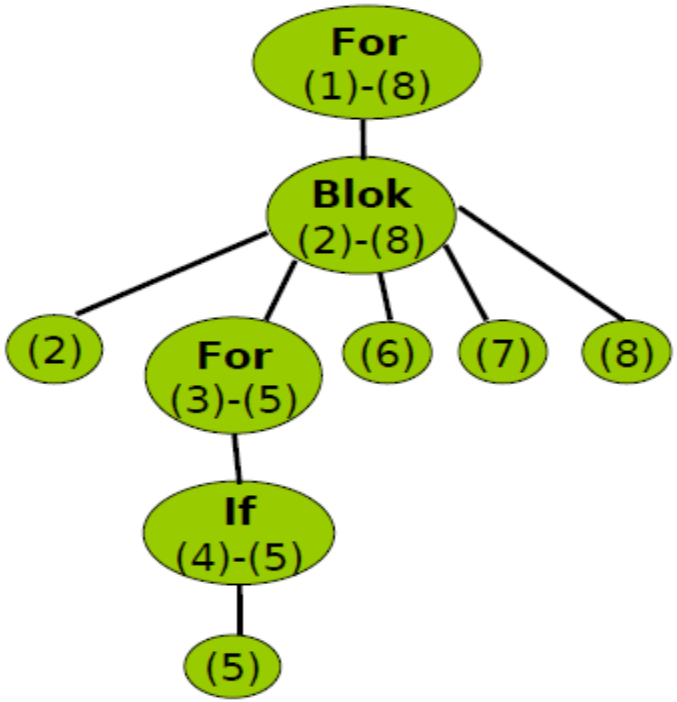


Przykład: „sortowanie przez wybieranie”

- Rozpoczynamy analizę „od liścia do korzenia”
  - Każda instrukcja przypisania (liście) wymaga czasu $O(1)$
  - Instrukcja „if” (4-5) wymaga czasu $O(1)$
  - Instrukcja ‘for” (3)-(5) wymaga czasu $O(n-i-1)$ oraz i<n
  - Instrukcja „for” (2)-(8) może być dalej ograniczona przez $O(n-1)$
  - Instrukcja „for” (1)-(8) może być ograniczona przez $O(n\,(n-1))$

Odrzucając wyraz mniej znaczący otrzymujemy oszacowanie czasu działania jako $O(n^2)$.

Drzewo reprezentujące grupowanie instrukcji

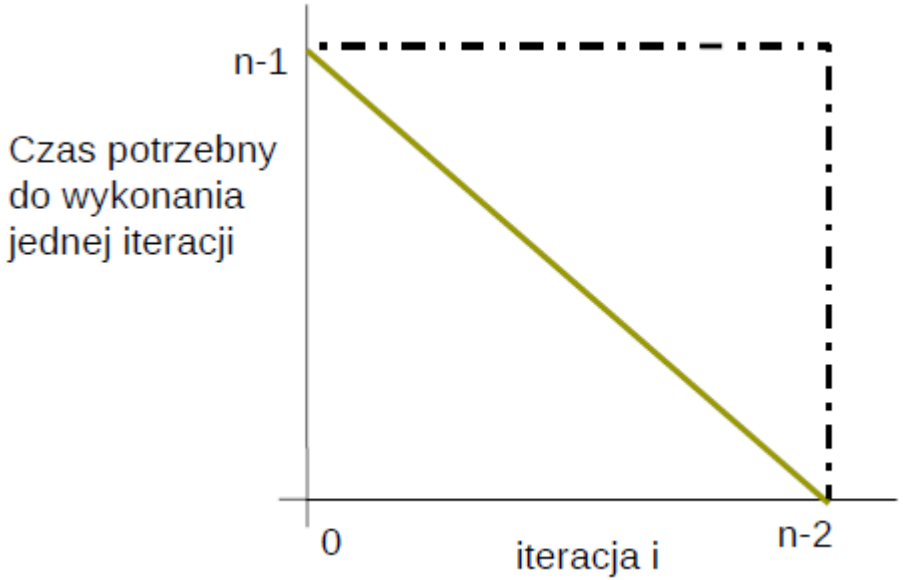


Przybliżone lub precyzyjne ograniczenie

- Dotychczas rozważaliśmy szacowanie czasu działania pętli używając ujednoliconego górnego ograniczenia, mającego zastosowanie w każdej iteracji pętli.
- Dla sortowania przez wybieranie, takie przybliżone ograniczenie prowadziło do szacowania czasu wykonania pętli $O(n^2)$.
- Można jednak dokonać bardziej szczegółowej analizy pętli i dokonać sumowania górnych ograniczeń poszczególnych iteracji. Część działania pętli z wartością **i** zmiennej indeksowej i wynosi **O(n-i-1)**, gdzie **i** przyjmuje wartości od **0** do **n-2**.
- Górne ograniczenie czasu niezbędne do wykonania wszystkich iteracji wynosi:

$$O\left(\sum_{n=0}^{n-2}(n-i-1)\right) = O\left(n(n-1)/2\right)$$

Przybliżone lub precyzyjne ograniczenie



- Górne ograniczenie czasu niezbędne do wykonania wszystkich iteracji wynosi:

$$O\left(\sum_{i=0}^{n-2}(n-i-1)\right) = O\left(n(n-1)/2\right)$$

Nie przejmuj się efektywnością algorytmu…

- **Nie przejmuj się efektywnością algorytmu… wystarczy poczekać kilka lat.**
- Taki pogląd funkcjonuje czasem w środowisku programistów, **nie określono przecież granicy rozwoju mocy obliczeniowych komputerów**.
- Nie należy się jednak z nim zgadzać w ogólności.
- Należy zdecydowanie przeciwstawiać się przekonaniu o tym, że ulepszenia sprzętowe uczynią pracę nad efektywnymi algorytmami zbyteczną.

Nie przejmuj się efektywnością algorytmu…

- Istnieją problemy których rozwiązanie za pomocą zasobów komputerowych **jest teoretycznie możliwe**, ale **praktycznie przekracza możliwości istniejących technologii**. Przykładem takiego problemu jest rozumienie języka naturalnego, przetwarzanie obrazów (do pewnego stopnia oczywiście) czy "inteligentna" komunikacja pomiędzy komputerami a ludźmi na rozmaitych poziomach.
- Kiedy pewne problemy stają się "proste"… Nowa grupa wyzwań, które na razie można sobie tylko próbować wyobrażać, **wytyczy nowe granice możliwości wykorzystania komputerów**.

Pytania do wykładu

1. **Podaj definicję notacji: „wielkie O”, Ω, Θ.**
2. **Podaj własności notacji „wielkie O”.**
3. **Wymień znane Ci klasy złożoności algorytmu.**
4. **Co to są funkcje niewspółmierne? Podaj przykład.**
5. **Co to jest złożoność obliczeniowa średnia? Uzasadnij pojęcie dla dowolnie wybranego algorytmu.**
6. **Co to jest złożoność obliczeniowa asymptotyczna? Uzasadnij pojęcie dla dowolnie wybranego algorytmu.**

Wykład 2c: Algorytmy i ich schematy blokowe

Algorytmy i ich schematy blokowe

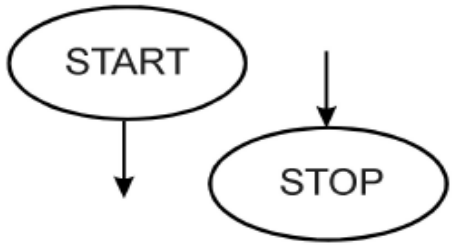
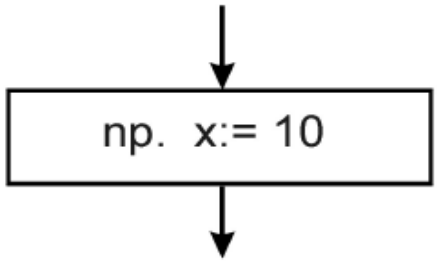
- **Proste algorytmy iteracyjne**
- **Algorytmy z wykorzystaniem rekurencji**
- **Algorytmy sortujące**

Wykład na podstawie skryptu:
D. Nyk, „Algorytmy w przykladach”
http://informatyka.2ap.pl/ftp/3d/algorytmy/podręcznik_algorytmy.pdf

Schemat blokowy

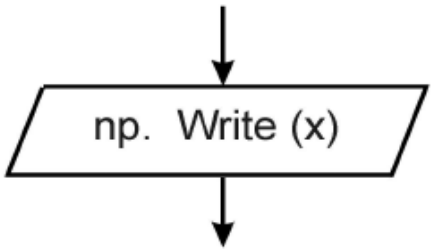
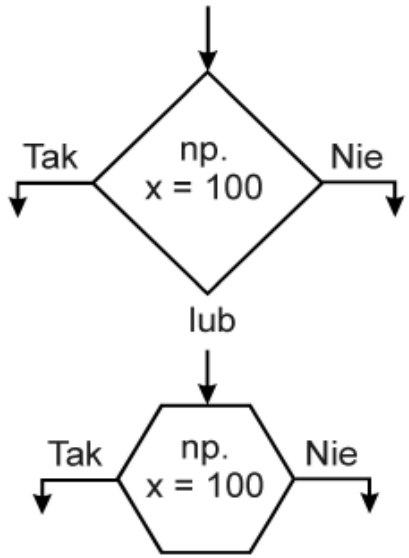
- **Przedstawia algorytm w postaci symboli graficznych, podając szczegółowo wszystkie operacje arytmetyczne, logiczne, przesyłania, pomocnicze wraz z kolejnością ich wykonywania**
- **Składa się z wielu elementów z których podstawowy jest blok**
- **Poniżej przedstawione typowe podstawowe bloki programów, istnieją oczywiście jeszcze inne.**

Schemat blokowy

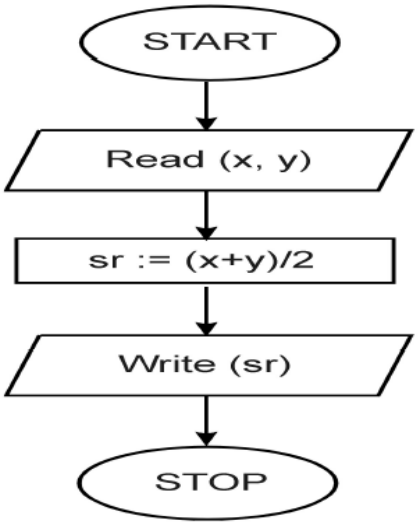
Wygląd bloku	Opis
	**Bloki graniczne** – początek i koniec algorytmu. Mają kształt owalu. Z bloku Start wychodzi tylko jedno połączenie; każdy schemat blokowy musi mieć dokładnie jeden blok START. Każdy schemat blokowy musi mieć co najmniej jeden blok STOP.
	**Łącznik** pomiędzy blokami – określa kierunek przepływu danych lub kolejność wykonywanych działań (ścieżka sterująca)
	**Blok kolekcyjny** – łączy kilka różnych dróg algorytmu
	**Blok operacyjny** – zawiera operację lub grupę operacji, w których wyniku ulega zmianie wartość zmiennej (tu : nadanie zmiennej x wartości 10). Bloki operacyjne mają kształt prostokąta , wchodzi do niego jedno połączenie i wychodzi też jedno.

y

Schemat blokowy

	**Blok wejścia / wyjścia** – blok odpowiedzialny za wykonanie operacji wprowadzania i wyprowadzania danych, wyników, komunikatów. Ma kształt równoległoboku, wchodzi i wychodzi z niego jedno połączenie.
	**Blok decyzyjny** – określa wybór jednej z dwóch możliwych dróg działania. Ma kształt rombu lub sześciokąta. Wchodzi do niego jedno połączenie, a wychodzą dwa : TAK – gdy warunek wpisany wewnątrz jest spełniony oraz NIE – gdy warunek wpisany wewnątrz nie jest spełniony. Wybór kształtu bloku zależy od nas.

