- Wykład *Algorytmy i struktury danych* jest poświęcony przede wszystkim metodom efektywnego rozwiązywania problemów na komputerze.
- Podstawowym elementem przy rozwiązywaniu zadanego problemu jest dobór algorytmu i struktury danych.
- Najważniejszymi aspektami algorytmu są jego **poprawność** i **złożoność** (czasowa i pamięciowa).

Złożoność czasowa

W przypadku złożoności czasowej, z reguły wyróżniamy pewną **operację dominującą**, a czas będziemy traktować jako liczbę wykonanych operacji dominujących.
W ten sposób analiza będzie zależna jedynie od algorytmu, a nie od implementacji i sprzętu.

W przypadku sortowania, operacją dominującą jest przeważnie porównanie dwóch elementów
W przypadku przeglądania drzewa operacją dominującą jest jedno przejście w drzewie między wierzchołkami.

Złożoność czasowa

- Zazwyczaj określamy pewien parametr *n*, będący rozmiarem problemu wejściowego i określamy złożoność jako funkcję *T(n)*, której argumentem jest rozmiar problemu.
- Z reguły będziemy przyjmować, że każda operacja arytmetyczna na małych liczbach daje się wykonać w jednym kroku.

Złożoność czasowa

- Złożoność algorytmu może być rozumiana w sensie złożoności najgorszego przypadku lub złożoności średniej.

  - Złożoność najgorszego przypadku nazywamy **złożonością pesymistyczną** - jest to maksymalna złożoność dla danych tego samego rozmiaru $T(n)$.

Złożoność czasowa

- W praktyce ważniejsza może się okazać **złożoność średnia** lub **oczekiwana**.

  W tym przypadku *T(n)* jest średnią (oczekiwaną) wartością złożoności dla wszystkich problemów rozmiaru *n*.
  Tego typu złożoność zależy istotnie od tego, jaka się pod tym kryje przestrzeń probabilistyczna danych wejściowych.
  Z reguły zakładamy, że wszystkie dane wejściowe tego samego rozmiaru mogą się pojawić z tym samym prawdopodobieństwem.

Oznaczenia:

Oznaczenia:

- D_n - zbiór zestawów danych wejściowych rozmiaru n;
- $t(d)$ - liczba operacji dominujących dla zestawu danych wejściowych d;
- X_n - zmienna losowa, której wartością jest $t(d)$ dla $d \in D_n$;
- p_{nk} - rozkład prawdopodobieństwa zmiennej losowej X_n, tzn. prawdopodobieństwo, że dla danych rozmiaru n algorytm wykona k operacji dominujących $(k \geq 0)$.

- Rozkład prawdopodobieństwa zmiennej losowej $\boldsymbol{X_n}$ wyznacza się na podstawie informacji o zastosowaniach rozważanego algorytmu.
- Gdy zbiór D_n jest skończony, przyjmuje się często model probabilistyczny, w którym każdy zestaw danych rozmiaru n może się pojawić na wejściu do algorytmu z jednakowym prawdopodobieństwem.

Pesymistyczna złożoność czasowa algorytmu

Pesymistyczna złożoność algorytmu to funkcja:

$$W(n) = \sup \{ t(d): \mathbf{d} \in \mathbf{D_n} \},$$

gdzie sup oznacza kres górny zbioru.

Oczekiwana złożoność czasowa algorytmu

Oczekiwana złożoność algorytmu to funkcja:

$$A(n) = \sum_{k \geq 0} k p_{nk}$$

(wartość oczekiwana zmiennej losowej X_n - $E(X_n)$

Aby stwierdzić, na ile funkcje $W(n)$ oraz $A(n)$ są reprezentatywne dla wszystkich danych wejściowych rozmiaru n, uwzględnia się miary wrażliwości algorytmu:

- **Miarę wrażliwości pesymistycznej**
- **Miarę wrażliwości oczekiwanej**

- Miara wrażliwości pesymistycznej

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

- Miara wrażliwości oczekiwanej

(odchylenie standardowe zmiennej losowej X_n)

$$\delta(n) = \sqrt{\mathrm{var}(X_n)}$$

gdzie: $\mathrm{var}(X_n) = \sum_{k \geq 0} \left(k - E(X_n)\right)^2 p_{nk}$

(var(X_n) jest wariancją zmiennej losowej X_n)

Przykład.

