

Algorytmy i Struktury Danych, 1. ćwiczenia

2023-10-02

Zadanie 1.1

```
Algorytm A
begin
  s* := 0;
  for i in [1..n] do
    for j in [i..n] do begin
      s := 0;
      for k in [i..j] do
        s := s + a[k]; {operacja dominująca}
      s* := MAX(s*, s)
    end
  end
end
```

Ile dokładnie operacji dominujących zostanie wykonanych w algorytmie A?

$$\begin{aligned} \sum_{1 \leq i \leq j \leq n} |j-i+1| &= \sum_{l=1}^n l \cdot (n+1-l) = \sum_{l=1}^n (ln+l-l^2) = \frac{n^2(n+1)}{2} + \frac{n(n+1)}{2} - \frac{n(n+1)(2n+1)}{6} \\ &= \frac{(n+1)(3n^2+3n-2n^2-n)}{6} = \frac{(n+1)(n^2+2n)}{6} \end{aligned}$$

w obliczeniach wykorzystujemy wzór:

$$\sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6}$$

https://en.wikipedia.org/wiki/Square_pyramidal_number

```
Algorytm B
begin
  s* := 0;
  for i in [1..n] do begin
    s := 0;
    for j in [i..n] do begin
      s := s + a[j]; {operacja dominująca}
      s* := MAX(s*, s)
    end;
  end
end
```

Ile dokładnie operacji dominujących zostanie wykonanych w algorytmie B?

$$\sum_{1 \leq i \leq n} (n+1-i) = \frac{n(n+1)}{2}$$

```

Algorytm C
begin
  s* := 0; p := 0;
  for i in [1..n] do begin
    p := p + a[i]; {operacja dominująca}
    s* := MAX(s*, p);
    if p < 0 then p := 0
  end
end

```

Udowodnij poprawność Algorytmu C podając stosowny niezmiennik pętli “for”.

Na początku każdego obrotu pętli, zmienna p zawiera maksymalną sumę spośród wszystkich sufiksów $A[1..(i-1)]$ (również pustego).

Jak długo będzie trwało wykonanie algorytmów dla $n = 1000^2$ przy założeniu, że w 1 sekundzie jest wykonywanych 1000^3 dodawań (operacji dominujących).

alg. A) $\frac{n^3}{6}$, czyli $\frac{1000^6}{6 \cdot 1000^3} \approx 166$ mln. sekund ≈ 5 lat

alg. B) $\frac{n^2}{2}$, czyli $\frac{1000^4}{2 \cdot 1000^3} = 500$ sekund ≈ 8 minut

alg. C) n , czyli $\frac{1000^2}{1000^3} = 0.001$ sekundy

Zadanie 1.2 Liczby Fibonacciego

Oblicz ile dodawań jest wykonywanych przy liczeniu F_n rekurencyjnie.

$$f(n) = \begin{cases} 0 & \text{dla } n \leq 1 \\ f(n-1) + f(n-2) + 1 & \text{wpp} \end{cases}$$

Odpowiedź:

- $f(n) = O(F_n)$ (ponieważ drzewo rekurencji będzie miało dokładnie F_n liści z wartością $1 = F_1$ oraz $\leq F_n$ liści z etykietą $0 = F_0$)
- a dokładniej $f(n) = F_{n+1} - 1$ (dowód przez indukcję)

n	F_n	liczba dodawań $f(n)$
0	0	0
1	1	0
2	1	1
3	2	2
4	3	4
5	5	7
6	8	12
7	13	20
8	21	33

Zaprojektuj algorytm obliczania liczby F_n wykonujący $O(\log n)$ operacji arytmetycznych, z wykorzystaniem wzoru rekurencyjnego: dla $n > 1$, $F_{2n-1} = F_n^2 + F_{n-1}^2$ oraz $F_{2n} = F_n^2 + 2F_nF_{n-1}$.

Algorytmy obliczające F_n używające $O(\log n)$ operacji arytmetycznych:

- obliczenie x^n można wykonać stosując $O(\log n)$ operacji arytmetycznych, ta sama zasada stosuje się do potęgowania macierzy
- uwaga! jeśli będziemy obliczać rzeczywisty koszt operacji arytmetycznych i stosować zwykły algorytm mnożenia liczb, to taki algorytm będzie miał złożoność $O(n^2)$ ponieważ F_n ma $O(n)$ bitów.

