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.

- 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 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)*.

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

Notacja Θ

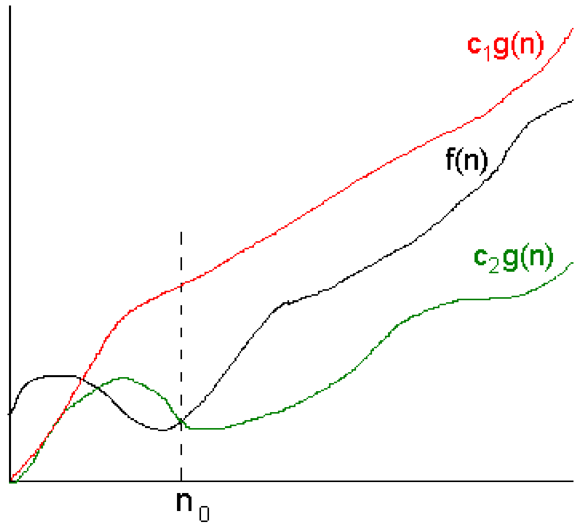
Notacja O

Notacja Ω

Notacja Θ

Dla danej funkcji *g(n)* przez ***Θ(g(n))*** *(„duże theta od g od n")* oznaczamy zbiór funkcji:

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

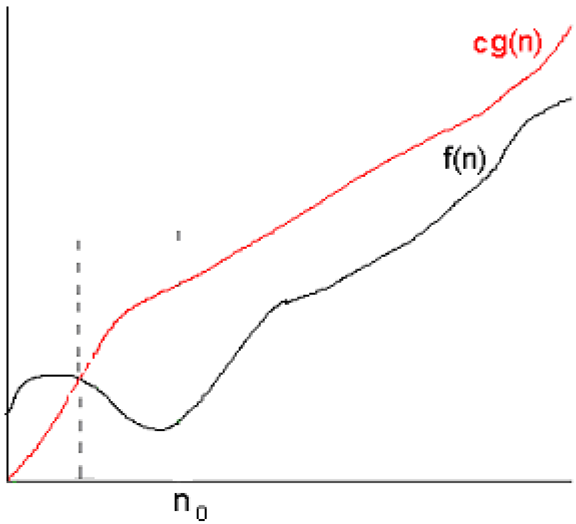


Notacja O

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

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

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

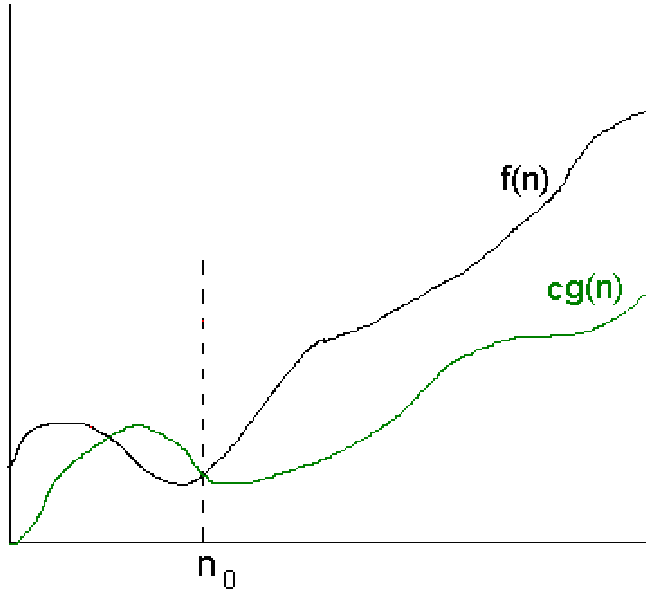


f(n)=O(g(n))

Notacja Ω

Notacja Ω asymptotycznie ogranicza funkcję od dołu.

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



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

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:

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

Analiza funkcji rekurencyjnych

Rekurencja polega na wywoływaniu danego podprogramu w jego treści.

Rekurencja:

- Bezpośrednia (podprogram wywołuje sam siebie)

- Pośrednia (podprogramy wywołują się naprzemiennie)

W języku C – rekurencyjne wywoływanie funkcji

Rekurencja

- Każde wywołanie funkcji tworzy nowe zmienne – często o nazwach już istniejących.
  Ze względu na ich lokalny zasięg, nie występują konflikty.