Przypuśćmy, że chcemy znaleźć pierwszą jedynkę w *n*-elementowej tablicy zerojedynkowej i nasz algorytm przegląda tablicę od strony lewej sprawdzając kolejne elementy. (zakładamy, że jedynka występuje w tablicy). Niech operacją dominującą będzie sprawdzenie jednego elementu.

```
int j=0;

while (a[j]!=1) j++;

p=j// numer komórki w której występuje pierwsza jedynka
```

Przykład.

Rozmiar danych wejściowych: *n*

Operacja dominująca: *porównanie a[j]!=1*

Pesymistyczna złożoność czasowa: *W(n)=n* (gdy „1” w ostatniej komórce)

Oczekiwana złożoność czasowa: *A(n)=?*

Załóżmy, że prawdopodobieństwo znalezienia 1 na każdym z *n* możliwych miejsc jest takie samo i wiadomo, że 1 jest w tablicy:

$$p_{nk}=1/n, \text{ dla } k=1, 2, ..., n$$

Wtedy:

$$A(n) = \sum_{k=1}^{n} kp_{nk} = \frac{1}{n} * \sum_{k=1}^{n} k = \frac{1}{n} * \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Pesymistyczna wrażliwość czasowa: $\Delta(n)=n-1$

Oczekiwana wrażliwość czasowa:

$$\delta(n) = \sqrt{\mathrm{var}(X_n)}$$

$$\mathrm{var}(X_n) = \sum_{k \geq 0} \left(k - E(X_n)\right)^2 p_{nk} = \sum_{k=1}^{n} \left(k - \frac{n+1}{2}\right)^2 \frac{1}{n} =$$

$$\frac{1}{n}\left(\frac{n(n+1)(2n+1)}{6} - \frac{2(n+1)}{2}\frac{n(n+1)}{2} + n\left(\frac{n+1}{2}\right)^2\right) =$$

$$\frac{(n+1)(2n+1)}{6} - \frac{(n+1)^2}{4} = \frac{n+1}{12}(4n+2-3n-3) = \frac{n^2-1}{12} \cong \frac{1}{12}n^2$$

czyli $\delta(n) \cong 0.29n$

W notacji używanej do opisu asymptotycznego czasu działania algorytmów korzysta się z funkcji, których zbiorem argumentów jest zbiór liczb naturalnych.

Notacja Θ

Notacja O

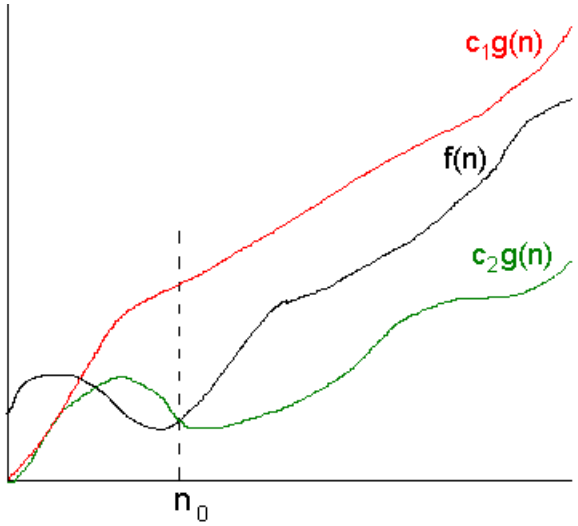
Notacja Ω

Notacja Θ

Dla danej funkcji *g(n)* przez ***Θ(g(n))*** *(„duże theta od g od n”)*

oznaczamy zbiór funkcji:

$$\Theta(g(n)) = \{ f(n) : \exists\, c_1, c_2, n_0 > 0 :\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n)\ \ \forall\, n > n_0 \}$$

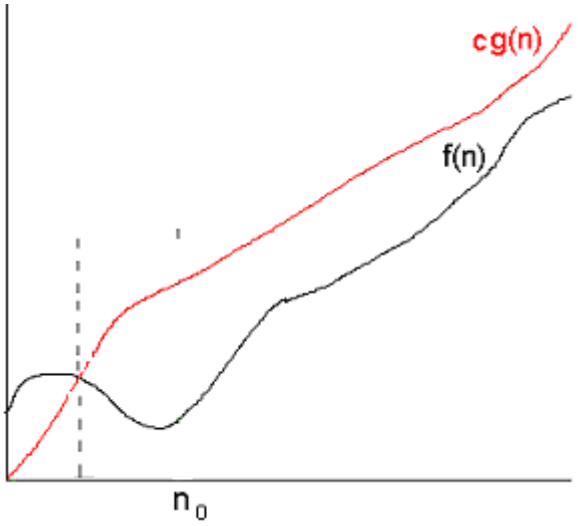


Notacja O

Notacja Θ asymptotycznie ogranicza funkcję od góry i od dołu. Kiedy mamy tylko **ograniczenie górne**, używamy **notacji *O***.

