ALGORYTMY I STRUKTURY DANYCH

Temat 1: **Algorytm – podstawowe pojęcia**

Wykładowca: dr inż. Zbigniew TARAPATA

e-mail: Zbigniew.Tarapata@isi.wat.edu.pl
http://www.tarapata.strefa.pl/p_algorytmy_i_struktury_danych/

Współautorami wykładu są: G.Bliźniuk, A.Najgebauer, D.Pierzchała

Treść wykładu

- Algorytmika – nauka o algorytmach;
- Algorytm, klasyfikacja algorytmów ;
- Algorytmy iteracyjne i rekurencyjne;
- Własności algorytmów ;
- Złożoność obliczeniowa algorytmów;
- Przykłady szacowania złożoności obliczeniowej algorytmów;

ALGORYTM - przypomnienie podstawowych pojęć

- ***Algorytmika*** **jest dziedziną wiedzy zajmującą się badaniem *algorytmów*;**
- **W informatyce jest ona nieodłącznie związana z algorytmami przetwarzania *struktur danych*;**
- **Potocznie *algorytm* jest rozumiany jako pewien przepis na wykonanie jakiegoś zestawu czynności, prowadzących do osiągnięcia oczekiwanego i z góry określonego celu;**
- **Mówi się również, że:**
- **- *algorytm* jest pewną ściśle określoną procedurą obliczeniową, która dla zestawu właściwych danych wejściowych wytwarza żądane dane wyjściowe;**
- **- *algorytm* jest to zbiór reguł postępowania umożliwiających rozwiązanie określonego zadania w skończonej liczbie kroków i w skończonym czasie.**

Termin algorytm wywodzi się od zlatynizowanej formy (Algorismus, Algorithmus) nazwiska matematyka arabskiego z IX w., Al-Chuwarizmiego.

ALGORYTM - przypomnienie podstawowych pojęć

W sposób formalny algorytm możemy zdefiniować następująco:

Oznaczmy przez:

We - zestaw danych wejściowych,

Wy - zestaw danych wyjściowych.

Algorytm jest rozumiany jako odwzorowanie O, które dla określonego zestawu We generuje zestaw Wy:

$$O: We \rightarrow Wy,$$

gdzie liczności zbiorów We i Wy mogą być różne.

Sposoby zapisu algorytmów

- Algorytm powinien *precyzyjnie* przedstawiać kolejne jego kroki. Do opisu tych kroków mogą być stosowane następujące sposoby:
  - zapisy werbalne,
  - zapisy formalne, np.:
    - zapisy graficzne (schematy blokowe),
    - formalne specyfikacje programów (VDM, CSP)
    - zapisy w postaci pseudokodów („paraprogramów")
    - implementacje programów w dowolnym *języku programowania*

ALGORYTM - przypomnienie podstawowych pojęć

- ***Język programowania*** **jest środkiem umożliwiającym zapis algorytmów w postaci zrozumiałej dla człowieka, a równocześnie przetwarzanej do postaci zrozumiałej dla komputera (maszyny algorytmicznej);**

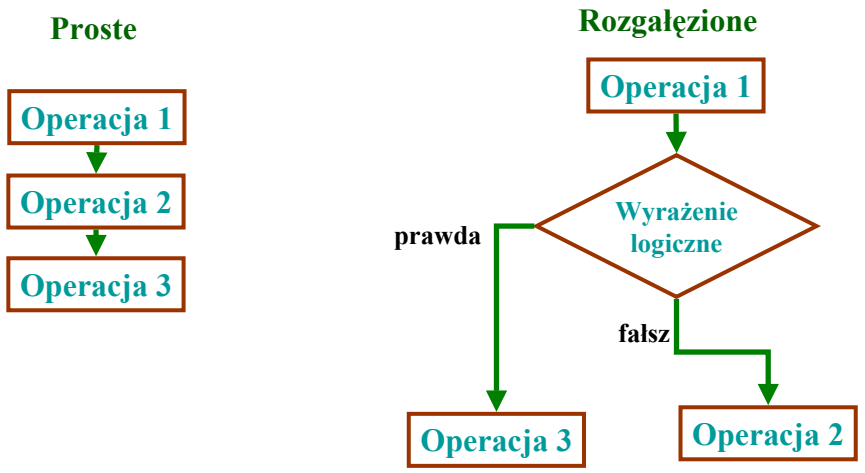
Klasyfikacja algorytmów (wybrane kategorie)