Schemat blokowy i specyfikacja programu



Algorytm liczenia średniej

Specyfikacją problemu algorytmicznego nazywamy dokładny opis problemu algorytmicznego, który ma zostać rozwiązany oraz podanie informacji o danych wejściowych i wyjściowych.
Czyli przed naszym algorytmem powinien znaleźć się dodatkowy zapis :

Problem algorytmiczny : obliczenie średniej arytmetycznej dwóch licz rzeczywistych

Dane wejściowe : $x , y \in R$

Dane wyjściowe : $sr \in R$ – średnia liczb x i y

Operandy i operatory

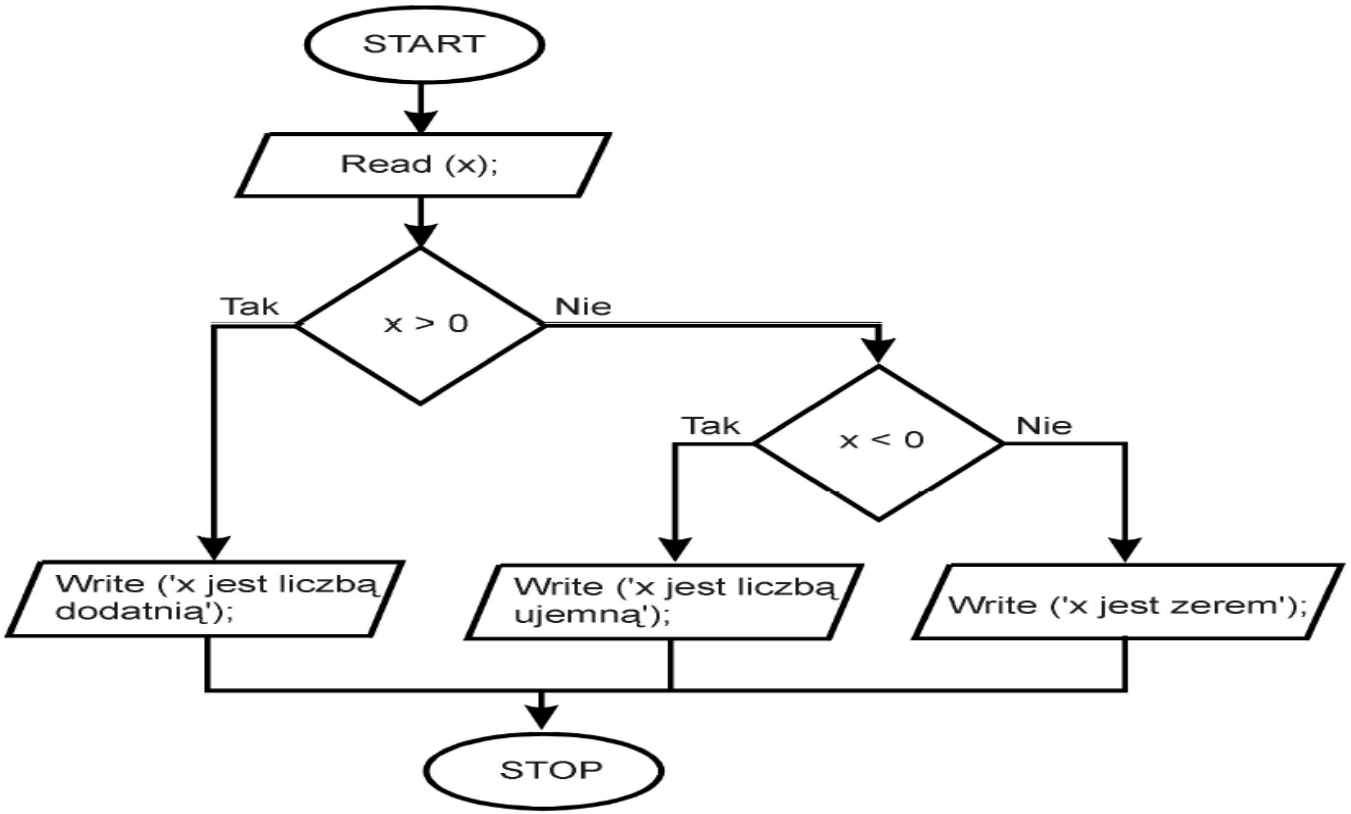
- **Stałe i zmienne łączymy operatorami aby otrzymać wyrażenie. Stałe i zmienne nazywamy operandami.**

Operatory arytmetyczne		Operatory relacji	
Symbol	Znaczenie	Symbol	Znaczenie
+	dodawanie	=	równy
-	odejmowanie	>	większy
*	mnożenie	>=	większy lub równy
/	dzielenie	<	mniejszy
div	dzielenie całkowite (3div2 = 1)	<=	mniejszy lub równy
mod	reszta z dzielenia liczb całkowitych	<>	różny

Operatory logiczne

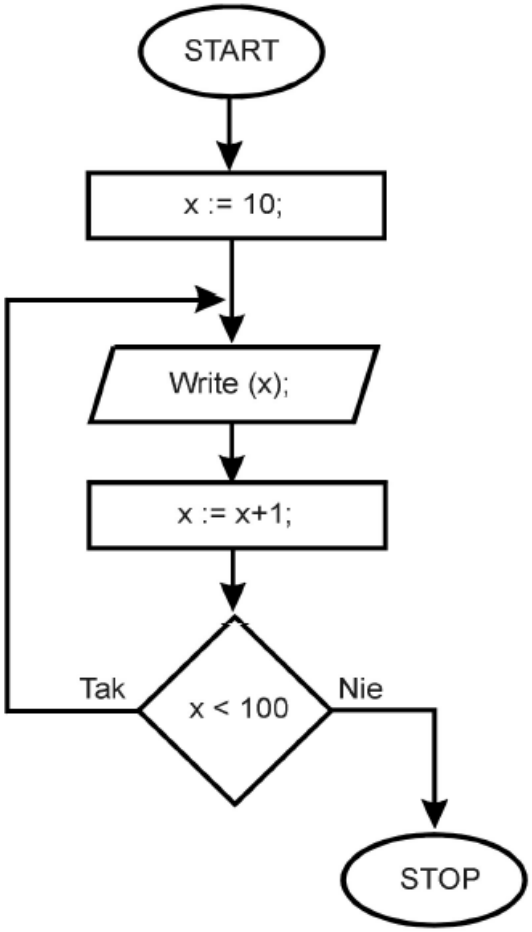
and	- koniunkcja (iloczyn zdań)	$\wedge$
or	- alternatywa (suma zdań)	$\vee$
not	- negacja (zaprzeczenie zdania)	$\sim$

Algorytmy z rozgałęzieniem

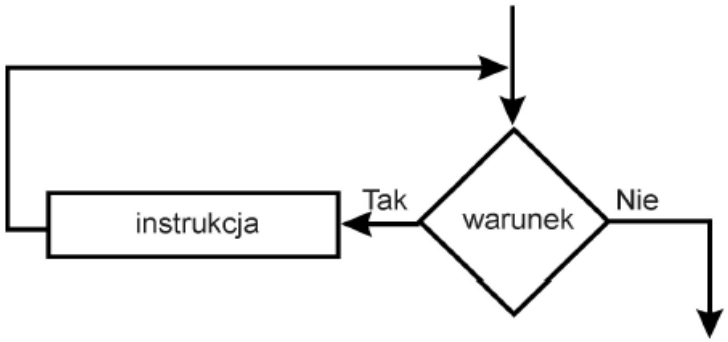


Warunek musi być tak określony, aby jego ocena prawdziwości była jednoznaczna.

Instrukcja iteracji

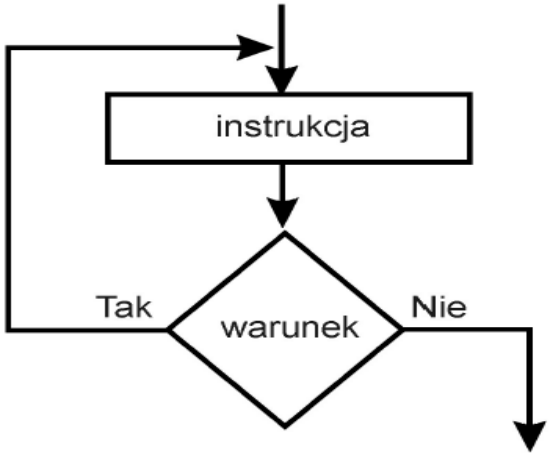


Poniżej przedstawiamy trzy podstawowe przypadki iteracji stosowanych w algorytmach.



Najpierw sprawdzany jest warunek, potem wykonywana jest instrukcja. (dopóki spełniony jest warunek wykonuj instrukcję