$$O(g(n)) = \{ f(n) : \exists\, c, n_0 > 0 :\ f(n) \leq c_2 g(n)\ \ \forall\, n > n_0 \}$$

Z notacji ***O*** korzystamy, gdy chcemy oszacować funkcję z góry z dokładnością do stałej.



$$f(n) = O(g(n))$$

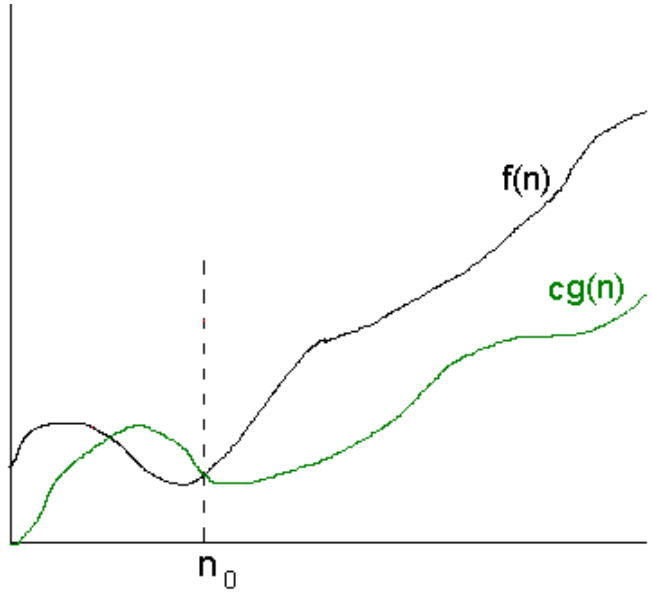
Uwaga: Notacja Θ jest silniejsza niż notacja O

Ponieważ notacja O odpowiada ograniczeniu górnemu, stosując ją do oszacowania pesymistycznego czasu działania algorytmu, uzyskujemy górne ograniczenie czasu działania tego algorytmu dla wszystkich danych wejściowych.

Notacja Ω

Notacja Ω asymptotycznie ogranicza funkcję od dołu.

$$\Omega(g(n)) = \{ f(n) : \exists\, c, n_0 > 0 :\ 0 \le cg(n) \le f(n)\ \forall\, n > n_0 \}$$



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

Twierdzenie

Dla każdych dwóch funkcji $f(n)$ i $g(n)$ zachodzi zależność:

$$f(n)=\Theta(g(n)) \Leftrightarrow (\ f(n)=O(g(n)) \ \text{ i } \ f(n)=\Omega(g(n)))$$

Ćw.1. Czy $T(n) = 3n^3 + 2n^2$ jest $O(n^3)$?

Dodawanie i mnożenie w notacji O

Dodawanie i mnożenie w notacji O

Reguła sumowania:

Załóżmy, że $T_1(n)$ i $T_2(n)$ są czasami wykonania fragmentów programu P_1 i P_2.

$T_1(n)$ jest $O(f(n))$ a $T_2(n)$ jest $O(g(n))$.

Wtedy czas wykonania fragmentów P_1 i P_2 jest :

$T_1(n)+ T_2(n)$ jest $O(\max(f(n), g(n))$.

Dodawanie i mnożenie w notacji O

Reguła mnożenia:

Załóżmy, że $T_1(n)$ i $T_2(n)$ są czasami wykonania fragmentów programu P_1 i P_2.