- ***algorytmy proste – rozgałęzione*** ***(nie występują albo występują rozgałęzienia),***
- ***algorytmy cykliczne – mieszane*** ***(z powrotami albo bez powrotów),***
- ***algorytmy równoległe - sekwencyjne*** **(kroki algorytmu wykonywane kolejno w sekwencji lub równolegle),**
- ***algorytmy numeryczne - algorytmy nienumeryczne*** **(wykonywanie obliczeń lub przetwarzanie danych),**
- ***algorytmy rekurencyjne - algorytmy iteracyjne*** **(algorytm w kolejnych krokach wywołuje sam siebie dla nowych wartości parametrów wykonania lub wykonuje obliczenia w pętli dla zmieniającej się wartości jej niezmiennika).**

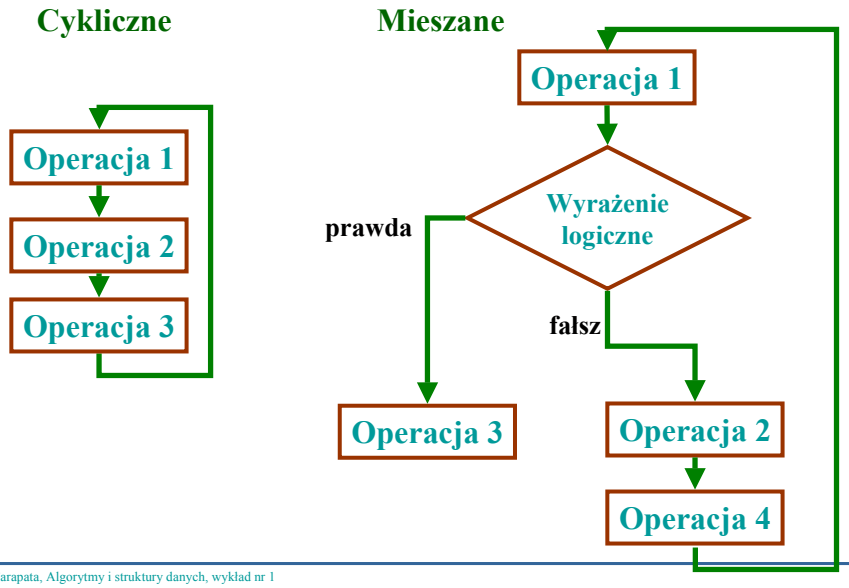
Klasyfikacja algorytmów (wybrane kategorie), c.d.

- ***algorytmy zachłanne*** ***(wykonują działanie które wydaje się najlepsze w danej chwili, nie uwzględniając tego co może się stać w przyszłości);***
- ***algorytmy „dziel i zwyciężaj”*** ***(dzielimy problem na mniejsze części tej samej postaci co pierwotny , aż rozmiar problemu stanie się tak mały, że rozwiązanie będzie oczywiste lub będzie można użyć jakiejś innej efektywnej metody rozwiązania);***
- ***algorytmy oparte o programowanie dynamiczne*** **(wykorzystują własność niektórych problemów polegającą na tym, że optymalne rozwiązanie w iteracjach wcześniejszych gwarantuje otrzymanie optymalnego rozwiązania w kolejnych iteracjach);**
- ***algorytmy z powrotami*** **(*wymagają zapamiętania wszystkich wykonanych ruchów czy też wszystkich odwiedzonych stanów aby możliwe było cofanie posunięć w celu dojścia do rozwiązania*);**

Algorytm prosty, a rozgałęziony

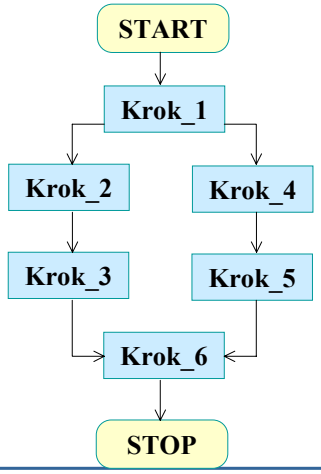


Algorytm cykliczny, a mieszany

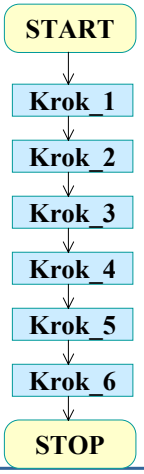


Algorytm równoległy, a sekwencyjny

Algorytm równoległy:

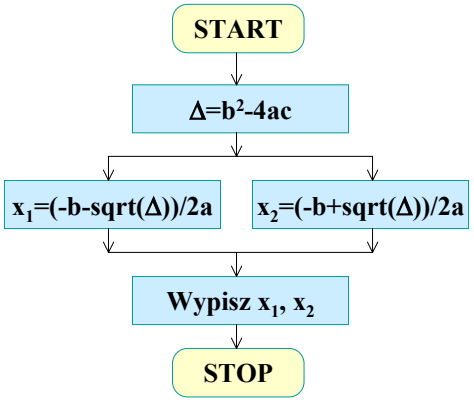


Algorytm sekwencyjny:

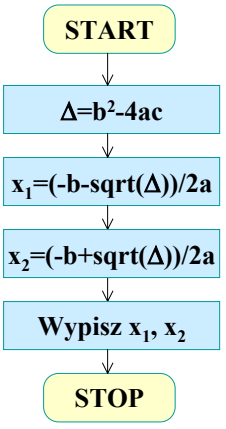


Algorytm równoległy, a sekwencyjny

Algorytm równoległy:

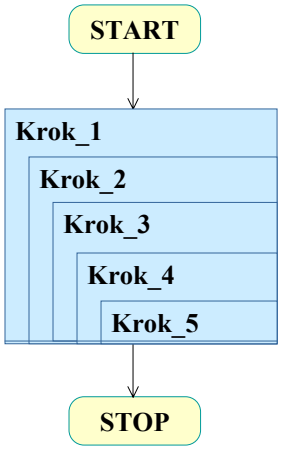


Algorytm sekwencyjny:

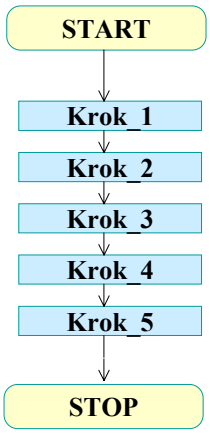


Algorytm iteracyjny, a rekurencyjny

Algorytm rekurencyjny:



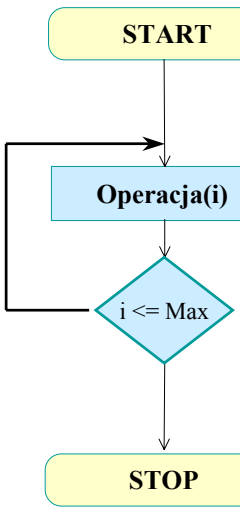
Algorytm iteracyjny:

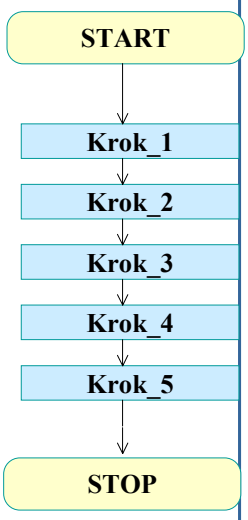


Algorytmy iteracyjne

Przykład:

```
silnia=1;
for(i=2;i<=5;++i)
    silnia = silnia * i;
return silnia;
```

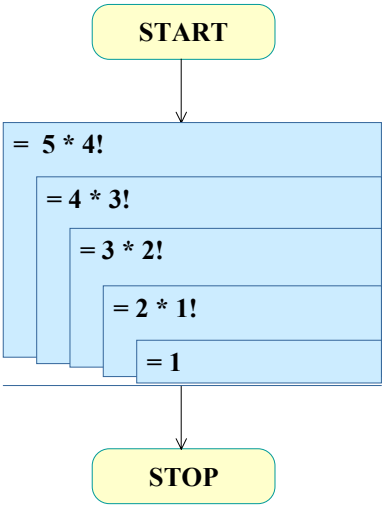




Algorytmy rekurencyjne

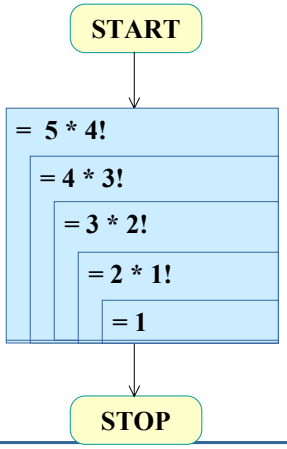
Algorytm wyliczający silnię.