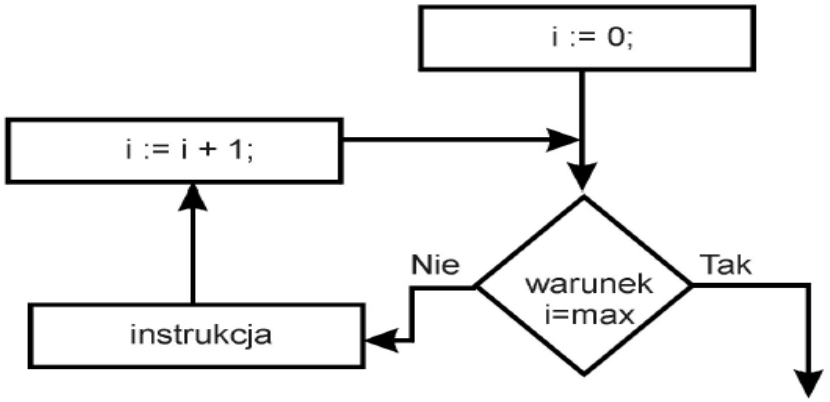
Instrukcja iteracji



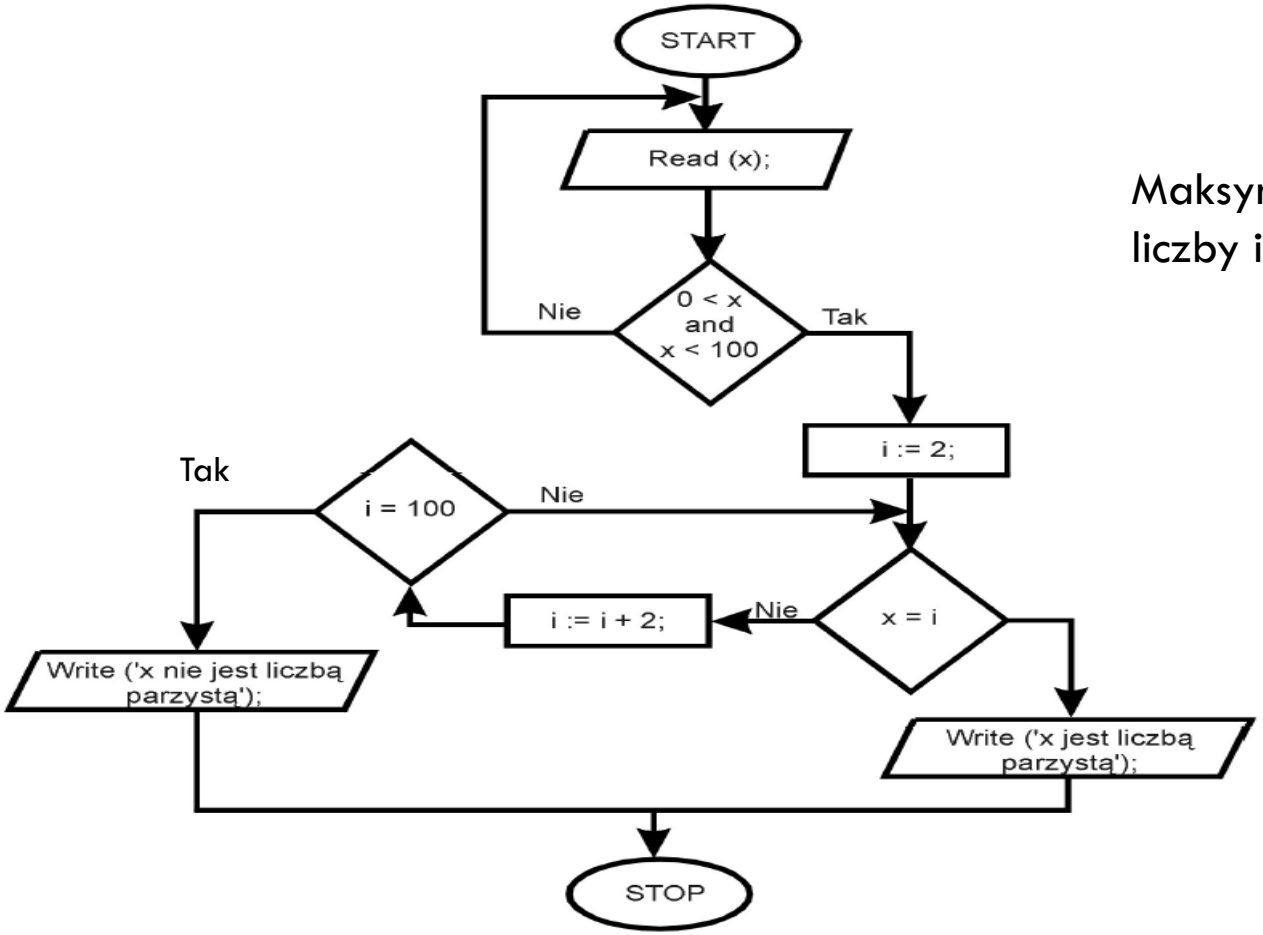
Najpierw wykonywana jest instrukcja , a potem sprawdzany jest warunek (wykonuj instrukcję dopóki spełniony jest warunek

Instrukcja wykonywana jest z góry ustaloną (max) ilość razy
For zmienna := wyrażenie

W pętli tej wartość licznika „i" zwiększana jest o 1 po każdej wykonywanej instrukcji czyli jest **inkrementowana**

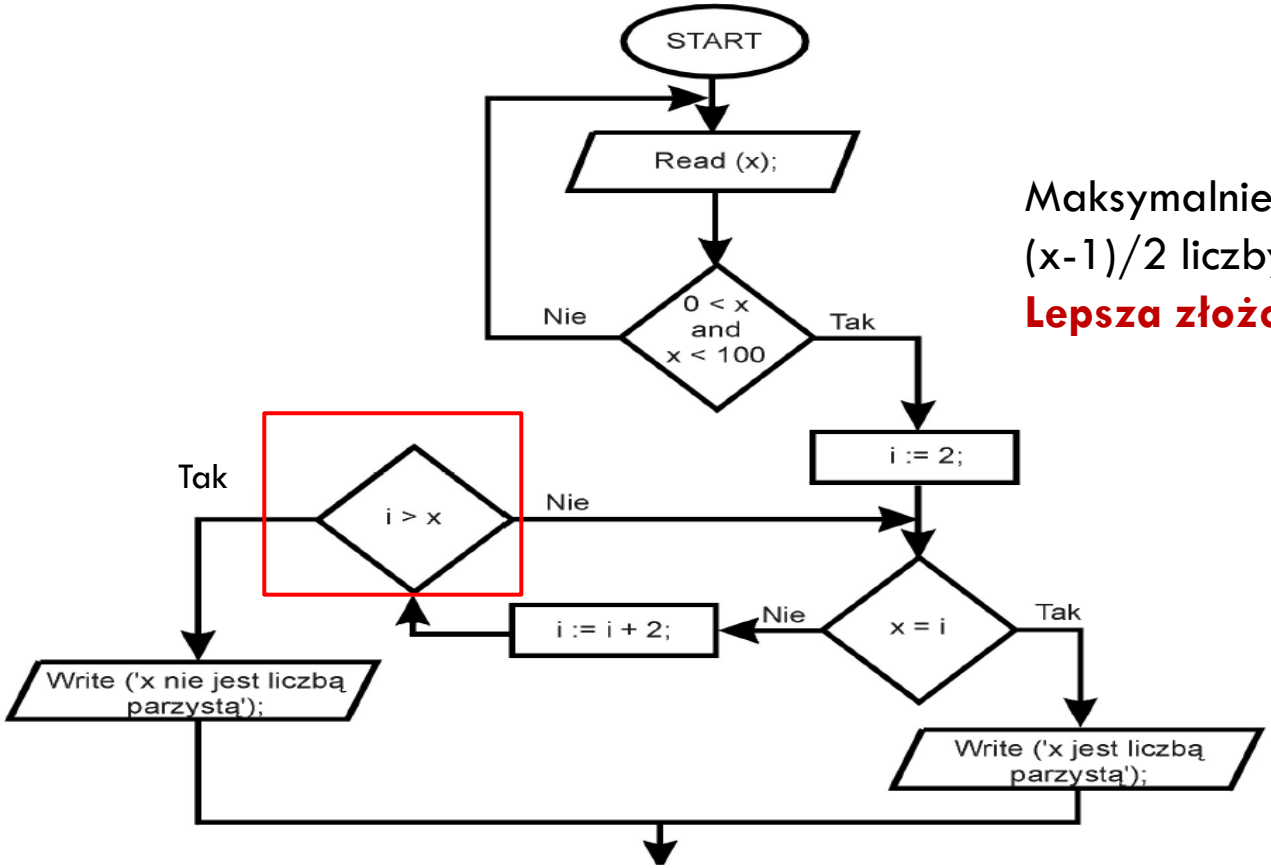


Badanie parzystości: algorytm 1



Maksymalnie 49 porównań liczby i z liczba 100

Badanie parzystości: algorytm 2



Maksymalnie porównań (x-1)/2 liczby i z liczba x.

Lepsza złożoność obliczeniowa

Algorytm Euklidesa

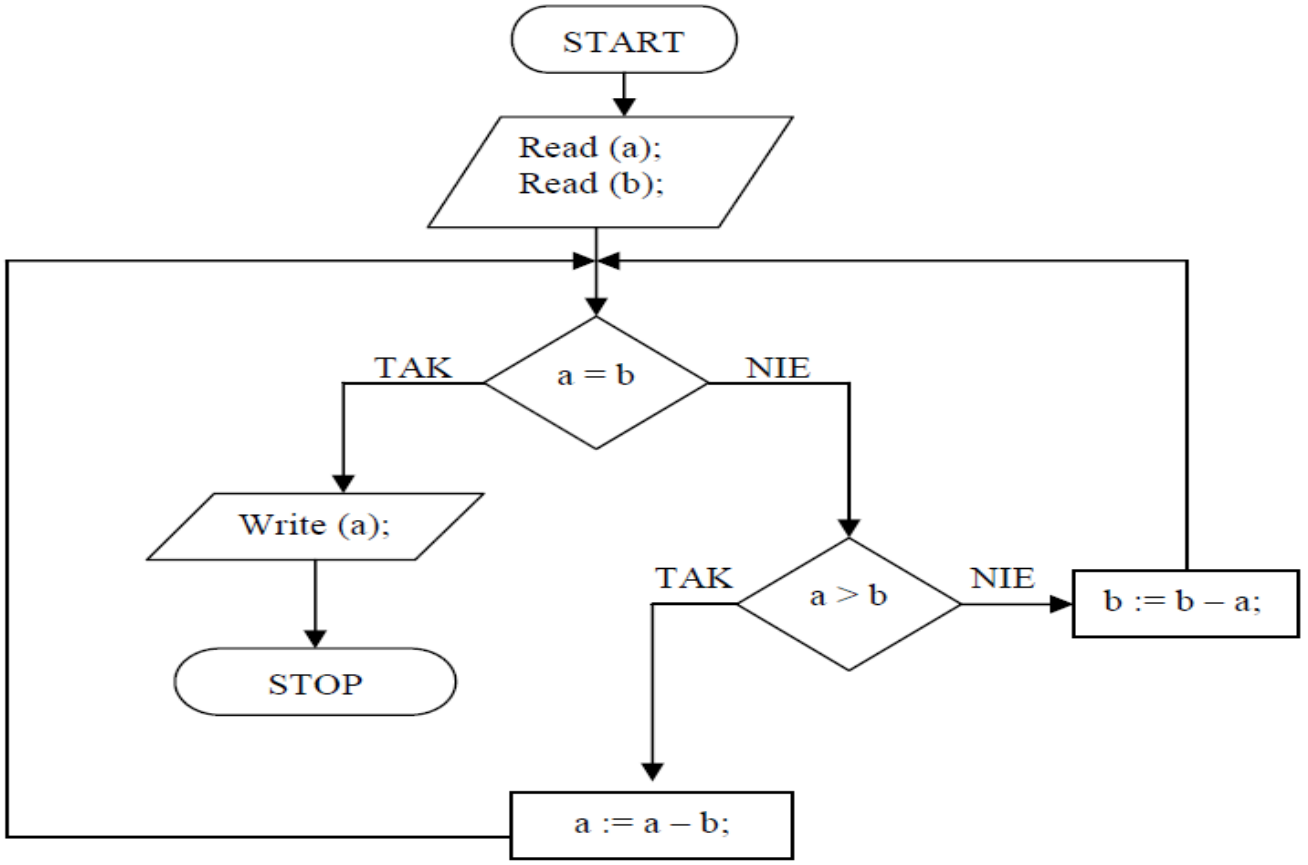
- Największy wspólny dzielnik dwóch liczb.