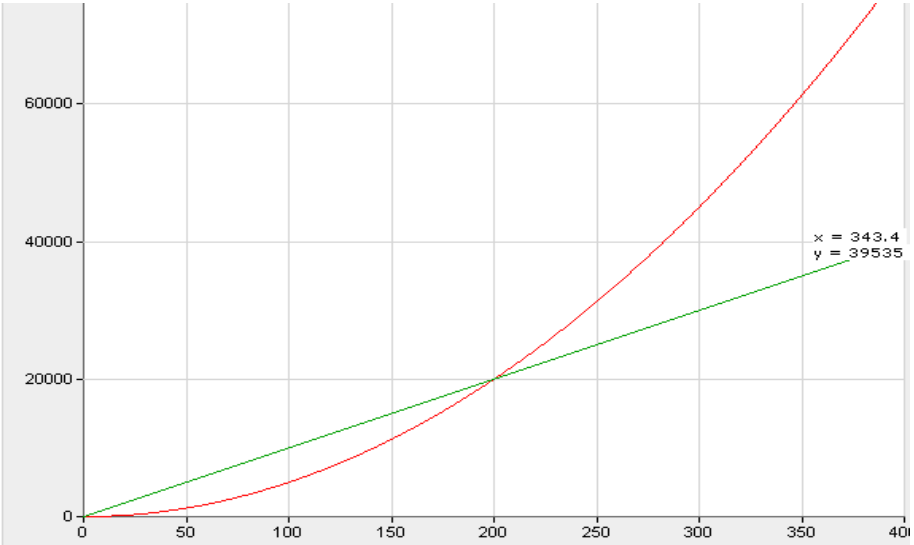
$T_1(n)$ jest $O(f(n))$ a $T_2(n)$ jest $O(g(n))$.

Wtedy:

$T_1(n)*T_2(n)$ jest $O(f(n)* g(n))$.

Zadanie:

Mamy dwa algorytmy rozwiązujące pewien problem: algorytm A1 i algorytm A2.
Czas wykonania tych programów dla danych o rozmiarze n wynosi odpowiednio:
T(A1,n)=100*n
T(A2,n)=1/2n^2
Jakiego rzędu są te algorytmy?
Który algorytm jest lepszy?
Który zastosowac dla danych rozmiaru 100,200,300?



Przykład obliczania złożoności algorytmu (sortowanie bąbelkowe)

Przykład obliczania złożoności algorytmu (sortowanie bąbelkowe)

```
sortowaniebąbelkowe();
int x;
{
(1)for (int i=1; i<n; i++)
(2)  {for (int j=n-1; j>=i; j--)
(3)     if (a[j-1] >a[j])
          { //zamiana
(4)          x=a[j-1];
(5)          a[j-1]=a[j];
(6)          a[j]=x;
          }
     }
}
```

Rozmiar wejścia *n* oznacza tu liczbę elementów, które mają być posortowane.
Uwagi:

1. Każda instrukcja podstawienia zajmuje pewna stałą ilość czasu, niezależnie od *n*. Zatem można powiedzieć, że instrukcje (4), (5) i (6) są *O(1)*. Z reguły sumowania złożoność fragmentu (4)(5)(6) jest *O(max(1,1,1))=O(1)*.
2. Sprawdzenie warunku dla instrukcji *if* jest *O(1)*.
   Nie wiemy, czy instrukcje (4)-(6) będą zawsze wykonywane. Licząc złożoność pesymistyczną zakładamy, że tak. Czyli instrukcje (3)-(6) mają złożoność *O(1)*.
3. Dla pętli (2)-(6) czas wykonania jest sumą czasów wykonania wnętrza pętli dla każdego nawrotu pętli. Należy przyjąć, że zwiększanie zmiennej iteracyjnie, sprawdzenie warunku wyjścia oraz ewentualny skok do początku pętli zajmuję

O(1). Wnętrze pętli też jest *O(1),* a liczba iteracji pętli wynosi *n-i.* Zatem na podstawie reguły mnożenia złożoność fragmentu (2)-(6) jest *O((n-i)*1)=O(n-i).*

4. Ponieważ pętla (1) wykonuje się (n-1) razy, to złożoność czasowa całego algorytmu wynosi:

$$\sum_{i=1}^{n-1}(n-i) = n-1+n-2+\ldots+n-(n-1) =$$

$$n-1+n-2+\ldots+1 = \frac{1+(n-1)}{2}(n-1) =$$

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

Zatem cały program jest $O\left(\frac{n^2}{2} - \frac{n}{2}\right)$ co jest $O\left(\frac{n^2}{2}\right)$(z reguły sumowania), co jest $O\left(n^2\right)$ (z reguły mnożenia).