Silnia n!:
Jeśli n=0 lub n=1, to n!=1
Jeśli n > 1, to n! = n * (n-1)!

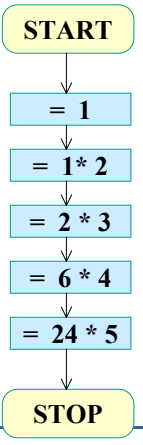


Algorytmy iteracyjne i rekurencyjne - porównanie

Algorytm rekurencyjny:



Algorytm iteracyjny:

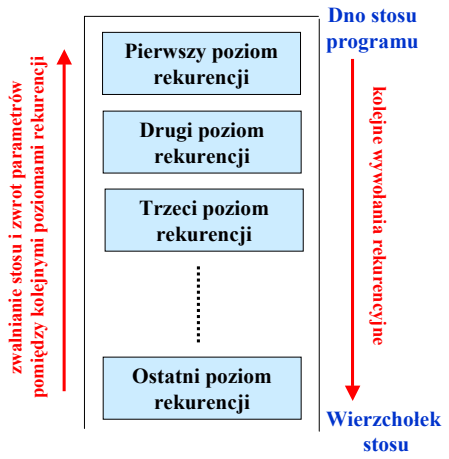


Wywołanie funkcji rekurencyjnej

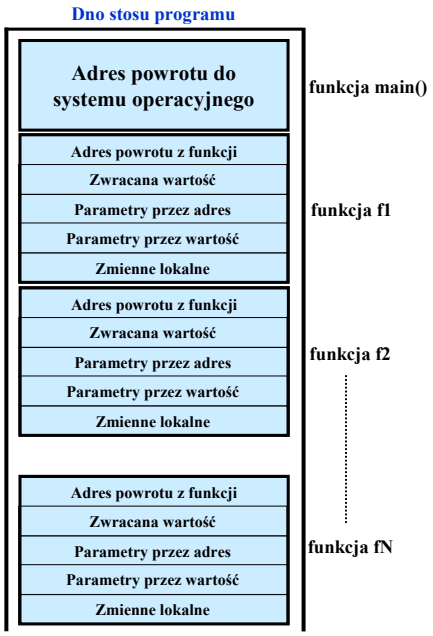
- **Rekurencja oznacza wywołanie funkcji (procedury) przez tę samą funkcję**
- **Ważne jest, aby kolejne wywołania funkcji (procedury) rekurencyjnej były realizowane dla kolejnych wartości parametrów formalnych w taki sposób, aby nie doszło do zjawiska „nieskończonej pętli rekurencyjnych wywołań funkcji”**

Wywołanie funkcji rekurencyjnej

- Kolejne wywołania funkcji rekurencyjnej są związane z odkładaniem na stosie programu kolejnych rekordów aktywacji procedury
- W wyniku kończenia działania poszczególnych funkcji na kolejnych poziomach rekurencji kolejne rekordy aktywacji są zdejmowane ze stosu



Stos programu w wywołaniach rekurencyjnych – przykład C/C++



- Stos programu w wywołaniach rekurencyjnych jest bardziej eksploatowany niż wtedy, gdy wywołania nie są rekurencyjne

  Kolejne poziomy rekurencji wymagają odkładania na stosie programu kolejnych rekordów aktywacji funkcji

  W przypadku procedur mechanizm obsługi stosu jest analogiczny. Różnica polega na tym, że nie adres do zwracanej wartości jest „void”

Przykłady funkcji rekurencyjnych

- Znana już funkcja „silnia”:

$$n! = \begin{cases} 1, & \text{dla } n=0 \\ n*(n-1)! & \text{dla } n>0 \end{cases}$$ (warunek zakończenia rekurencji)

Definicja ciągu liczb wymiernych:

$$f(n) = \begin{cases} 1, & \text{dla } n=0 \\ f(n-1) + (1/f(n-1)), & \text{dla } n>0 \end{cases}$$ (warunek brzegowy, zakończenia), określa ciąg o wartościach:

1, 2, 5/2, 29/10, 941/290, 969581/272890,.....