Pobieramy dwie liczby naturalne, od większej z nich odejmujemy mniejszą, a następnie większą liczbę zastępujemy różnicą. Postępujemy tak do momentu, gdy dwie liczby będą równe. Otrzymana liczba będzie NWD.
Przykład : a =12 , b = 20, ponieważ b > a to b = 20–12=8 , teraz a > b czyli a=12–8 =4, dalej b>a czyli b= 8–4 = 4 i a = 4 stąd NWD = 4

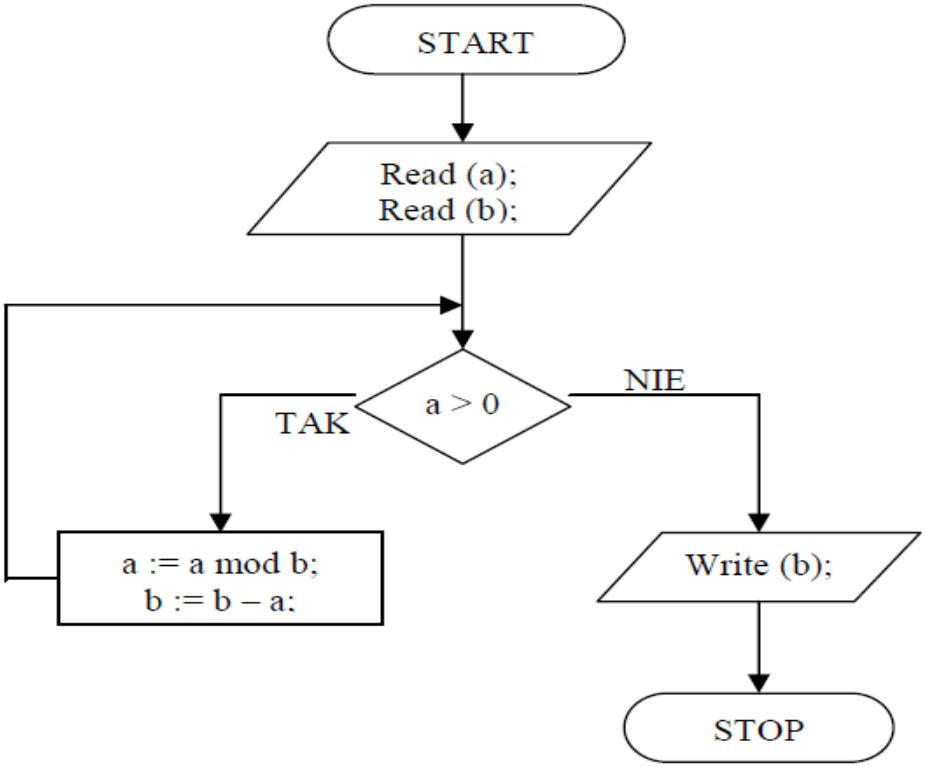
Równie często stosuje się modyfikację algorytmu NWD (a,b) = NWD(a mod b, b – (a mod b))

Będziemy iteracyjnie zmieniać wartości a i b aż do momentu, gdy a osiągnie wartość 0.
Przykład : a =12 , b = 20 , a = 12 mod 20 = 12 , b = 20 – 12 = 8 następnie a = 12 mod 8 = 4 , b = 8 – 4 = 4 , i następny krok a = 4 mod 4 = 0 czyli b = NWD = 4

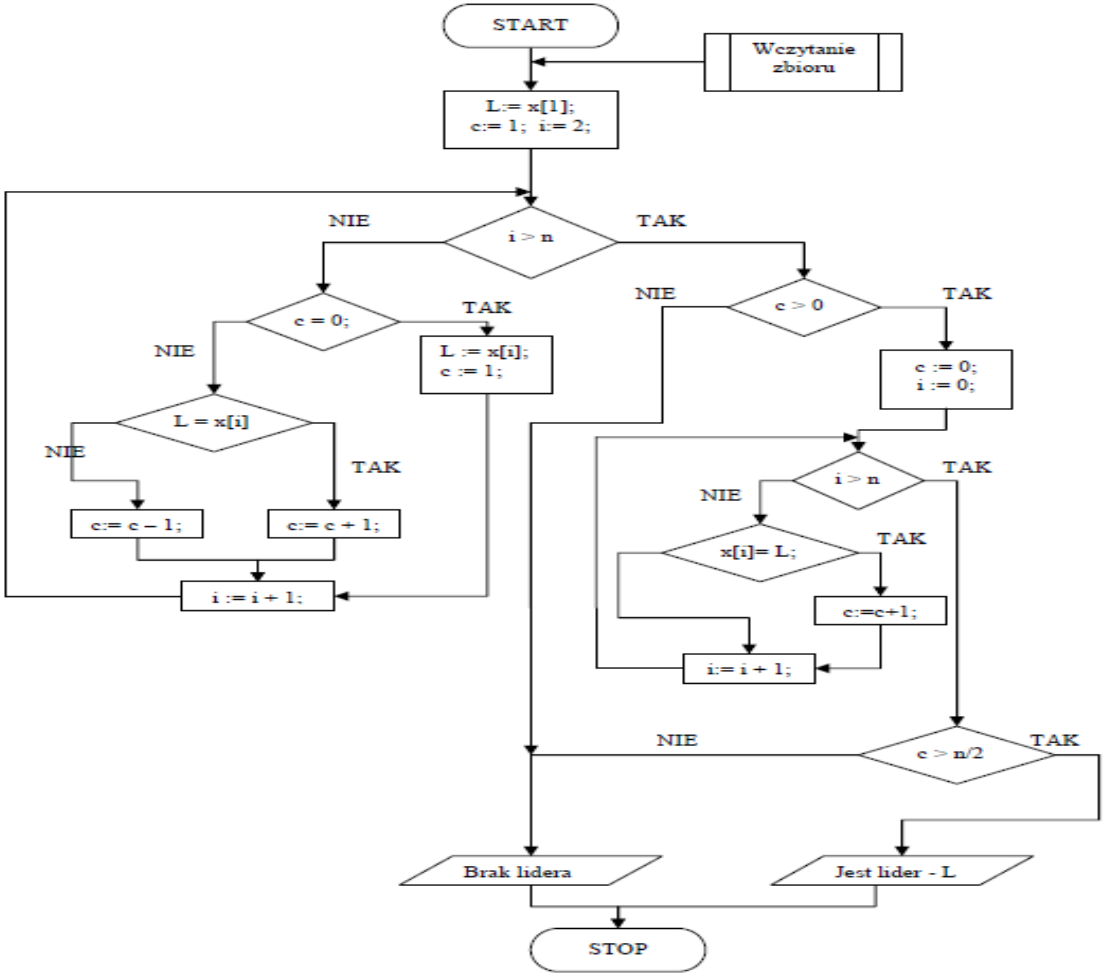
Algorytm Euklidesa: wersja 1



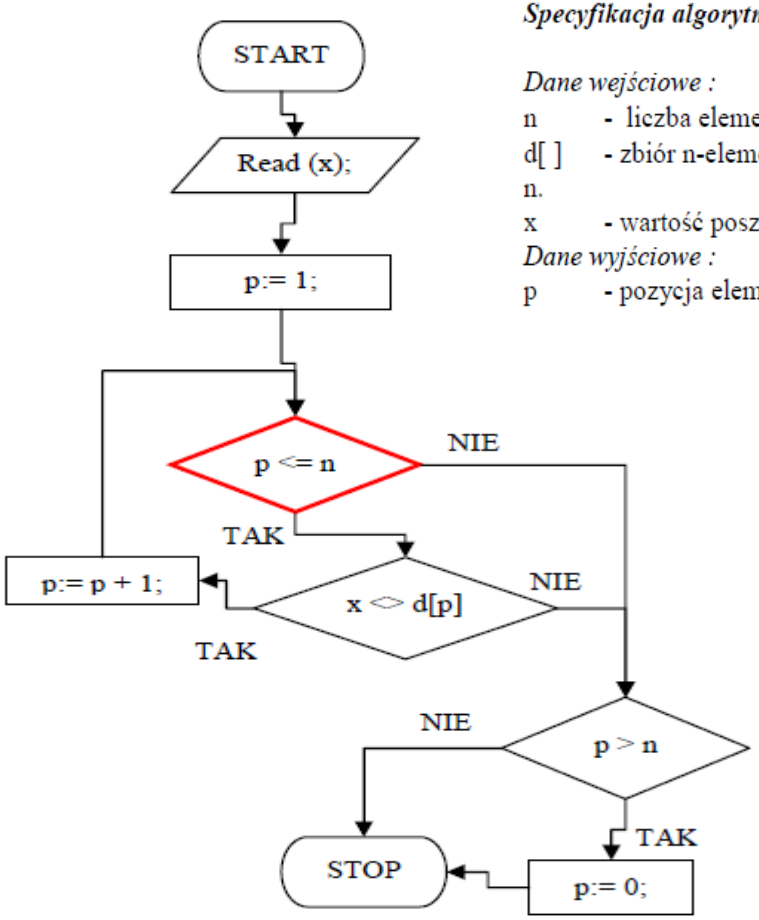
Algorytm Euklidesa: wersja 2



Poszukiwanie lidera zbioru



Przeszukiwanie sekwencyjne



Specyfikacja algorytmu :

Dane wejściowe :

n - liczba elementów w sortowanym zbiorze, $n \in N$

d[ ] - zbiór n-elementowy, który będzie przeszukiwany. Elementy zbioru mają indeksy od 1 do n.

x - wartość poszukiwana

Dane wyjściowe :

p - pozycja elementu x w zbiorze d[ ]. Jeśli $p = 0$, to element x w zbiorze nie występuje.

Warunek gwarantuje zakończenie pętli, możemy też wprowadzić wartownika

Złożoność obliczeniowa O(n)

Poszukiwanie najczęstszego elementu występującego w zbiorze

Specyfikacja problemu

Dane wejściowe :

n - liczba elementów w zbiorze wejściowym
d[ ] - zbiór wejściowy, w którym dokonujemy poszukiwań. Indeksy elementów rozpoczynają się od 1. Zbiór musi posiadać miejsce na dodatkowy element, który zostanie dopisany na końcu.

