

# Algorytmy i Struktury Danych, 2. ćwiczenia

2024-10-16

## Zadanie 2.1

---

**Algorytm 1:** Sortowanie Szymka R.

---

$X := \{x_1, x_2, \dots, x_n\}$   
Zainicjuj stos  $S$  jako pusty  
**while**  $Not\ Empty(X)$  **do**  
    Weź dowolny element  $x$  ze zbioru  $X$   
     $X := X - \{x\}$   
    Usuń ze stosu  $S$  wszystkie elementy większe od  $x$  i wstaw je z  
    powrotem do zbioru  $X$   
    Umieść  $x$  na wierzchołku stosu  $S$   
wypisz kolejno elementy ze stosu  $S$

---

Przyjmijmy, że operacjami dominującymi są operacje stosowe TOP, PUSH, POP.

a) Jaka jest pesymistyczna złożoność sortowania algorytmem Szymka R.?

**Rozwiązanie:** Bez straty ogólności możemy potraktować ciąg  $\{x_1, \dots, x_n\}$  jako permutację  $\{1, \dots, n\}$ .

Jeśli potraktujemy zawartość stosu jako licznik binarny (1 jeśli element jest na stosie, 0 wpp, przy czym najstarszy bit odpowiada 1, a najmłodszy bit odpowiada  $n$ ), to możemy zauważyć, że każdy obrót pętli zwiększa licznik. Czyli jeśli na stos zostanie element minimalny, to już tam zostaje.

Niech  $T(n)$  oznacza pesymistyczną złożoność czasu sortowania dla  $n$  elementów. Najgorszym scenariuszem, jest ten w którym element 1 zostanie wylosowany dopiero wtedy, gdy nie będzie już innych elementów do wyboru, czyli na stosie znajdują się wszystkie pozostałe  $n - 1$  elementów. W momencie gdy wylosujemy element 1 musimy wykonać  $n - 1$  operacji TOP,  $n - 1$  operacji POP i 1 operację PUSH (razem  $2n - 1$  operacji). Następnie problem zostaje zredukowany do posortowania pozostałych  $n - 1$  elementów.

Taką sytuację możemy wyrazić wzorem:

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

$n$	$T(n)$
1	1
2	5
3	15
4	37
5	83

$$T(n) = 2^k T(n-k) + \sum_{i=0}^{k-1} 2^i (2n-1-2i)$$

$$T(n) = (2n-1) \cdot (2^n - 1) - 2 \sum_{i=1}^{n-1} (i \cdot 2^i)$$

Korzystając z wzoru  $\sum_{i=1}^n i \cdot 2^i = 2 + 2^{n+1}(n-1)$ :

$$T(n) = (2n-1) \cdot (2^n - 1) - 2(2 + 2^n(n-2))$$

$$T(n) = n2^{n+1} - 2n - 2^n + 1 - 4 - 2^{n+1}(n-2)$$

$$T(n) = 2^{n+2} - 2n - 2^n - 3$$

Czyli  $T(n) = O(2^n)$ .

b) Załóżmy teraz, że element  $x$  wybieramy losowo ze zbioru  $X$  z rozkładem jednostajnym - każdy element może zostać wylosowany z prawdopodobieństwem  $1/|X|$ . Udowodnij, że oczekiwana liczba losowań elementu  $x$  w algorytmie wynosi  $O(n^2)$ .

**Rozwiązanie:**

- udowodnijmy, że po  $O(n)$  losowania element maksymalny trafi na spód stosu

Niech  $Y$  to zmienna losowa oznaczająca liczbę losowań w algorytmie do momentu w którym zostanie wylosowane MAX. Dodatkowo wprowadźmy zmienną losową  $Y'$  oznaczającą liczbę losowań w procesie Bernoulliego do pierwszego sukcesu przy założeniu, że pojedynczym losowaniu prawdopodobieństwo sukcesu to  $p = \frac{1}{n}$ .

Chcemy oszacować (z góry)  $E[Y]$ , jednak ze względu na stopień złożoności algorytmu policzenie dokładnej wartości  $E[Y]$  nie jest łatwe. Możemy natomiast zauważyć, że  $E[Y] \leq E[Y']$  — w każdej turze algorytmu prawdopodobieństwo wylosowania MAX może się zmieniać, jednak zawsze  $\geq \frac{1}{n}$  stąd "łatwiej/szybciej" osiągnąć sukces w  $Y$  niż  $Y'$ .

Ponieważ  $E[Y'] = 1/p = n$  stąd  $E[Y] \leq n$ .