Funkcja rekurencyjna – ciąg liczb Fibonacciego

- Ciąg liczb Fibonacciego jest wyliczany wg formuły:

$$Fib(n)=\begin{cases} n, & \text{dla } n<2 \\ Fib(n-2) + Fib(n-1), & \text{dla } n>=2 \end{cases}$$

- Rekurencyjna implementacja w języku C:

```
long intFib (int n)
{
  if (n<2)
    return n;
 else
    return Fib(n-2) + Fib (n-1);
}
```

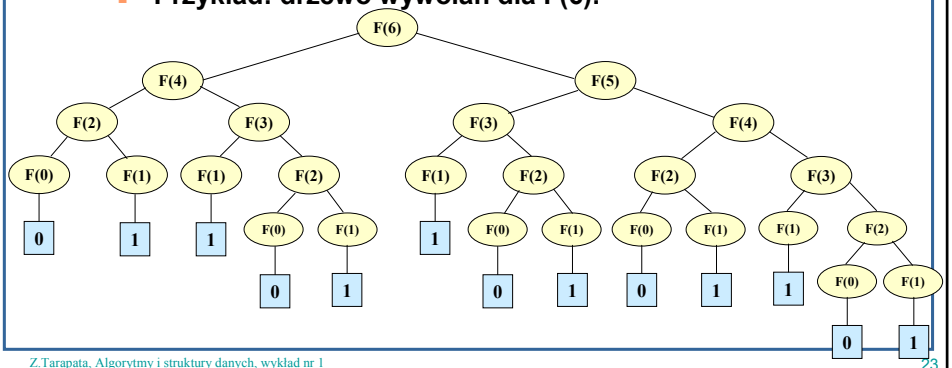
Czy na pewno stos programu „wytrzyma" taką realizację funkcji rekurencyjnej Fib?

Efektywność rekurencyjnego wykonania funkcji Fibonacciego

n	Fib(n+1)	Liczba dodawań	Liczba wywołań
6	13	12	25
10	89	88	177
15	987	986	1 973
20	10 946	10 945	21 891
25	121 393	121 392	242 785
30	1 346 269	1 346 268	2 692 537

Efektywność rekurencyjnego wykonania funkcji Fibonacciego, cd.

- Okazuje się więc, że rekurencyjna implementacja funkcji Fibonacciego jest niezwykle nieefektywna. Stos programu nie jest praktycznie w stanie zrealizować tego algorytmu już dla liczb większych od 9. Oznacza to, że program ma zbyt dużą „złożoność pamięciową".
- Przykład: drzewo wywołań dla F(6):



Iteracyjne wykonanie rekurencyjnej funkcji Fibonacciego

- Bardziej efektywna jest iteracyjna implementacja funkcji Fibonacciego. Nie przepełniamy wtedy stosu programu i wykonujemy mniejszą liczbę przypisań wartości niż w implementacji rekurencyjnej, dla której liczba dodawań wynosi Fib(n+1)-1, natomiast liczba przypisań jest równa: 2*Fib(n+1)-1.
- Przykład implementacji funkcji Fibonacciego metodą iteracyjną :

```
long int IteracyjnyFib(int n)
{ register int i=2, last=0, tmp;  long int current =1;
   if (n<2)
     return n;
   else {
       for (  ; i<=n; ++i)  {
           tmp = current;
           current += last;
           last = tmp;
       }
    return current;
  }
}
```

Efektywność iteracyjnego wykonania rekurencyjnej funkcji Fibonacciego

n	Liczba przypisań dla algorytmu iteracyjnego	Liczba przypisań (wywołań) dla algorytmu rekurencyjnego
6	15	25
10	27	177
15	42	1 973
20	57	21 891
25	72	242 785
30	87	2 692 537

Algorytmy zachłanne

Wykonują działanie które wydaje się najlepsze w danej chwili, nie uwzględniając tego co może się stać w przyszłości. Zaletą jest to że nie traci czasu na rozważanie co może się stać później. Zadziwiające jest że w wielu sytuacjach daje on optymalne rozwiązanie.
Technicznie mówimy: decyzja lokalnie optymalna.

Jak wydać resztę przy minimalnej ilości monet?:
użyj zawsze najpierw monetę o największej dopuszczalnej wartości.

Jak znaleźć globalne maximum? Rozpocznij od pewnej liczby, kolejno powiększaj ją o ustaloną wielkość tak długo jak funkcja wzrasta. Gdy wartość funkcji zaczyna się zmniejszać przerwij i cofnij się do ostatniej pozycji.

Algorytmy „dziel i zwyciężaj”

- **Dzielimy problem na mniejsze części tej samej postaci co pierwotny.**
- **Teraz te podproblemy 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 podproblemów muszą być połączone w celu utworzenia rozwiązania całego problemu.**

Metoda zazwyczaj implementowana z zastosowaniem technik rekurencyjnych.