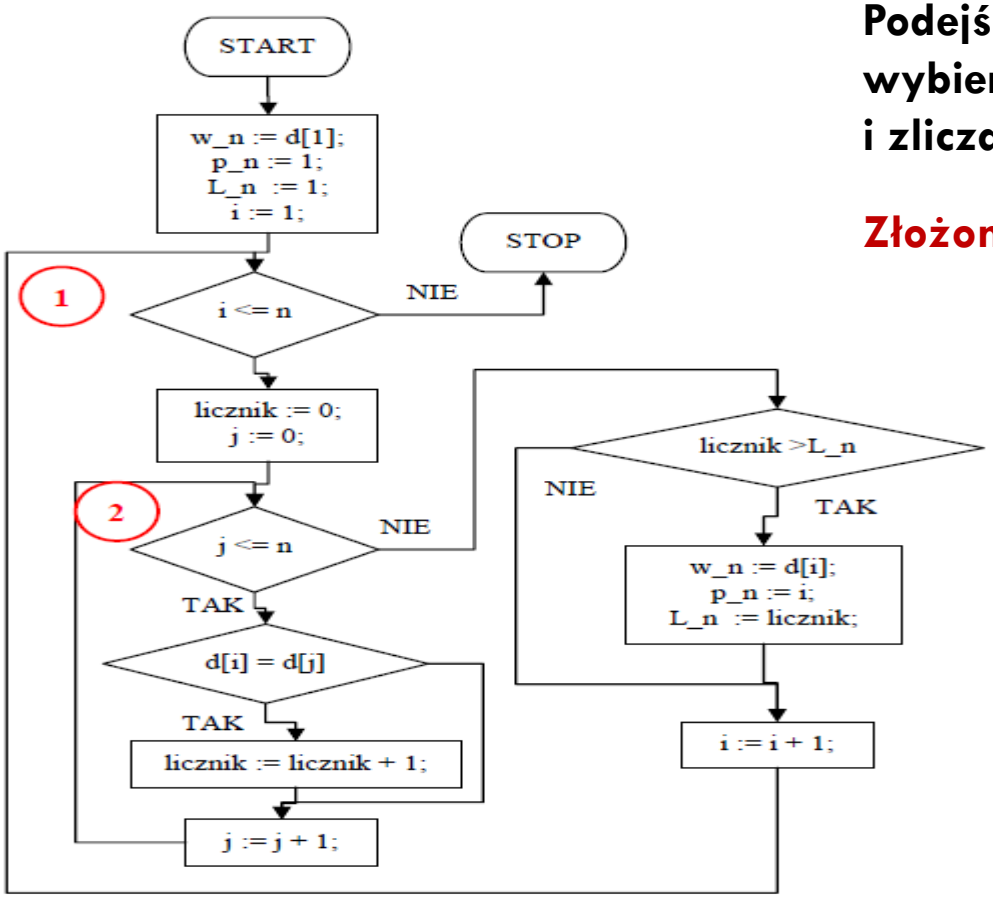
Dane wyjściowe :

w_n - wartość elementu powtarzającego się najczęściej
p_n - pierwsza pozycja elementu najczęstszego
L_n - liczba wystąpień najczęstszego elementu

Zmienne pomocnicze :

i,j - zmienne licznikowe pętli
licznik - licznik wystąpień elementu

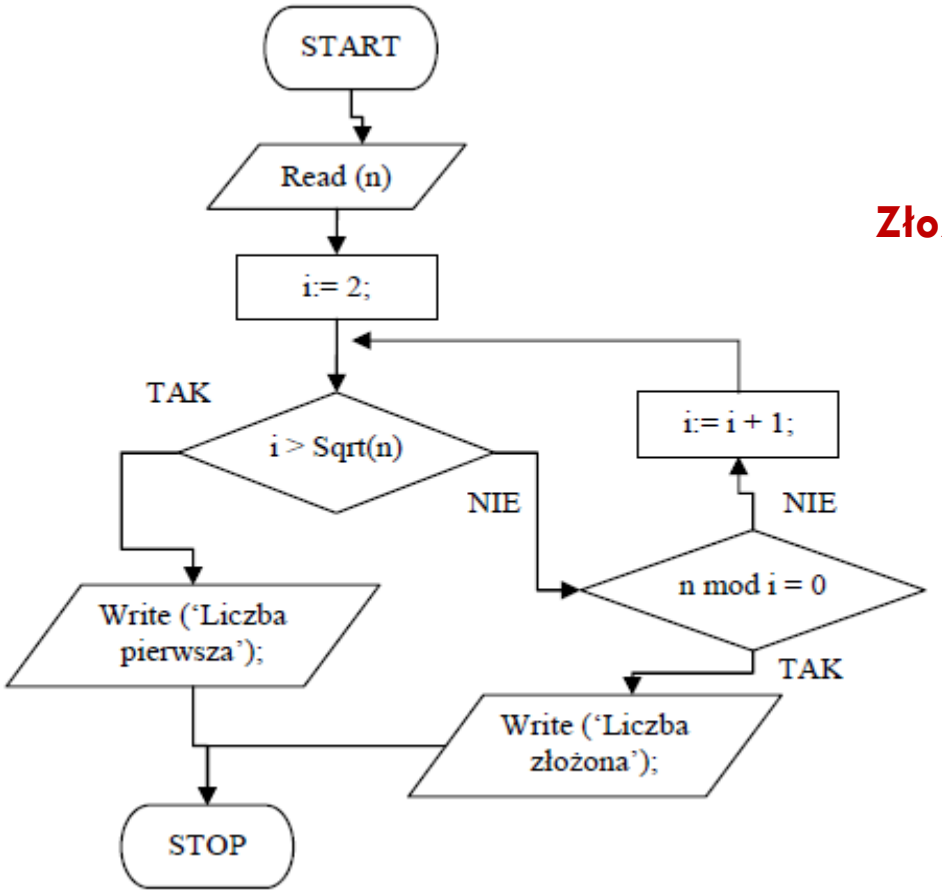
Poszukiwanie najczęstszego elementu występującego w zbiorze



Podejście bezpośrednie:
wybieramy kolejne elementy zbioru i zliczamy częstość ich występowania.

Złożoność obliczeniowa $O(n^2)$

Algorytm sprawdzający czy liczba jest liczba pierwszą.

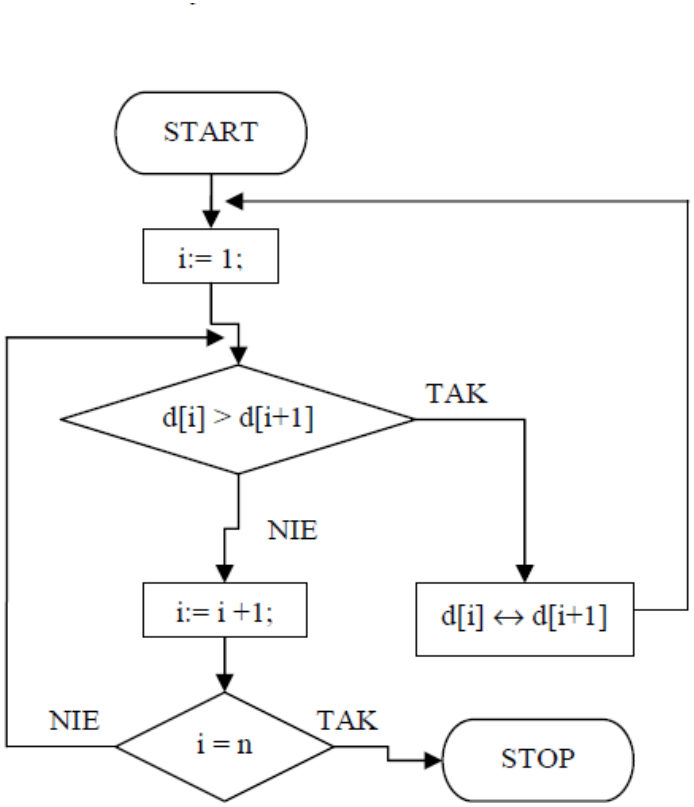


Złożoność obliczeniowa $O(n^{1/2})$

Złożoność obliczeniowa

Porównanie klas złożoności obliczeniowych		
Klasa złożoności obliczeniowej	**Nazwa klasy złożoności obliczeniowej**	**Cechy algorytmu**
$\Theta(1)$	stała	działa prawie natychmiast
$\Theta(\log n)$	logarytmiczna	bardzo szybki
$\Theta(n)$	liniowa	szybki
$\Theta(n \log n)$	liniowo-logarytmiczna	dosyć szybki
$\Theta(n^2)$	kwadratowa	wolny dla dużych n
$\Theta(n^3)$	sześcienna	wolny dla większych n
$\Theta(2^n)$, $\Theta(n!)$	wykładnicza	nierealizowalny dla większych n

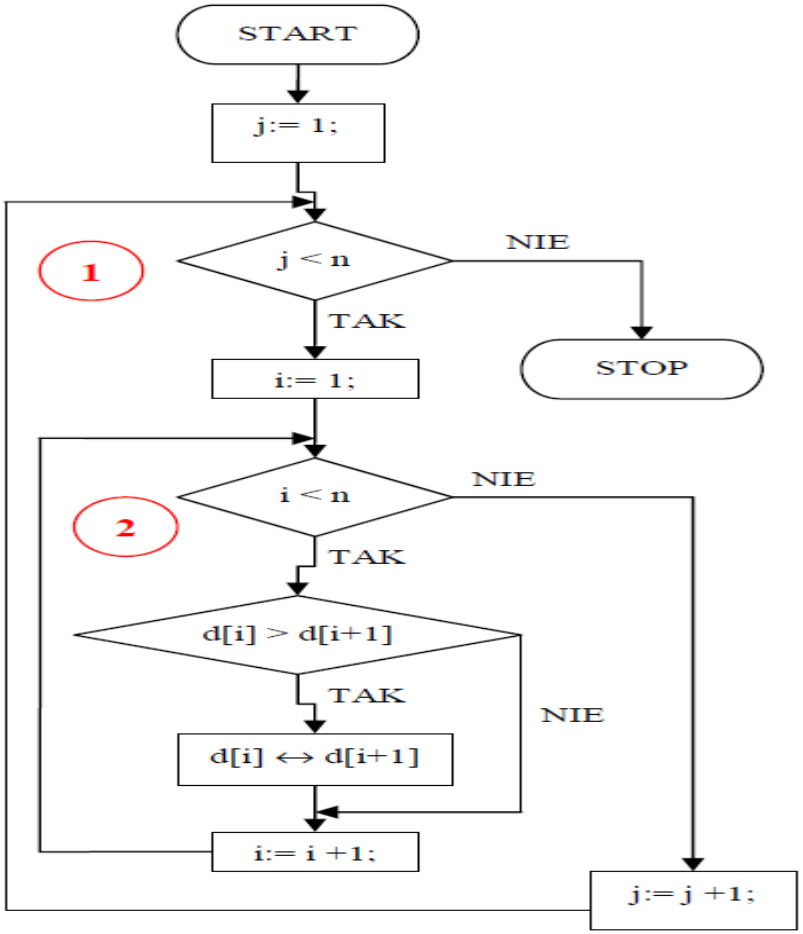
Sortowanie naiwne



Cechy Algorytmu Sortowania Naiwnego	
klasa złożoności obliczeniowej optymistyczna	$\Theta(n) - \Theta(n^2)$
klasa złożoności obliczeniowej typowa	$\Theta(n^3)$
klasa złożoności obliczeniowej pesymistyczna	$\Theta(n^3)$
Sortowanie w miejscu	TAK
Stabilność	TAK

Pesymistyczna:
dla zbiorów posortowanych odwrotnie
Optymistyczna:
dla zbiorów uporządkowanych z niedużą ilością elementów nie na swoich miejscach