- Konieczność poprawnego zdefiniowania warunku zakończenia pętli – groźba zapętlenia

Rekurencja

- Rekurencji należy unikać, jeżeli istnieje rozwiązanie iteracyjne. Może się jednak zdarzyć, że skomplikowane rozwiązanie iteracyjne działa o wiele gorzej (dłużej się wykonuje, potrzebuje więcej pamięci itd…) niż rozwiązanie iteracyjne.

- Algorytmy, które w swej istocie są rekurencyjnie, nie iteracyjne, powinny być zaimplementowane jako funkcje rekurencyjne

Rekurencja -silnia

- Silnia dla liczby nieujemnej, całkowitej, zdefiniowana jest jako:

$$n! = \begin{cases} 1 & n = 0 \\ \prod_{i=1}^{k} k & n > 0 \end{cases}$$

- Rekurencyjny wzór na obliczenie n! zapisuje się jako: n!=n*(n-1)!

- Ze wzoru wynika, że aby obliczyć np. 4!, należy najpierw obliczyć 3!. Ale żeby obliczyć 3! trzeba obliczyć 2! itd. aż dojdziemy do 0!, które wynosi 1.

- Sposób obliczenia 4! wygląda więc następująco:
  4!=4*3!=4*3*2!=4*3*2*1!=4*3*2*1*0!=4*3*2*1*1=24

Rekurencja -silnia

- Definicja rekurencyjna n!:

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

Rekurencyjna definicja silni prowadzi do następującego algorytmu:

Rekurencja -silnia

- Definicja rekurencyjna n!:

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

Rekurencyjna definicja silni prowadzi do następującego algorytmu:

```
unsigned int Factorial (unsigned int n)
{
(1) if (n == 0)
(2)    return 1;
    else
(3)    return n * Factorial (n - 1);
}
```

Analiza funkcji rekurencyjnych-silnia

Analiza złożoności:

Analiza funkcji rekurencyjnych-silnia

Analiza złożoności:

instr	n=0	n>0
1	*O(1)*	*O(1)*
2	*O(1)*	-
3	-	*T(n-1)+O(1)*
razem	*O(1)*	*T(n-1)+O(1)*

Z tabeli wynika, że złożoność czasowa rekurencyjnego algorytmu *Factorial* dana jest rekurencyjnym wzorem:

$$T(n) = \begin{cases} O(1) & n = 0 \\ T(n-1) + O(1) & n > 0 \end{cases}$$

Analiza funkcji rekurencyjnych-silnia

Jak rozwiązać takie równanie?

Pominiemy O(.), rozwiążemy równanie i wstawimy O(.).

Rozwiązujemy więc równanie:

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + 1 & n > 0 \end{cases}$$

Jest to tak zwane równanie rekurencyjne.

Analiza funkcji rekurencyjnych-silnia

Analiza funkcji rekurencyjnych-silnia

Najprostszą metodą rozwiązywania równań rekurencyjnych jest metoda powtórzonych podstawień (nazywana też **rozwinięciem równania do sumy**):

1. Rozpisujemy równania jawnie dla wszystkich *n*.

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

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

.

.

.

$$T(1) = T(0) + 1$$

$$T(0) = 1$$

2. Po podstawieniach otrzymujemy:

$$T(n) = T(n-1) + 1 = T(n-2) + 1 + 1 = T(0) + n = 1 + n$$

czyli rekurencyjny algorytm obliczania silni jest klasy *O(n)*

Analiza złożoności – równania rekurencyjne

Wyznaczenie złożoności algorytmu sprowadza się często do rozwiązania równania rekurencyjnego.

Mamy następujące równanie rekurencyjne:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor n/2 \rfloor) + c & n > 1 \end{cases}$$

(równanie to otrzymujemy jako równanie złożoności wtedy, kiedy problem rozmiaru *n* sprowadza się do problemu o połowę mniejszego)

Trudno jest w tym przypadku zastosować rozwinięcie równania do sumy.

Zastosujemy zmianę zmiennych: podstawiamy $n = 2^k$

Analiza złożoności – równania rekurencyjne

Analiza złożoności – równania rekurencyjne

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor n/2 \rfloor) + c & n > 1 \end{cases}$$

Trudno jest w tym przypadku zastosować rozwinięcie równania do sumy.