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).

Własności algorytmów

- ***adekwatność*** - czy algorytm realizuje obliczenia (przetwarzanie) zgodnie z przyjętym celem realizacji obliczeń (przetwarzania)
- ***własność stopu*** - zostały zdefiniowane kryteria zatrzymania wykonywania algorytmu w warunkach poprawnego i niepoprawnego zakończenia obliczeń
- ***jednoznaczność*** - algorytm jest zapisany w sposób na tyle precyzyjny, że jego wykonanie jest prawie automatycznym powtarzaniem kolejnych kroków
- ***powtarzalność*** - każde wykonanie algorytmu przebiega według takiego samego schematu działania i prowadzi do rej samej klasy rozwiązań
- ***złożoność obliczeniowa*** - nakład czasu lub zasobów maszyny realizującej algorytm, niezbędny dla jego prawidłowego wykonania

Złożoność obliczeniowa algorytmów - analiza algorytmów

- ***Analiza algorytmów*** **- złożoność obliczeniowa jest podstawową własnością określaną dla algorytmów. Zadaniem analizy algorytmu jest określenie tej złożoności, a co za tym idzie *realizowalności algorytmu.***
- ***Złożoność zasobowa (pamięciowa)*** **- wyrażana w skali zajętości zasobów (PAO, pamięci zewnętrznych itp.) niezbędnych dla realizacji algorytmu**
- ***Złożoność czasowa*** **- wyrażana w skali czasu wykonania algorytmu (liczba kroków, aproksymowany czas rzeczywisty)**

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Oznaczenia:

α - algorytm rozwiązujący decyzyjny problem Π ;
D_n - zbiór danych rozmiaru *n* dla rozważanego problemu;
$t(I)$ - liczba kroków DTM potrzebna do rozwiązania konkretnego problemu $I \in \mathbf{D}_n$ o rozmiarze $\mathrm{N}(\mathrm{z}) = \mathrm{n}$ przy pomocy algorytmu α.

Definicja

Złożonością obliczeniową (pesymistyczną) algorytmu α nazywamy funkcję postaci

(1) $$W_\alpha(\mathbf{n}) = \max\{\mathbf{t}(I) : I \in \mathbf{D}_n\}$$

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Definicja

Mówimy, że algorytm α ma złożoność obliczeniową wielomianową, jeśli istnieje stała c > 0 oraz wielomian p(n) takie, że:

$$\bigwedge_{n \geq n_0} W_\alpha(n) \leq cp(n)$$

co zapisujemy $W_\alpha(n) = O(p(n))$.

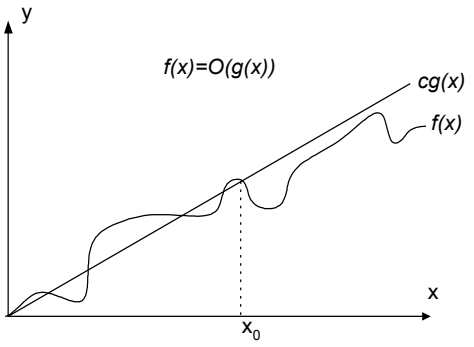
W innych przypadkach mówimy, że algorytm α ma złożoność wykładniczą.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Definicja rzędów złożoności obliczeniowej

Niech $R^* = R^+ \cup \{0\}$.

Mówimy, że funkcja $f(x):R^* \to R^*$ jest rzędu[1] $O(g(x))$ ($g(x):R^* \to R^*$), jeśli istnieje taka stała $c>0$ oraz $x_0 \in R^*$, że dla każdego $x \geq x_0$ zachodzi $f(x) \leq c\,g(x)$ (f nie rośnie szybciej niż g).

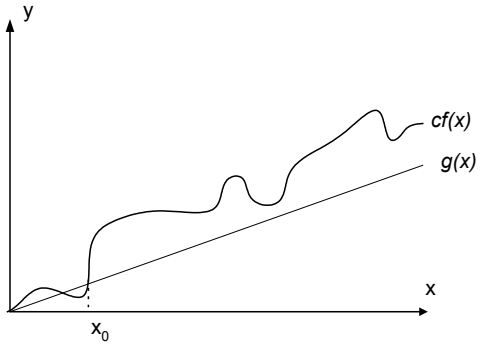


[1] $f(x) = O(g(x))$, gdy $\lim_{x \to \infty} \frac{f(x)}{g(x)} = c$ dla pewnego $c \geq 0$.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Definicja rzędów złożoności obliczeniowej, c.d.