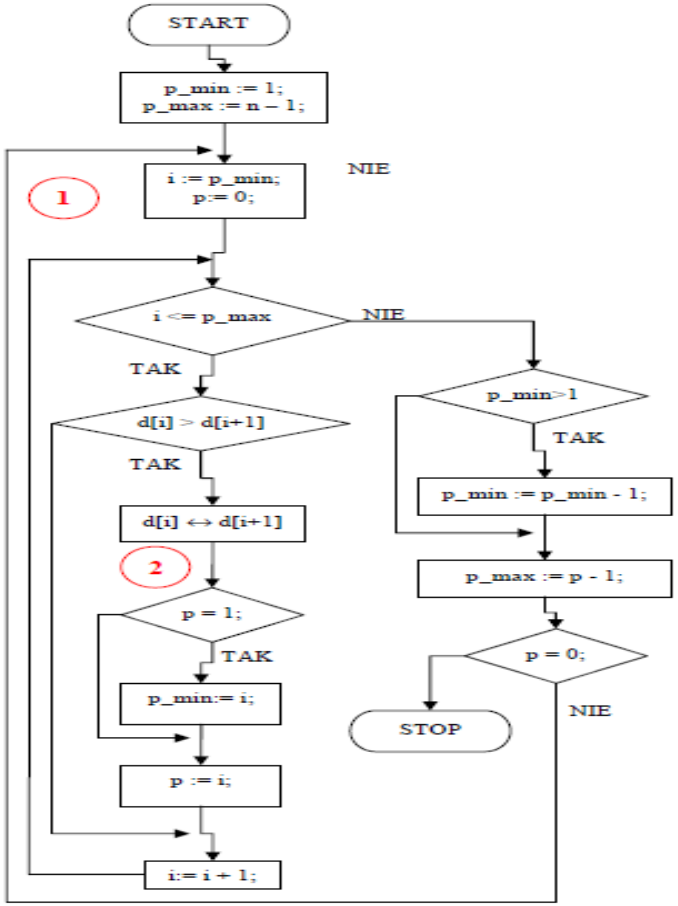
Sortowanie bąbelkowe



Sortowanie wykonywane jest w dwóch zagnieżdżonych pętlach. Pętla zewnętrzna nr 1 kontrolowana jest przez zmienną j. Wykonuje się ona n - 1 razy. Wewnątrz pętli nr 1 umieszczona jest pętla nr 2 sterowana przez zmienną i. Wykonuje się ona również n - 1 razy. W efekcie algorytm wykonuje w sumie: $T_1(n) = (n - 1)^2 = n^2 - 2n + 1$ obiegów pętli wewnętrznej, po których zakończeniu zbiór zostanie posortowany.

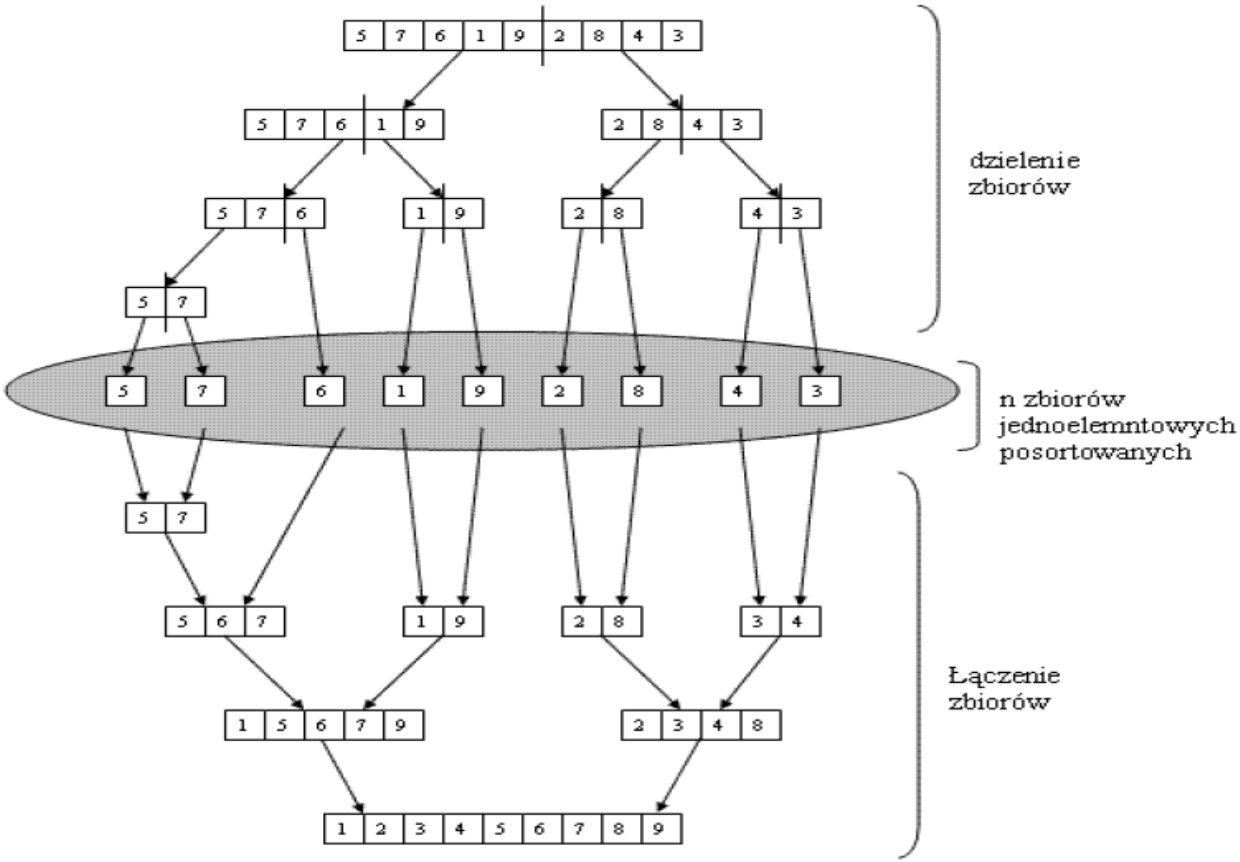
Cechy Algorytmu Sortowania Bąbelkowego wersja nr 1	
klasa złożoności obliczeniowej optymistyczna	$\Theta(n^2)$
klasa złożoności obliczeniowej typowa	$\Theta(n^2)$
klasa złożoności obliczeniowej pesymistyczna	$\Theta(n^2)$
Sortowanie w miejscu	TAK
Stabilność	TAK

Sortowanie bąbelkowe: modyfikacje



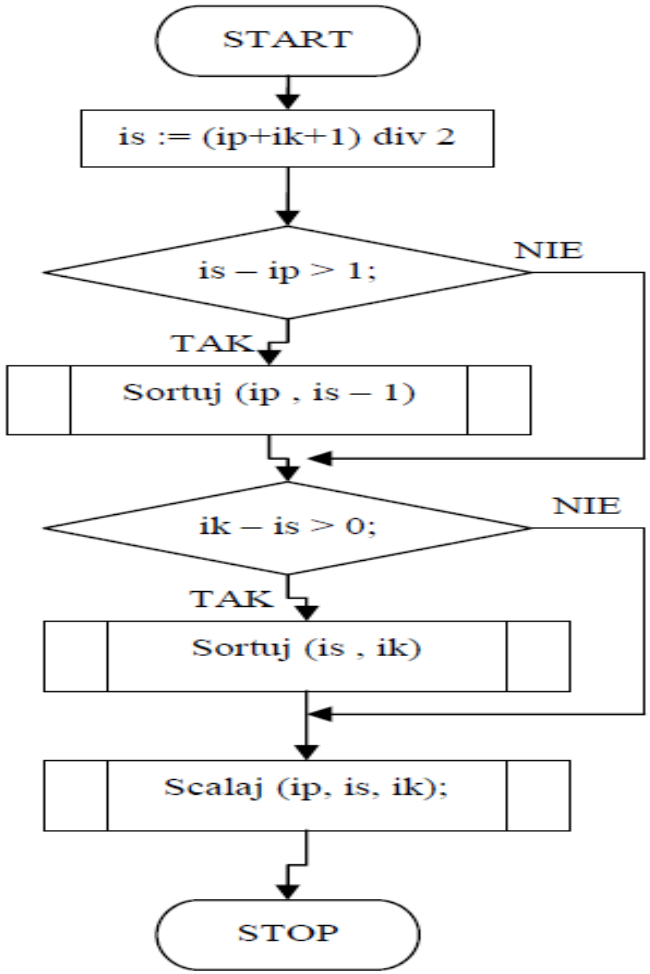
Cechy Algorytmu Sortowania Bąbelkowego wersja nr 2	
klasa złożoności obliczeniowej optymistyczna	$\Theta(n)$
klasa złożoności obliczeniowej typowa	$\Theta(n^2)$
klasa złożoności obliczeniowej pesymistyczna	$\Theta(n^2)$
Sortowanie w miejscu	TAK
Stabilność	TAK

Rekurencja: sortowanie



Przykład : sortujemy zbiór o postaci: { 6 5 4 1 3 7 9 2 }

Rekurencja: program sortuj



Złożoność obliczeniowa

O(n log(n))

Dane wejściowe :

d [ ]	- zbiór scalany
ip	- indeks pierwszego elementu w młodszym podzbiorze, ip ∈ N
ik	- indeks ostatniego elementu w starszym podzbiorze, ik ∈ N

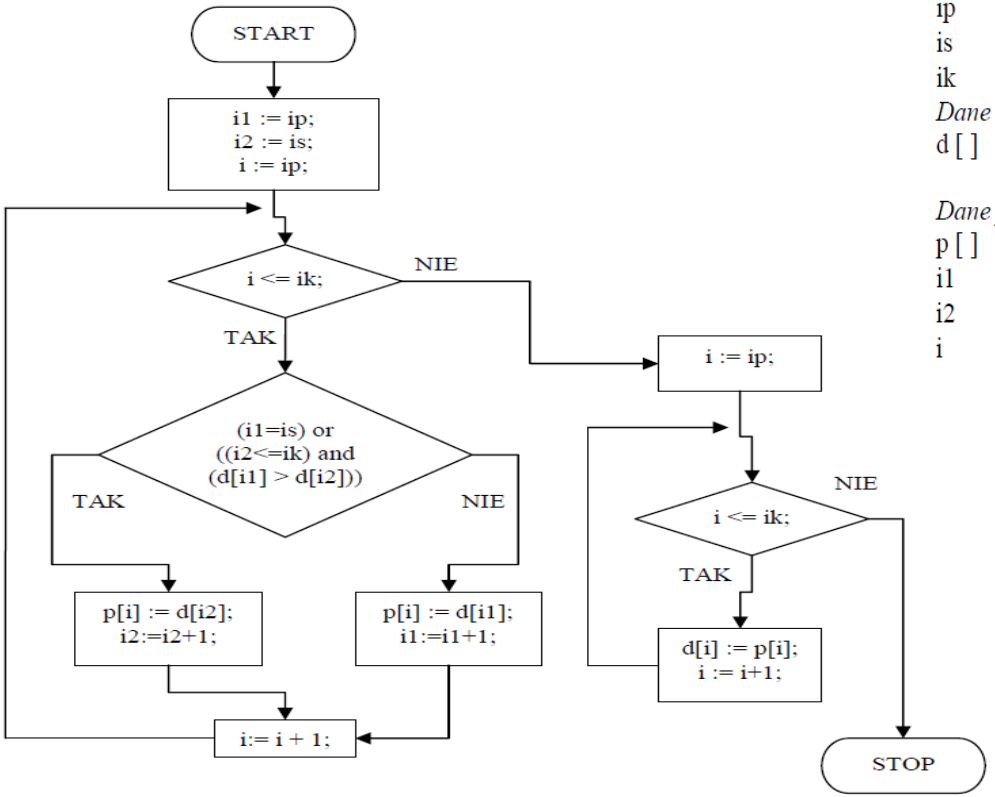
Dane wyjściowe :

d [ ]	- zbiór scalony

Dane pomocnicze :

is	- indeks pierwszego elementu w starszym podzbiorze, is ∈ N

Rekurencja: blok scalaj



Dane wejściowe :

d [ ]	- zbiór scalany
ip	- indeks pierwszego elementu w młodszym podzbiorze, ip ∈ N
is	- indeks pierwszego elementu w starszym podzbiorze, is ∈ N
ik	- indeks ostatniego elementu w starszym podzbiorze, ik ∈ N