Zastosujemy zmianę zmiennych: podstawiamy $n = 2^k$

$$T(2^k) = T(2^k/2) + c = T(2^{k-1}) + c$$

Teraz możemy zastosować metodę rozwinięcia równania do sumy:

$$T(2^k) = T(2^{k-1}) + c = T(2^{k-2}) + c + c = T(2^0) + kc = kc = c\log n$$

Stąd wynika, że

T(n)=O(log$n$)

Analiza złożoności – równania rekurencyjne

Mamy następujące równanie rekurencyjne:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + c & n > 1 \end{cases}$$

(równanie to otrzymujemy jako równanie złożoności wtedy, kiedy problem rozmiaru *n* sprowadza się do dwóch podproblemów rozmiaru *n/2* +stała liczba działań)

podobnie jak poprzednio, trudno jest w tym przypadku zastosować rozwinięcie równania do sumy.

Zastosujemy zmianę zmiennych: podstawiamy $n = 2^k$

$$T(2^k) = T(2^k/2) + T(2^k/2) + c = 2T(2^{k-1}) + c$$

Analiza złożoności – równania rekurencyjne

Analiza złożoności – równania rekurencyjne

Teraz możemy zastosować metodę rozwinięcia równania do sumy:

$$T(2^k) = 2T(2^{k-1}) + c = 2\left(2T(2^{k-2}) + c\right) + c = 2^2\left(T(2^{k-2})\right) + 2c + c =$$

$$2^k T(2^0) + c\left(2^{k-1} + 2^{k-2} + 2^0\right) = 2^k T(1) + c\left(2^{k-1} + 2^{k-2} + 2^0\right) =$$

$$c\left(2^{k-1} + 2^{k-2} + 2^0\right) = c\frac{2^{k-1} + 2^0}{2-1} = c(2^k - 1) = c(n-1)$$

Stąd wynika, że

$$T(n) = O(n)$$

Analiza złożoności – równania rekurencyjne

Mamy następujące równanie rekurencyjne:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + cn & n > 1 \end{cases}$$

(równanie to otrzymujemy jako równanie złożoności wtedy, kiedy problem rozmiaru *n* sprowadza się do dwóch podproblemów rozmiaru *n/2* +liniowa liczba działań - np. MERGESORT)

podobnie jak poprzednio, trudno jest w tym przypadku zastosować rozwinięcie równania do sumy.

Zastosujemy zmianę zmiennych: podstawiamy $n = 2^k$

$$T(2^k) = T(2^k/2) + T(2^k/2) + c2^k = 2T(2^{k-1}) + c2^k$$

Analiza złożoności – równania rekurencyjne

Analiza złożoności – równania rekurencyjne

Teraz możemy zastosować metodę rozwinięcia równania do sumy:

$$T(2^k) = 2T(2^{k-1}) + c2^k = 2\left(2T(2^{k-2}) + c2^{k-1}\right) + c2^k =$$

$$2^2T\left(2^{k-2}\right) + c2^k + c2^k = 2^2T\left(2^{k-2}\right) + 2c2^k =$$

$$= 2^kT\left(2^0\right) + kc2^k = 0 + cn\log(n)$$

Stąd wynika, że

$$T(n) = \Theta(n\log n)$$

Obliczanie wartości wielomianu w punkcie:

Mamy **tablicę współczynników** wielomianu **A[0..n]**:

A[0] – a_0

A[1] – a_1

…

A[n] – a_n

Gdzie $\boxed{W(x,n) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n}$

Chcemy obliczyć wartość wielomianu w punkcie x.

Obliczanie wartości wielomianu w punkcie:

$$W(x,0) = a_0$$

$$W(x,1) = a_0 + a_1 x = W(x,0) + a_1 x$$

$$W(x,2) = a_0 + a_1 x + a_2 x^2 = W(x,1) + a_2 x^2$$

$$W(x,n) = W(x,n-1) + a_n x^n$$

Obliczanie wartości wielomianu - algorytm rekurencyjny:

$$\boxed{W(x,n) = W(x,n-1) + a_n x^n}$$

```
Double W(double x,int n)
{
(1)If (n==0)
(2)  return a[0]
(3)else return W(x,n-1)+a[n]*P(x,n)
}
```

gdzie P(x,n) podnosi x do potęgi n

Analiza funkcji rekurencyjnych-
obliczanie wartości wielomianu

Analiza funkcji rekurencyjnych- obliczanie wartości wielomianu

Analiza złożoności:

inst..	n=0	n>0
1	*O(1)*	*O(1)*
2	*O(1)*	-
3	-	*T(n-1)+O(n)*
razem	*O(1)*	*T(n-1)+O(n)*

Z tabeli wynika, że złożoność czasowa rekurencyjnego algorytmu *W()* dana jest rekurencyjnym wzorem:

$$T(n) = \begin{cases} O(1) & n = 0 \\ T(n-1) + O(n) & n > 0 \end{cases}$$

Analiza funkcji rekurencyjnych- obliczanie wartości wielomianu

Pominiemy O(.), rozwiążemy równanie i wstawimy O(.).