Mówimy, że funkcja $f(x):R^*\to R^*$ jest rzędu[2] $\Omega(g(x))$ ($g(x):R^*\to R^*$), jeśli istnieje taka stała $c>0$ oraz $x_0\in R^*$, że dla każdego $x\geq x_0$ zachodzi $g(x)\leq c\cdot f(x)$ (f nie rośnie wolniej niż g).



[2] $f(x)=\Omega(g(x))$, gdy $\lim_{x\to\infty}\frac{f(x)}{g(x)}=\infty$ lub $\lim_{x\to\infty}\frac{f(x)}{g(x)}=c>0$.

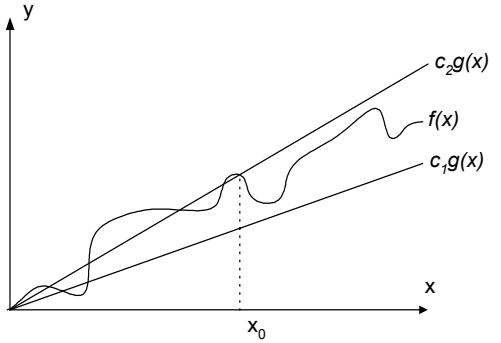
SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Definicja rzędów złożoności obliczeniowej, c.d.

Mówimy, że funkcja $f(x):R^*\to R^*$ jest rzędu $\Theta(g(x))$ ($g(x):R^*\to R^*$), jeśli istnieją takie stałe c_1, $c_2>0$ oraz $x_0\in R^*$, że dla każdego $x\geq x_0$ zachodzi

$$c_1g(x)\leq f(x)\leq c_2g(x)$$

(f rośnie tak samo jak g).



Jeżeli funkcja $f(x)$ jest $\mathbf{O(g(x))}$ oraz $\mathbf{\Omega(g(x))}$, to jest $\mathbf{\Theta(g(x))}$.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Definicja rzędów złożoności obliczeniowej, Przykład

Przykład 1

- $2\,|\sin x| = O(\log x)$, $\quad 2\,|\sin x| = O(1)$
- $x^3+5x^2+2=O(x^3)$ oraz $x^3+5x^2+2=\Omega(x^3)$, więc
  $x^3+5x^2+2=\Theta(x^3)$
- $f(x)=x^2+bx+c=\Theta(x^2)$,
  gdyż $\forall x \geq x_0 = 2 \cdot \max\left\{\frac{|b|}{a}, \sqrt{\frac{|c|}{a}}\right\}$ zachodzi:
  $c_1 g(x) \leq f(x) \leq c_2 g(x)$ dla $c_1 = \frac{1}{4}a$, $c_2 = \frac{7}{4}a$, $g(x)=x^2$
- $\frac{1}{1+x^2} = O(1)$, $\forall x \geq x_0 = 1$ **oraz np. dla $c \geq 1$;**
- $x! = O(x^x)$, $\forall x \geq x_0 = 1$ **oraz np. dla $c \geq 1$.**

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Własności funkcji rzędów złożoności obliczeniowej – potencjalne problemy

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.

Przykład 3:
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 tez stała (10^8), która w tym przypadku jest zbyt duża, aby notacja była znacząca.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ
Jak mierzyć czas działania algorytmu?

Sposób pomiaru ilości pracy wykonanej przez algorytm powinien zapewniać:

- **porównanie efektywności dwóch różnych algorytmów rozwiązujących ten sam problem;**
- **oszacowanie faktycznej wydajności metody, abstrahując od komputera, języka programowania, umiejętności programisty i szczegółów technicznych implementacji (sposobu inkrementacji zmiennych sterujących pętli, sposobu obliczania indeksów zmiennych ze wskaźnikami itp.);**

WNIOSEK:
Dobrym przybliżeniem czasochłonności algorytmu jest obliczenie liczby przejść przez wszystkie pętle w algorytmie.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ
Jak mierzyć czas działania algorytmu?, c.d.

W praktyce zlicza się tzw. operacje podstawowe dla badanego problemu lub klasy rozważanych algorytmów, tj. takie, które są najczęściej wykonywane (ignorując pozostałe operacje pomocnicze, takie jak instrukcje inicjalizacji, instrukcje organizacji pętli itp.).