Dane wyjściowe :

d [ ]	- zbiór scalony

Dane pomocnicze :

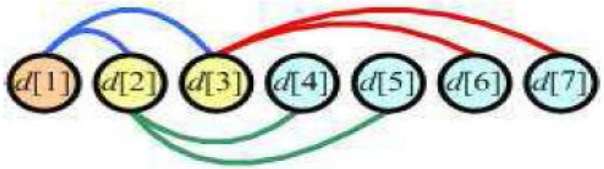
p [ ]	- zbiór pomocniczy zawierający tyle samo elementów ile zbiór d
i1	- indeks elementów w młodszej połówce zbioru d[ ], i1 ∈ N
i2	- indeks elementów w starszej połówce zbioru d[ ], i2 ∈ N
i	- indeks elementów w zbiorze p[ ], i ∈ N

Złożoność obliczeniowa O(n)

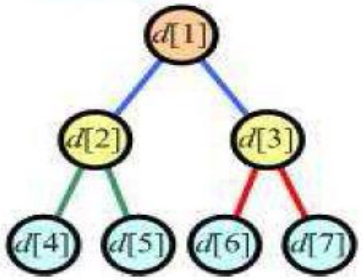
Sortowanie stogowe

Zastosujmy następujące odwzorowanie:

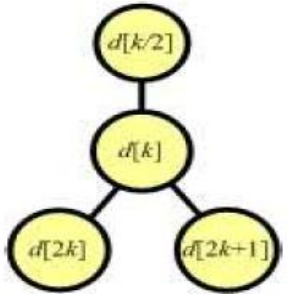
- Element d[1] będzie zawsze korzeniem drzewa.
- i-ty poziom drzewa binarnego wymaga 2^{i-1} węzłów. Będziemy je kolejno pobierać z tablicy.



Otrzymamy w ten sposób następujące odwzorowanie elementów tablicy w drzewo binarne:



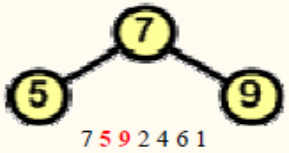
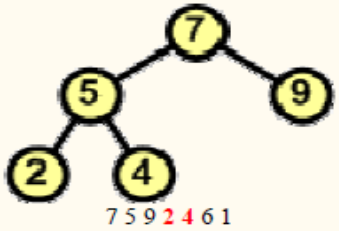
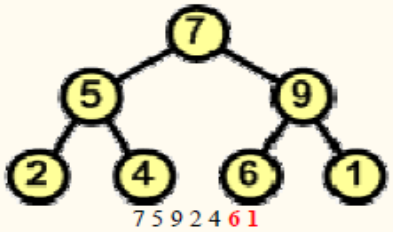
Dla węzła k-tego wyprowadzamy następujące wzory:



węzły potomne mają indeksy równe:
2k - lewy potomek
2k+1 - prawy potomek
węzeł nadrzędny ma indeks równy *[k / 2]* (dzielenie całkowitoliczbowe)

Drzewo binarne

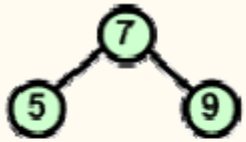
Przykład : Skonstruować drzewo binarne z elementów zbioru {7 5 9 2 4 6 1}

Operacja	Opis
	Konstrukcję drzewa binarnego rozpoczynamy od korzenia, który jest pierwszym elementem zbioru, czyli liczbą 7.
	Do korzenia dołączamy dwa węzły potomne, które leżą obok w zbiorze. Są to dwa kolejne elementy, 5 i 9.
	Do lewego węzła potomnego (5) dołączamy jego węzły potomne. Są to kolejne liczby w zbiorze, czyli 2 i 4.
	Pozostaje nam dołączyć do prawego węzła ostatnie dwa elementy zbioru, czyli liczby 6 i 1. Drzewo jest kompletne.

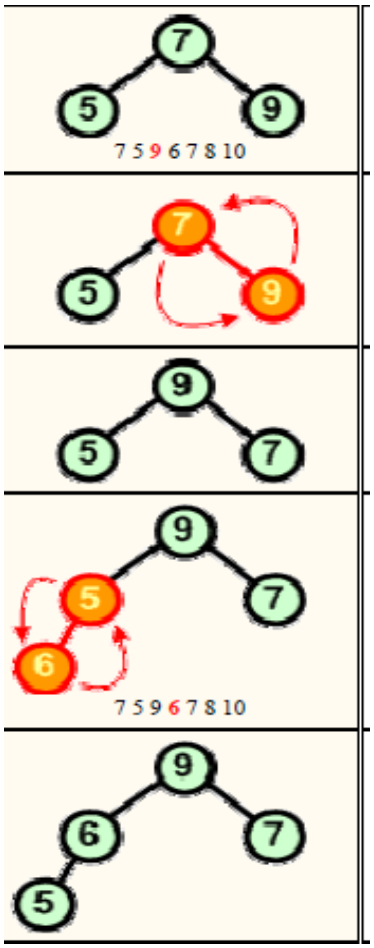
Kopiec : tworzenie

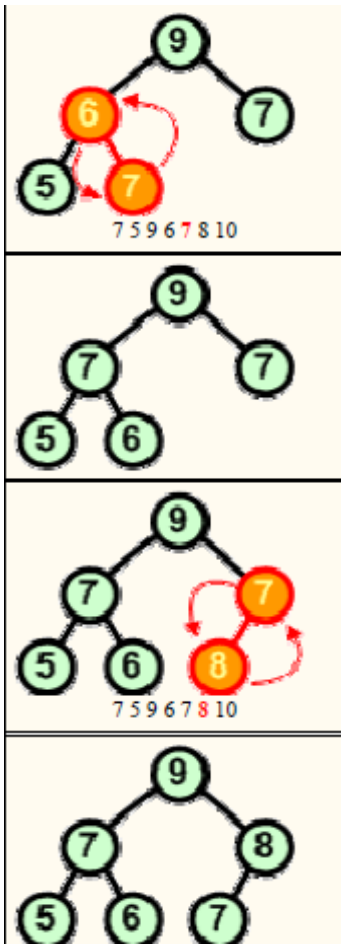
Kopiec jest drzewem binarnym, w którym wszystkie węzły spełniają następujący warunek (zwany warunkiem kopca) : węzeł nadrzędny jest większy lub równy węzłom potomnym (w porządku malejącym relacja jest odwrotna - mniejszy lub równy).

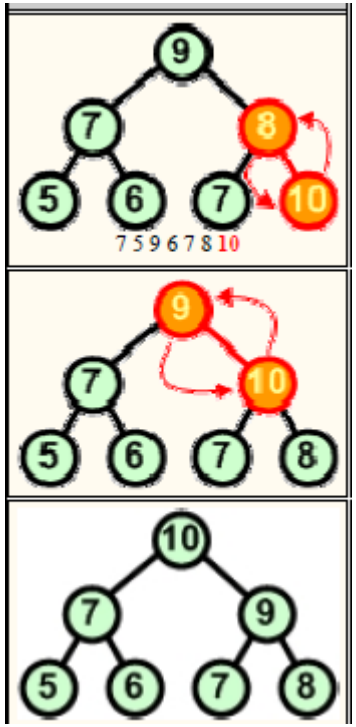
Przykład : Skonstruować kopiec z elementów zbioru {7 5 9 6 7 8 10}

Operacja	Opis
	Budowę kopca rozpoczynamy od pierwszego elementu zbioru, który staje się korzeniem.
	Do korzenia dołączamy następny element. Warunek kopca jest zachowany.
	Dodajemy kolejny element ze zbioru.
	Po dodaniu elementu 9 warunek kopca przestaje być spełniony. Musimy go przywrócić. W tym celu za nowy węzeł nadrzędny wybieramy nowo dodany węzeł. Poprzedni węzeł nadrzędny wędruje w miejsce węzła dodanego - zamieniamy węzły 7 i 9 miejscami.

Kopiec

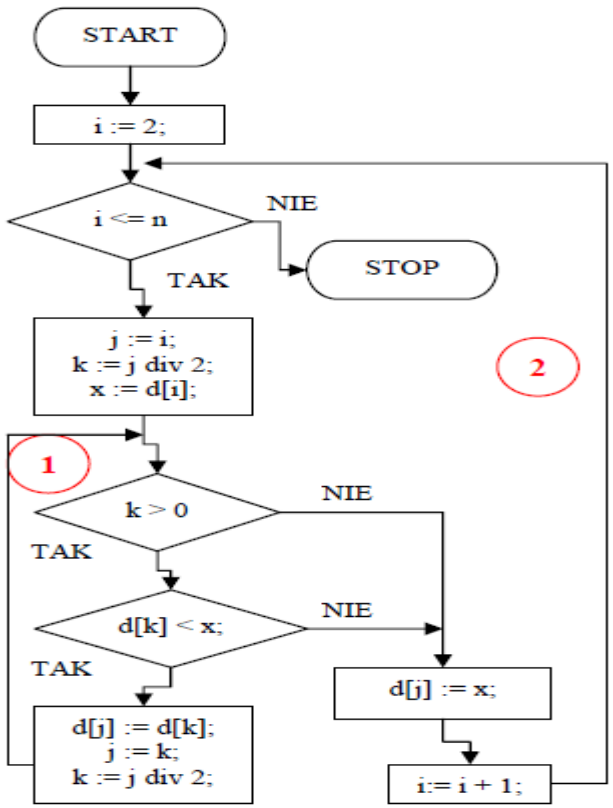






Kopiec

Schemat blokowy



Algorytm tworzy kopiec w tym samym zbiorze wejściowym d[ ]. Nie wymaga zatem dodatkowych struktur danych i ma złożoność pamięciową klasy $\Theta(n)$.
Pętla nr 1 wyznacza kolejne elementy wstawiane do kopca. Pierwszy element pomijamy, ponieważ zostałby i tak na swoim miejscu. Dlatego pętla rozpoczyna wstawianie od elementu nr 2.
Wewnątrz pętli nr 1 inicjujemy kilka zmiennych:
j - pozycja wstawianego elementu (liścia)
k - pozycja elementu nadrzędnego (przodka)
x - zapamiętuje wstawiany element

Następnie rozpoczynamy pętlę warunkową nr 2, której zadaniem jest znalezienie w kopcu miejsca do wstawienia zapamiętanego elementu w zmiennej x. Pętla ta wykonuje się do momentu osiągnięcia korzenia kopca (k = 0) lub znalezienia przodka większego od zapamiętanego elementu. Wewnątrz pętli przesuwamy przodka na miejsce potomka, aby zachować warunek kopca, a następnie przesuwamy pozycję j na pozycję zajmowaną wcześniej przez przodka. Pozycja k staje się pozycją nowego przodka i pętla się kontynuuje. Po jej zakończeniu w zmiennej j znajduje się numer pozycji w zbiorze d[ ], na której należy umieścić element w x.
Po zakończeniu pętli nr 1 w zbiorze zostaje utworzona struktura kopca.