Rozwiązujemy więc równanie:

$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + n & n > 0 \end{cases}$$

Po podstawieniach otrzymujemy:

T(n)= *O(n^2)*

Obliczanie wartości wielomianu

- Schemat Hornera

Przyjmijmy, że:

$$W(x,n) = a_n + a_{n-1}x + a_{n-2}x^2 + \ldots + a_0x^n$$

$$W(x,n) = a_n + x(a_{n-1} + x(a_{n-2} + \ldots x(a_1 + a_0x)\ldots)$$

Przykładowo:

$$W(x,2) = a_2 + a_1x + a_0x^2 = a_2 + x(a_1 + xa_0)$$

$$W(x,3) = a_3 + a_2x + a_1x^2a_0x^3 = a_3 + x(a_2 + x(a_1 + xa_0))$$

Obliczanie wartości wielomianu

- Schemat Hornera

$$\boxed{W(x,n) = a_n + x(a_{n-1} + x(a_{n-2} + \ldots x(a_1 + a_0x)\ldots)}$$

Oznaczmy:

$$T(x,0) = a_0$$

$$T(x,1) = a_0x + a_1 = T(x,0)x + a_1$$

$$T(x,2) = (a_0x + a_1)x + a_2 = T(x,1)x + a_2$$

$$T(x,n) = a_n + x(a_{n-1} + xa_{n-2} + \ldots + x(a_1 + xa_0)\ldots) = a_n + xT(x,n-1)$$

Schemat Hornera – algorytm rekurencyjny

```
Double Horner(double x, int n)
{
If (n==0) return a[0] else
return Horner(x,n-1)*x+a[n]
}
```

$T(n)=O(n)$

(analiza złożoności analogiczna do analizy funkcji Factorial)

Analiza funkcji rekurencyjnych- liczby Fibonacciego

Ciąg Fibonacciego, to ciąg liczb spełniających:

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$

Kolejne elementy tego ciągu to: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,...

Kolejnym elementem ciągu jest suma dwóch poprzednich elementów.

Nierekurencyjna wersja obliczania liczb Fibonacciego

Nierekurencyjna wersja obliczania liczb Fibonacciego

```
unsigned int Fibonacci (unsigned int n)
 {
      int previous = -1;
      int result = 1;
      for (unsigned int i = 0; i <= n; ++i)
      {
      int const sum = result + previous;
      previous = result;
      result = sum;
       }
       return result;
   }
```

Nierekurencyjna wersja obliczania liczb Fibonacciego

5. Instrukcje (2), (3) są *O(1)*. Z reguły sumowania złożoność fragmentu (2)(3) jest *O(max(1,1))=O(1)*
6. Instrukcje (5), (6), (7) są rzędu *O(1)*. Z reguły sumowania złożoność fragmentu (5)(6)(7) jest *O(max(1,1,1))=O(1)*
7. Dla pętli (4)-(7) 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+1*. Zatem na podstawie reguły mnożenia złożoność fragmentu (4)-(7) jest *O((n+1)*1)=O(n+1)*.