Ponieważ większość programów to programy złożone z wielu modułów lub podprogramów, a w każdym takim podprogramie inna instrukcja może grać rolę operacji podstawowej, więc fragmenty większej całości analizuje się zwykle oddzielnie i na podstawie skończonej liczby takich modułów szacuje się czasochłonność algorytmu jako całości.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ
Jak mierzyć czas działania algorytmu?, c.d.

Tabela 1 Przykłady operacji podstawowych dla typowych problemów obliczeniowych

Lp.	*Problem*	*Operacja*
1.	Znalezienie x na liście nazwisk.	Porównanie x z pozycją na liście.
2.	Mnożenie dwóch macierzy liczb rzeczywistych.	Mnożenie dwóch macierzy liczb typu real (lub mnożenie i dodawanie).
3.	Porządkowanie liczb.	Porównanie dwóch liczb (lub porównanie i zamiana).
4.	Wyznaczanie drogi najkrótszej w grafie zadanym w postaci listy sąsiadów.	Operacja na wskaźniku listy.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ
Jak mierzyć czas działania algorytmu?, c.d.

Zalety zliczania operacji podstawowych:

- **możliwość przewidywania zachowania się algorytmu dla dużych rozmiarów danych (jeżeli łączna liczba operacji jest proporcjonalna do liczby operacji podstawowych);**
- **swoboda wyboru operacji podstawowej.**

W skrajnym przypadku można wybrać rozkazy maszynowe konkretnego komputera. Z drugiej strony jedno przejście przez instrukcję pętli można również potraktować jako operację podstawową. W ten sposób możemy manipulować stopniem precyzji w zależności od potrzeb.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Przykłady praktyczne mierzenia złożoności obliczeniowej – Przykład 4

Przykład 4: **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" i zmianę wartości zmiennej „i". Mamy zatem 2n przypisań podczas całego wykonania pętli; jej asymptotyczna złożoność wynosi O(n).

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Przykłady praktyczne mierzenia złożoności obliczeniowej – Przykład 5

Przykład 5: **Pętla zagnieżdżona**

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

Na samym początku zmiennej „i" nadawana jest wartość początkowa. 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 $i \in \{1,...,n-1\}$, 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. Jej asymptotyczna złożoność wynosi $O(n^2)$.

Pętle zagnieżdżone mają zwykle większą złożoność niż pojedyncze, jednak nie musi tak być zawsze.

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ
Przykłady praktyczne mierzenia złożoności obliczeniowej – Przykład 6

Analiza tych dwóch przypadków była stosunkowo prosta ponieważ liczba iteracji nie zależała od wartości elementów tablicy.
Wyznaczenie złożoności asymptotycznej jest trudniejsze jeżeli liczba iteracji nie jest zawsze jednakowa.

Przykład 6: **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 raz. Złożoność algorytmu jest więc *O(n).*

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ
Przykłady praktyczne mierzenia złożoności obliczeniowej – Przykład 6, c.d.

Analiza tych dwóch przypadków była stosunkowo prosta ponieważ liczba iteracji nie zależała od wartości elementów tablicy.
Wyznaczenie złożoności asymptotycznej jest trudniejsze jeżeli liczba iteracji nie jest zawsze jednakowa.

Przykład 6: **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 rosnąco, to pętla zewnętrzna wykonuje się n-1 razy, a w każdym jej przebiegu pętla wewnętrzna wykona się i razy dla i $\in$ {1,...,n-1}.
Złożoność algorytmu jest więc *$O(n^2)$.*

SZACOWANIE ZŁOŻONOŚCI OBLICZENIOWEJ

Przykłady praktyczne mierzenia złożoności obliczeniowej – Przykład 6, c.d.

Analiza tych dwóch przypadków była stosunkowo prosta ponieważ liczba iteracji nie zależała od wartości elementów tablicy.
Wyznaczenie złożoności asymptotycznej jest trudniejsze jeżeli liczba iteracji nie jest zawsze jednakowa.

Przykład 6: **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;   }
```

=> Z reguły dane nie są uporządkowane i ocena złożoności algorytmu jest rzeczą niełatwą, ale bardzo istotną. Staramy się wyznaczy złożoność w „przypadku optymistycznym", „przypadku pesymistycznym" oraz „przypadku średnim". Często posługujemy się przybliżeniami opartymi o wcześniej zdefiniowane notacje „duże O, Ω i Θ".

Dziękuję za uwagę