**Fakt 1.**  $E[Y'] = 1/p$

$$\begin{aligned} E[Y'] &= p + 2(1-p)p + 3(1-p)^2p + \dots \\ (1-p)E[Y'] &= (1-p)p + 2(1-p)^2p + 3(1-p)^3p + \dots \end{aligned}$$

po odjęciu stronami:

$$\begin{aligned} pE[Y'] &= p + (1-p)p + (1-p)^2p + \dots \\ E[Y'] &= \sum_{i=0}^{\infty} (1-p)^i = \frac{1}{1-(1-p)} = \frac{1}{p} \end{aligned}$$

- ponieważ po  $n$  wyborach minimum ciąg zostanie posortowany, stąd oczekiwana liczba losowań to  $O(n^2)$ .

c) *Dokonaj analizy oczekiwanej liczby operacji dominujących w algorytmie Szymka R. w opisanym wyżej modelu probabilistycznym.*

**Rozwiązanie:** Niech #POP, #PUSH, #TOP odpowiednie liczby operacji:

- #PUSH jest równa liczbie losowań, więc z punktu b) wynika, że oczekiwana liczba operacji wynosi  $O(n^2)$ ,
- #POP  $\leq$  #PUSH ponieważ, żeby zdjąć element ze stosu, to trzeba go tam wcześniej włożyć,
- #TOP  $\leq$  #POP + #PUSH,
- stąd oczekiwana wartość #POP + #PUSH + #TOP jest rzędu  $O(n^2)$ .

## Zadanie 2.2 - sortowanie ciągów $k$ -dobrych

Niech  $n$  będzie dodatnią liczbą całkowitą. Dla dodatniej liczby całkowitej  $k$  powiemy, że ciąg liczb  $a[1], \dots, a[n]$  jest  $k$ -dobry, jeśli każda inwersja  $(i, j)$ ,  $1 \leq i < j \leq n$ , spełnia  $j \leq i + k$ .

- [8 punktów] zaproponuj asymptotycznie optymalny ze względu na porównania algorytm sortujący ciągi  $k$ -dobre. Uzasadnij asymptotyczną optymalność swojego algorytmu. Uwaga: w tym zadaniu argumentami funkcji złożoności są  $k$  i  $n$ .
- [5 punktów] zaproponuj efektywny czasowo i pamięciowo algorytm, który sprawdza czy ciąg liczb  $a[1], \dots, a[n]$  dla zadanej liczby całkowitej  $k$ , jest  $k$ -dobry. Uzasadnij poprawność algorytmu i dokonaj analizy czasowej i pamięciowej

**Rozwiązanie:** (a) Dowolny algorytm sortujący oparty o porównania musi wykonać  $\Omega(n \log k)$  porównań.

Rozważmy permutacje postaci  $\pi = (B_1, B_2, \dots, B_{n/k})$ , gdzie  $B_i$  to dowolna permutacja liczb  $\{(i-1)k+1, \dots, ik\}$  (w szczególności  $B_1 = 1..k, B_2 = k+1..2k$ ).

Oczywiście wszystkie takie permutacje są  $k$ -dobre (inwersje występują tylko wewnątrz bloków). Tego typu permutacji jest  $(k!)^{n/k}$ .

Algorytm (dla dowolnych ciągów  $k$ -dobrych):

sortuj kolejno bloki długości  $2k$  zaczynające się na pozycjach  $1, k+1, 2k+1, \dots$

(b) Niech  $pmax[i]$  oznacza  $\max a[1..i]$  (maksymalne wartości dla wszystkich prefiksów tablicy, uwaga!  $pmax[i]$  nie zawiera  $a[i]$ ). Tablica  $a$  jest  $k$ -dobra, jeśli

$$\forall_{k \leq j \leq n} pmax[j-k] \leq a[j]$$

## Zadanie 2.3 - przesunięcie cykliczne

---

**Funkcja** CyclicLeftShift( $a, k$ )

---

```
 $n := \text{len}(a)$   
Reverse( $a, 1, n - k$ )  
Reverse( $a, n - k + 1, n$ )  
Reverse( $a, 1, n$ )
```

---

## Zadanie 2.4a - stabilne sortowanie ciągów 0/1

---

**Algorytm 2:** StableMergeSort01( $A$ )

---

```
Input: array  $A[1..n]$   
Output: number of elements with value 0  
if  $n = 1$  then  
| return (1 if  $A[1] = 0$  else 0)  
else  
|  $m = \lfloor \frac{n}{2} \rfloor$   
|  $c_1 = \text{StableMergeSort01}(A[1, \dots, m])$   
|  $c_2 = \text{StableMergeSort01}(A[m + 1, \dots, n])$   
| if  $c_1 < m$  and  $c_2 > 0$  then  
| | Swap( $A[c_1 + 1, \dots, m], A[m + 1, \dots, m + c_2]$ ) – przy użyciu  
| | CyclicShift  
| return  $c_1 + c_2$ 
```

---

Złożoność czasowa:  $T(n) = 2T(n/2) + O(n) = O(n \log n)$ . Złożoność pamięciowa:  $O(\log n)$  (ale można to też zapisać w pamięci  $O(1)$ ).