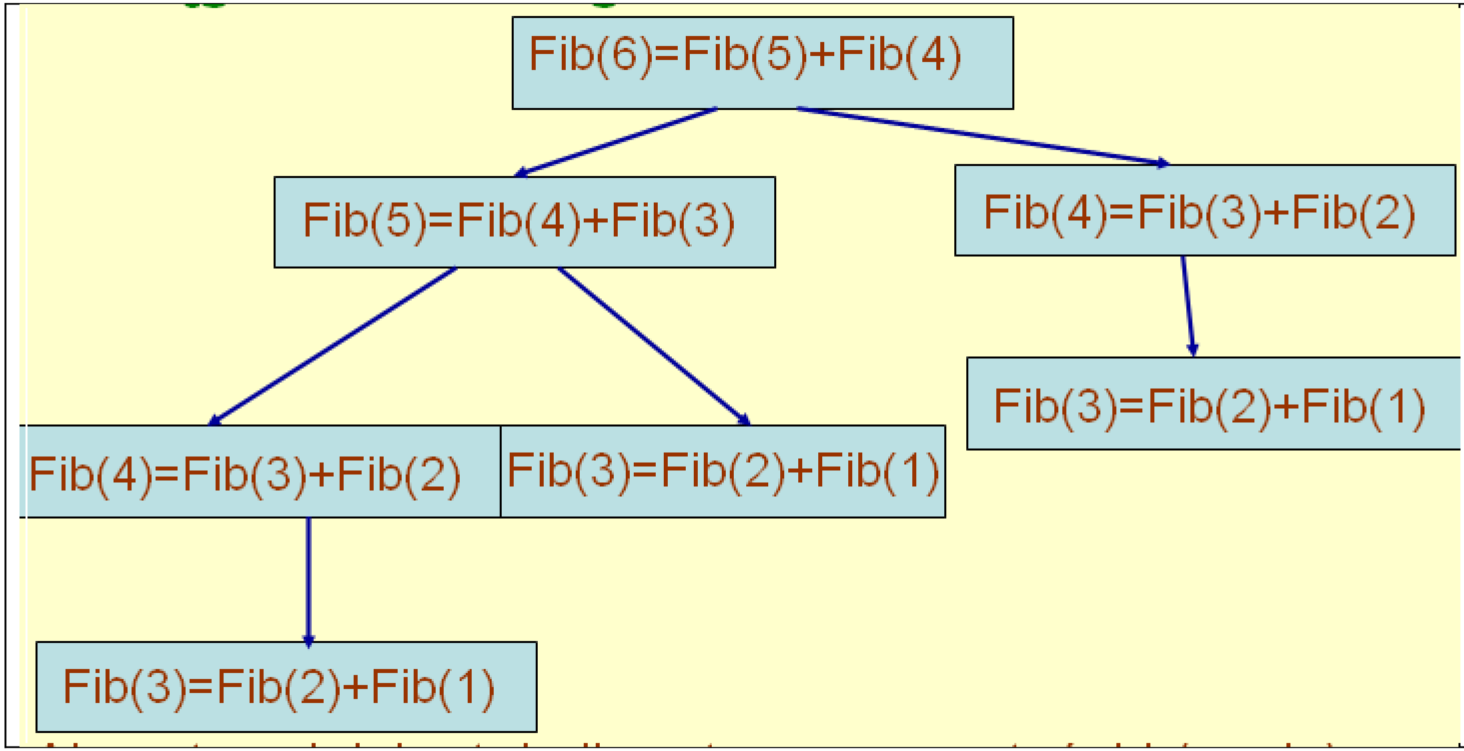
Nierekurencyjna wersja obliczania liczb Fibonacciego

8. Ponieważ złożoność czasowa fragmentu (2)(3) jest rzędu złożoność(1), a złożoność pętli (4)-(7) jest rzędu *O(n+1),* wiec z reguły sumowania złożoność całego algorytmu jest *O(n+1)* czyli *O(n)*

Rekurencyjna wersja obliczania liczb Fibonacciego

Rekurencyjna wersja obliczania liczb Fibonacciego

```
unsigned int Fibonacci (unsigned int n)
{(1)    if (n == 0 || n == 1)
 (2)   return n;
    else
(3) return Fibonacci (n - 1) + Fibonacci (n - 2);}
```



Algorytm wielokrotnie liczy te same wartości (wada!)

Rekurencyjna wersja obliczania liczb Fibonacciego

```
unsigned int Fibonacci (unsigned int n)
{
(1)     if (n == 0 || n == 1)
(2) return n;
     else
(3) return Fibonacci (n - 1) + Fibonacci (n - 2);
}
```

instr	N<2	n>=2
1	*O(1)*	*O(1)*
2	*O(1)*	-
3	-	$T(n-1)+T(n-2)+O(1)$
razem	*O(1)*	$T(n-1)+T(n-2)+O(1)$

Rekurencyjna wersja obliczania liczb Fibonacciego

Z tabeli wynika, że złożoność czasowa rekurencyjnego algorytmu Fibonacci dana jest rekurencyjnym wzorem:

$$T(n) = \begin{cases} O(1) & n < 2 \\ T(n-1) + T(n-2) + O(1) & n \geq 2 \end{cases}$$

Jak rozwiązać takie równanie?

Pominiemy O(.), rozwiążemy równanie i wstawimy O(.).