Funkcja FibMatrix(n)

if $n == 0$ **then**

return 0

 oblicz $M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$ używając $O(\log n)$ mnożeń macierzy

return $M[0][1]$

Zadanie 1.3 – rekurencyjne mnożenie

Podaj równanie rekurencyjne na koszt mnożenia liczb x , y i rozwiąż je.

Złożoność oryginalnego algorytmu (używającego 4 mnożeń):

$$T(n) = 4T(n/2) + O(n)$$

$$T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$$

Mnożenie dużych liczb

Funkcja Mult(a, b)

niech n oznacz długość liczb a, b

if $n \leq 1$ **then**

użyj zwykłego mnożenia

else

 dzielimy (tekstowo) a i b na pary dwóch krótszych liczb (o $n/2$ cyfrach)

 niech $a = a_1 a_2$ ($|a_1| = |a_2| = n/2$)

 niech $b = b_1 b_2$ ($|b_1| = |b_2| = n/2$)

$A = \text{mult}(a_1, b_1)$

$B = \text{mult}(a_2, b_2)$

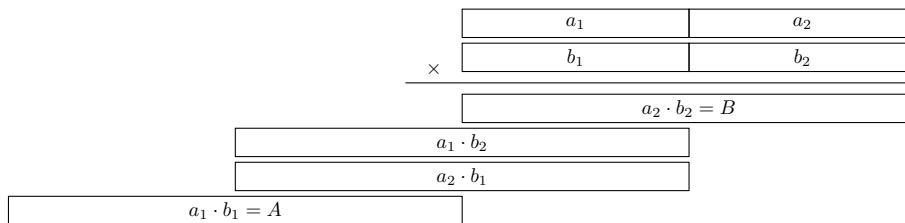
$C = \text{mult}(a_1 + a_2, b_1 + b_2)$

$D = C - (A + B)$ (co jest równoważne $D = a_1 b_2 + a_2 b_1$)

return $A * 10^n + D * 10^{n/2} + B$;

Złożoność algorytmu:

$$T(n) = 3T\left(\frac{n}{2}\right) + O(n) = O(n^{\log_2 3}) \approx O(n^{1.58496})$$



Rysunek 1: Mnożenie dużych liczb

Zadanie 1.4 – pary

```
Suma(S)::
begin
  z := 0; koszt := 0;
  while |S| != 1 do begin
    (x,y) := Para(S);
    S := S \ {x,y};
    S := S + {x+y};
    koszt := koszt + x + y;
  end;
  return koszt
end;
```

Jak zaimplementować funkcję *Para* (zwracającą parę dowolnych elementów z S), żeby zminimalizować wartość *koszt*?

Optymalną strategią dla funkcji *Para* jest zawsze wybór dwóch najmniejszych wartości z S . Dowód poprawności ten sam co w Huffman Coding https://en.wikipedia.org/wiki/Huffman_coding.

Jak zaimplementować funkcję *Para* (zwracającą parę sąsiednich elementów z ciągu S , zakładamy, że nowy element $x + y$ zastępuje usuwaną parę z S), żeby zminimalizować wartość *koszt*?

Dla ciągu $S = a_1, \dots, a_n$, definiujemy pomocnicze sumy częściowe $Z[i, j] = \sum_{k=i}^j a_k$ (można je efektywnie wyliczać np. przez sumy prefiksowe ciągu). Optymalną wartość *koszt* można obliczyć używając programowania dynamicznego:

$$Opt[i, i] = 0$$

$$Opt[i, j] = Z[i, j] + \min_{i \leq k < j} (Opt[i, k] + Opt[k + 1, j])$$

Zadanie 1.5 – sortowanie przez wstawianie

Dokonaj analizy pesymistycznej złożoności obliczeniowej tego algorytmu dla następujących przypadków:

a) $|a[i] - a[j]| < 2020$, dla każdej pary $1 \leq i, j \leq n$ takiej, że $|i - j| < 2020$

Odpowiedź: $O(n^2)$ ponieważ ciąg odwrotnie uporządkowany spełnia warunki.

b) $|i - a[i]| < 2020$, dla każdego $1 \leq i \leq n$

Odpowiedź: $O(n)$ ponieważ liczba inwersji w a jest $\leq 4040n$ (dla $j - i \geq 4040$ możemy udowodnić, że $a[i] \leq a[j]$).

c) dla co najwyżej 2020 elementów zachodzi $i \neq a[i]$, $1 \leq i \leq n$

Odpowiedź: $O(n)$ ponieważ liczba inwersji w a jest $\leq 2020n$ (każda inwersja zawiera jakiś element $i \neq a[i]$).