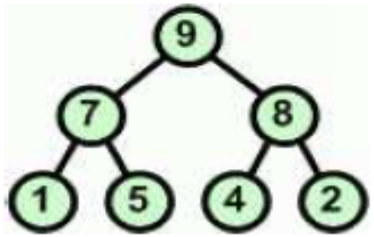
Złożoność obliczeniowa O(n log(n))

Kopiec: rozbieranie

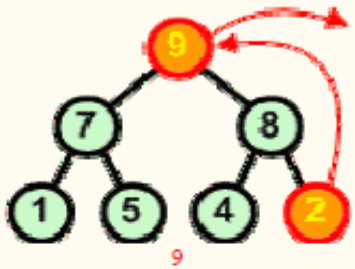
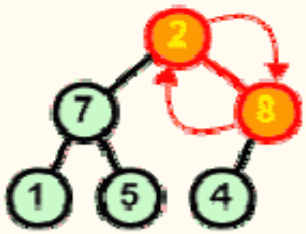
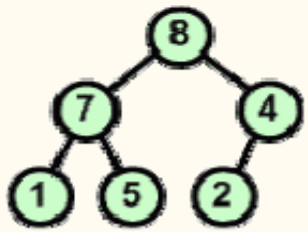
Zasady rozbioru kopca :

1. Zamień miejscami korzeń z ostatnim liściem, który wyłącz ze struktury kopca. Elementem pobieranym z kopca jest zawsze jego korzeń, czyli element największy.
2. Jeśli jest to konieczne, przywróć warunek kopca idąc od korzenia w dół.
3. Kontynuuj od kroku 1, aż kopiec będzie pusty.

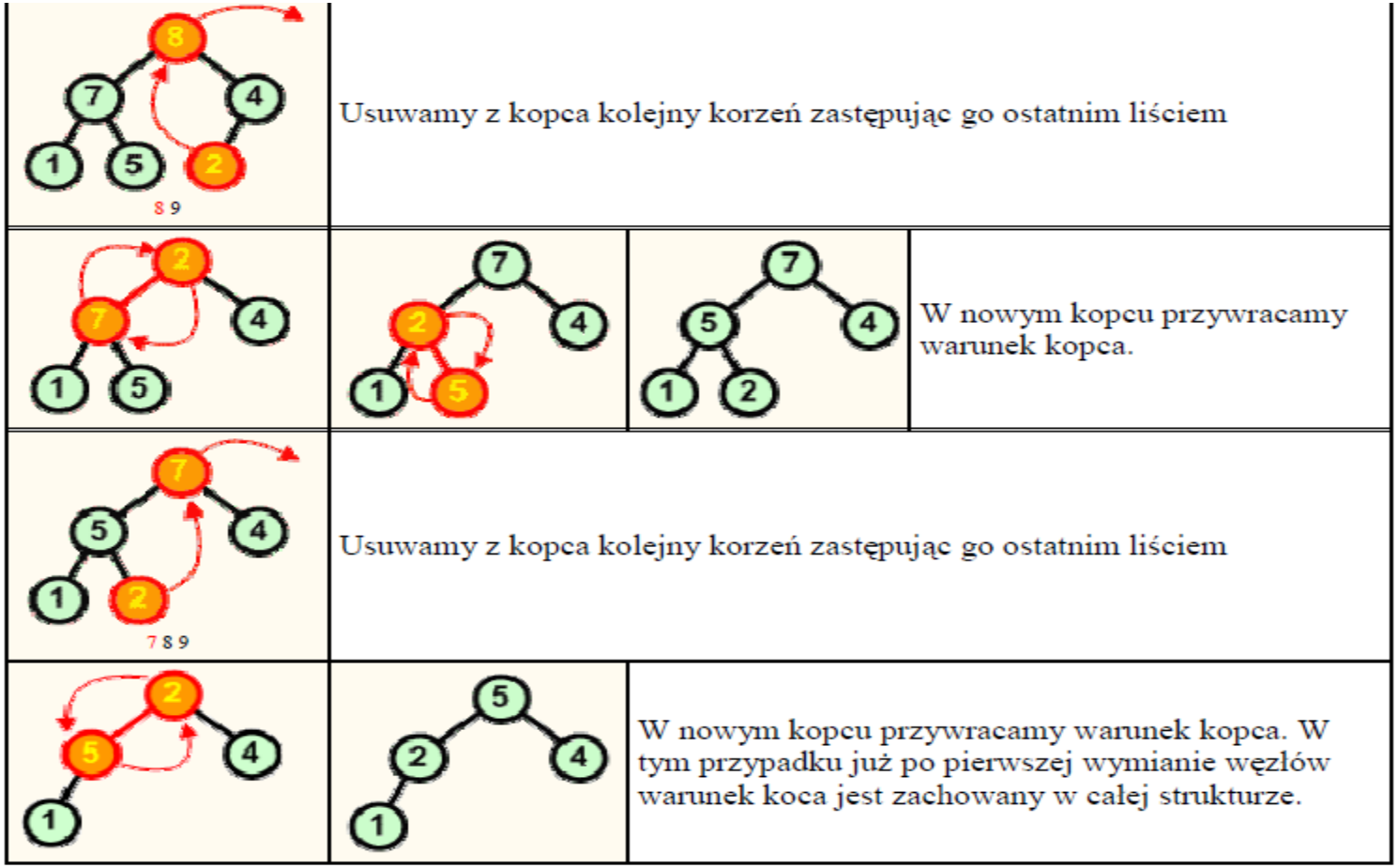
Przykład : Rozebrać kopiec



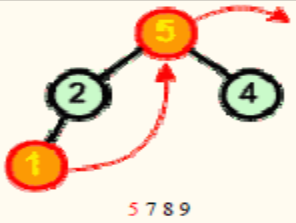
Kopiec: rozbieranie

Z	Operacja	Opis
1 2 3 P		Rozbiór kopca rozpoczynamy od korzenia, który usuwamy ze struktury kopca. W miejsce korzenia wstawiamy ostatni liść.
	  	Poprzednia operacja zaburzyła strukturę kopca. Idziemy zatem od korzenia w dół struktury przywracając warunek kopca - przodek większy lub równy od swoich potomków. Praktycznie polega to na zamianie przodka z największym potomkiem. Operację kontynuujemy dotąd, aż natrafimy na węzły spełniające warunek kopca.

Kopiec: rozbieranie

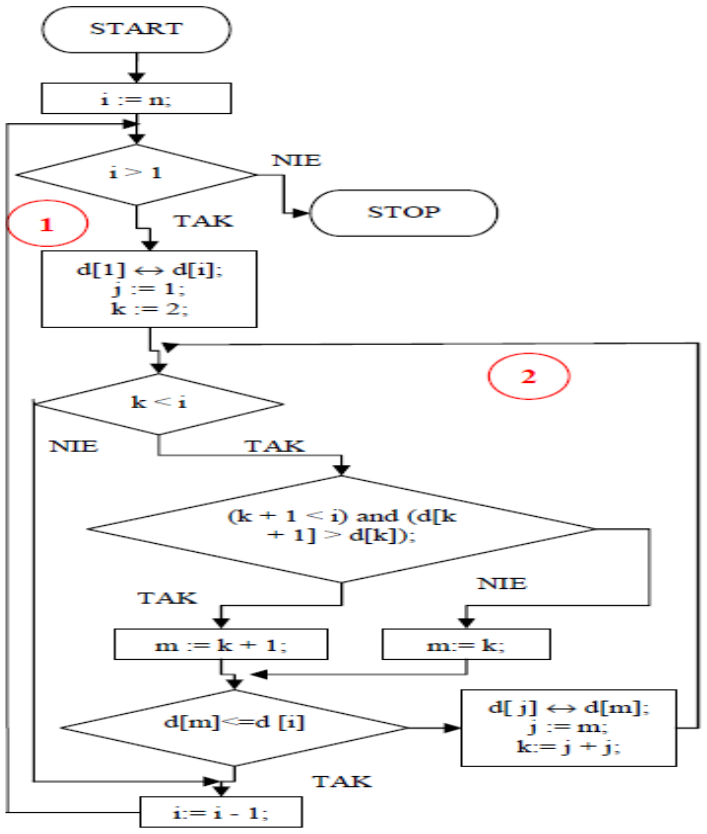


Kopiec: rozbieranie

	Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem
 	Przywracamy warunek kopca w strukturze.
	Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem
 	Przywracamy warunek kopca w strukturze.
	Usuwamy z kopca kolejny korzeń zastępując go ostatnim liściem.
	Po wykonaniu poprzedniej operacji usunięcia w kopcu pozostał tylko jeden element - usuwamy go. Zwróć uwagę, iż usunięte z kopca elementy tworzą ciąg uporządkowany.

Kopiec: rozbieranie

Schemat blokowy



Rozbiór kopca wykonywany jest w dwóch zagnieżdżonych pętlach. Pętla nr 1 zamienia miejscami kolejne liście ze spodu drzewa z korzeniem. Zadaniem pętli nr 2 jest przywrócenie w strukturze warunku kopca.

Złożoność obliczeniowa

O(n log(n))

Sortowanie przez kopcowanie

Krok 1 : Tworz_Kopiec
Krok 2 : Rozbierz_Kopiec
Krok 3 : Zakończ algorytm

Cechy Algorytmu Sortowania Przez Kopcowanie	
klasa złożoności obliczeniowej optymistyczna	Θ (n log n)
klasa złożoności obliczeniowej typowa	
klasa złożoności obliczeniowej pesymistyczna	
Sortowanie w miejscu	TAK
Stabilność	NIE

Ponieważ sortowanie przez kopcowanie składa się z dwóch następujących bezpośrednio po sobie operacji o klasie czasowej złożoności obliczeniowej Θ(n log n), to dla całego algorytmu klasa złożoności również będzie wynosić Θ (n log n).

Algorytmy sortujące

Nazwa algorytmu sortującego	Klasa złożoności			Stabilność	Sortowanie w miejscu	Zalecane?
	optymistyczna	typowa	pesymistyczna			
Zwariowane	$\Theta(1)$	$\Theta(n*n!)$	$\Theta(\infty)$	NIE	TAK	NIE!!!
Naiwne	$\Theta(n)$ ... $\Theta(n^2)$	$\Theta(n^3)$	$\Theta(n^3)$	TAK	TAK	NIE
Bąbelkowe wersja 1	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	TAK	TAK	NIE
Bąbelkowe wersja 2	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	TAK	TAK	TAK
Przez wybór	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	NIE	TAK	TAK/NIE
Przez wstawianie	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$	TAK	TAK	TAK
Metodą Shella	$\Theta(n^{1,14})$	$\Theta(n^{1,15})$	$\Theta(n^{1,15})$	NIE	TAK	TAK
Przez łączenie	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	TAK	NIE	TAK
Przez kopcowanie	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	NIE	TAK	TAK
Szybkie	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$	NIE	TAK	TAK
Kubełkowe wersja I	$\Theta(m + n)$	$\Theta(m + n)$	$\Theta(m + n)$	NIE	NIE	TAK/NIE
Radix Sort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$	TAK	NIE	TAK