Rozwiązujemy więc równanie:

$$T(n) = \begin{cases} 1 & n < 2 \\ T(n-1) + T(n-2) + 1 & n \geq 2 \end{cases}$$

Rekurencyjna wersja obliczania liczb Fibonacciego

Etap 1: Pokażemy przez indukcję, że:

$$\forall\, n \geq 0\; T(n) \geq F_{n+1}$$

Dowód:

1. n=0: $T(0) = 1 \geq F_1 = 1$

   n=1: $T(1) = 1 \geq F_2 = 1$

2. założenie indukcyjne:

$$T(i) \geq F_{i+1} \forall\, i \leq k \quad \text{dla pewnego k>=1}$$

3. $T(k+1) \geq F_{k+2}$?

$$T(k+1) = T(k) + T(k-1) \geq F_{k+1} + F_k + 1 \geq F_{k+2} + 1 \geq F_{k+2}$$

Udowodniliśmy wiec indukcyjnie, że $\forall\, n \geq 0\; T(n) \geq F_{n+1}$

Tak więc $T(n) = \Omega\,(F_{n+1})$. Czy to dobra złożoność?

Rekurencyjna wersja obliczania liczb Fibonacciego

Na podstawie poniższego twierdzenia pokażemy, że wersja nierekurencyjna algorytmu *Fibonacii* jest dużo lepsza.

Twierdzenie 2.

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right) \text{ gdzie}$$

$$\phi = \left(1 + \sqrt{5}\right)/2$$

$$\phi = \left(1 - \sqrt{5}\right)/2$$

Rekurencyjna wersja obliczania liczb Fibonacciego

Dowód indukcyjny

1. n=0: $F_1 = \frac{1}{\sqrt{5}}(1 - 1) = 0$

   n=1: $F_1 = \frac{1}{\sqrt{5}}\left(\left(1 + \sqrt{5}\right)/2 - \left(1 - \sqrt{5}\right)/2\right) = \frac{1}{\sqrt{5}}\left(1 + \sqrt{5} - 1 + \sqrt{5}\right)/2 =$

   $\frac{1}{\sqrt{5}}\sqrt{5} = 1$

2. założenie indukcyjne:

$$F_i = \frac{1}{\sqrt{5}}\left(\phi^i - \hat{\phi}^i\right) \forall\, i \leq k \quad \text{dla pewnego k>=1}$$

Rekurencyjna wersja obliczania liczb Fibonacciego

3. $F_{k+1} = \frac{1}{\sqrt{5}}\left(\phi^{k+1} - \hat{\phi}^{k+1}\right)$?

Wskazówka:

$\phi^2 = \left(\left(1+\sqrt{5}\right)/2\right)^2 = 1/4 + \sqrt{5}/2 + 5/4 =$

$1 + (1+\sqrt{5})/2 = 1 + \phi$

$\hat{\phi}^2 = \left(\left(1-\sqrt{5}\right)/2\right)^2 = 1/4 - \sqrt{5}/2 + 5/4 =$

$1 + (1-\sqrt{5})/2 = 1 + \hat{\phi}$

Rekurencyjna wersja obliczania liczb Fibonacciego

$$F_{k+1} = F_k + F_{k-1} =$$

$$\frac{1}{\sqrt{5}}\left(\phi^{k} - \hat{\phi}^{k}\right) + \frac{1}{\sqrt{5}}\left(\phi^{k-1} - \hat{\phi}^{k-1}\right) =$$

$$\frac{1}{\sqrt{5}}\left(\phi^{k-1}(1+\phi) - \hat{\phi}^{k-1}\left(1+\hat{\phi}\right)\right) =$$

$$\frac{1}{\sqrt{5}}\left(\phi^{k-1}\phi^{2} - \hat{\phi}^{k-1}\hat{\phi}^{2}\right) =$$

$$\frac{1}{\sqrt{5}}\left(\phi^{k+1} - \hat{\phi}^{k+1}\right)$$

Na podstawie Twierdzenia 2 mamy:

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right) \text{ gdzie } \begin{array}{l} \phi = \left(1+\sqrt{5}\right)/2 \\ \hat{\phi} = \left(1-\sqrt{5}\right)/2 \end{array}$$

Rozważmy $\hat{\phi}$. Ponieważ $\hat{\phi} = \left(1-\sqrt{5}\right)/2$, więc $\left|\hat{\phi}\right| < 1$.

Im większe *n*, tym mniejsze $\left|\hat{\phi}^n\right|$.

Tak więc dla odpowiednio dużych *n* mamy:

$F_n = \Omega\left(\phi^n\right)$.

Ponieważ $\phi^n \approx 1.62 > 3/2$ mamy:

$F_n=\Omega((3/2)^n)$ - czas wykonania rekurencyjnej funkcji Fibonacci rośnie **wykładniczo** wraz ze wzrostem *n!*