Analiza złożoności- sortowanie kubełkowe

Przykład 2. Sortowanie kubełkowe

Algorytm sortowania kubełkowego rozwiązuje następujący problem:

Posortować elementy tablicy o n elementach będących liczbami naturalnymi z przedziału *0,…,m-1*, dla pewnego zadanego *m*

Algorytm sortowania kubełkowego wygląda następująco:

1.Tworzona jest tablica *m* kubełków. Kubełki zainicjowane są wartością 0

2. Pętla uaktualniająca liczniki – kubełek *i-ty* pamięta liczbę wystąpień liczby *i* w tablicy

3. Posortowana tablica jest otrzymana w wyniku wstawienia wymaganej liczby „0” na początek tablicy, następnie wymaganej liczby „1”,itd., aż do liczb „*m-1*”

Analiza złożoności- sortowanie kubełkowe

Analiza złożoności- sortowanie kubełkowe

```
unsigned int const m = ?;

void BucketSort (unsigned int a [], unsigned int n)
{
(1)     int buckets [m];

(2)     for (unsigned int j = 0; j < m; ++j)
(3)        buckets [j] = 0;
(4)     for (unsigned int i = 0; i < n; ++i)
(5)        ++buckets [a [i]];
(6)     for (unsigned int i = 0, j = 0; j < m; ++j)
(7)     for (unsigned int k = buckets [j]; k > 0; --k)
(8)         a [i++] = j;
}
```

Analiza złożoności- sortowanie kubełkowe

instr	n=0
Pętla 2-3	*O(m)*
Pętla 4-5	*O(n)*
Pętla 6-8	?
razem	*?*

Rozważmy pętle 6-8. Wykonujemy dokładnie *m* iteracji pętli wewnętrznej.

Liczba iteracji pętli wewnętrznej zależy od wartości `bucket[j]`.

Ponieważ w tablicy wejściowej mamy *n* elementów, w najgorszym przypadku wartość `bucket[j]` może wynosić *n*.

Tak więc złożoność czasowa pętli 6-8 jest rzędu *O(mn)*

Analiza złożoności- sortowanie kubełkowe

Czy jest to dobre ograniczenie górne?

Rozważmy pętle wewnętrzną 7-8.

Podczas *j-tej* iteracji pętli wewnętrznej, pętla wewnętrzna wykonuje `bucket[j]` iteracji.

Warunek pętli wewnętrznej jest więc sprawdzany `bucket[j]+1` razy.

Liczba wykonanych testów warunkowych wynosi:

$$\sum_{j=0}^{m-1} \left(bucket[j] + 1 \right) = \sum_{j=0}^{m-1} bucket[j] + \sum_{j=0}^{m-1} 1 = n + m$$

Analiza złożoności- sortowanie kubełkowe

instr	n=0
Pętla 2-3	$O(m)$
Pętla 4-5	$O(n)$
Pętla 6-8	$O(m+n)$
razem	$O(m+n)$

Większość rozważanych algorytmów ma złożoność czasową proporcjonalną do jednej z podanych funkcji:

- **$\log_2 n$ - złożoność logarytmiczna**
  - np. poszukiwanie binarne w ciągu uporządkowanym:

  -{ j=1; while (j<n) j=2*j; }

- **n - złożoność liniowa**
  - dla algorytmów, w których wykonywana jest pewna stała liczba działań dla każdego z *n* elementów wejściowych.

- **$n*\log_2 n$ - złożoność liniowo logarytmiczna**

- **n^2 - złożoność kwadratowa**
- **n^3 , n^4 - złożoności wielomianowe**
- **2^n - złożoność wykładnicza 2^n**
- **n! - złożoność wykładnicza n!**

UWAGA: Algorytmy o złożoności wykładniczej mogą być realizowane jedynie dla danych małych rozmiarów

Przy korzystaniu z wyników analizy złożoności algorytmów należy brać pod uwagę następujące uwarunkowania:

- wrażliwość algorytmu na dane wejściowe może spowodować, że faktyczne zachowanie algorytmu na używanych danych może odbiegać od zachowania opisanego funkcjami $W(n)$ i $A(n)$
- może być trudno przewidzieć rzeczywisty rozkład prawdopodobieństwa zmiennej losowej X_n
- czasami trudno porównać jednoznacznie działanie dwóch algorytmów: jeden działa lepiej dla pewnej klasy zestawów danych, a drugi dla innej
- algorytm i jego realizacja przeznaczona do wykonania są zwykle wyrażone w dwóch całkowicie różnych językach