## Zadanie 2.4b

Dana jest  $2n$ -elementowa tablica zawierająca  $n$  zer i  $n$  jedynek. Chcemy ją uporządkować tak, żeby zera i jedynki były ułożone na przemian, począwszy od zera, tj. 010101... Zaproponuj efektywny algorytm, który wykona to w miejscu i stabilnie (tj. kolejność zer i kolejność jedynek z wejścia muszą być zachowane).

**Rozwiązanie:** Posortuj stabilnie (ala MergeSort) a następnie rekurencyjnie poprzeplataj.

---

**Algorytm 3:** Sort( $A$ )

---

```
if  $|A| \geq 2$  then  
|  $(Z_l, O_l) = \text{Sort}(A[1..n/2])$   
|  $(Z_r, O_r) = \text{Sort}(A[n/2 + 1..n])$   
| Exchange( $O_l, Z_r$ )  
| return  $(Z_l + Z_r, O_l + O_r)$   
else  
| return  $(A, \emptyset)$  (if  $A=[0]$ ) or  $(\emptyset, A)$  otherwise
```

---

---

**Algorytm 4: Unpack( $A$ )**

---

```
if  $|A| > 2$  then
     $l = \lfloor |A|/4 \rfloor$ ;  $r = \lceil |A|/4 \rceil$  ;
    // zamień ciąg  $0^{\lfloor |A|/2 \rfloor} 1^{\lfloor |A|/2 \rfloor}$  na  $0^l 1^l 0^r 1^r$ 
    Exchange( $A[(l+1)..2l]$ ,  $A[(2l+1)..(2l+r)]$ )
    Unpack( $A[1..2l]$ )
    Unpack( $A[(2l+1)..n]$ )
```

---

## Zadanie 2.5, 2.6 i 2.7 - merge w miejscu

Scalanie w miejscu dla ciągów długości  $\sqrt{n}$  i  $n - \sqrt{n}$

---

**Algorytm 5: Merge( $A$ )**

---

Dana jest tablica  $A$  zawierająca dwa uporządkowane rosnąco ciągi:  
 $1..n - \sqrt{n}$  i  $n - \sqrt{n} + 1..n$ .  
Posortuj (używając alg. insertion sort) ciąg  $n - 2\sqrt{n} + 1..n$   
Scal ciąg  $1..n - 2\sqrt{n}$  i  $n - 2\sqrt{n} + 1..n - \sqrt{n}$  używając obszaru  
 $n - \sqrt{n} + 1..n$  jako bufor  
Posortuj (używając alg. insertion sort) ciąg  $n - \sqrt{n} + 1..n$

---

Scalanie dwóch uporządkowanych w ciągu przy użyciu bufora:

---

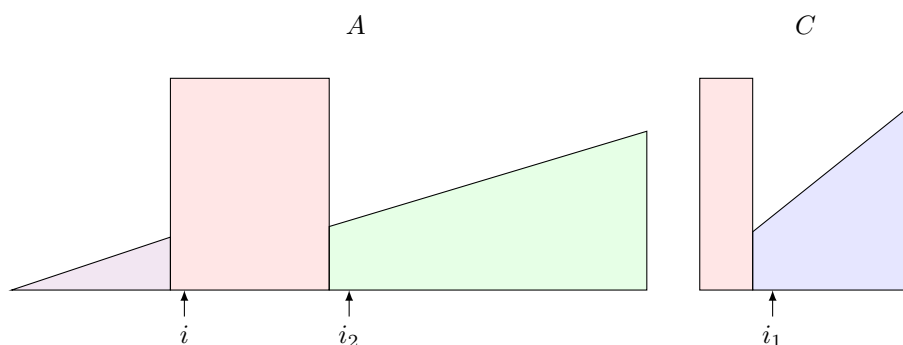
```
def merge( $A$ ,  $n1$ ,  $C$ ):
    """scalanie uporządkowanych  $A[0..n1)$  i  $A[n1..)$ 
    przy użyciu bufora  $C$  (długości  $\geq \min(n1, |A|-n1)$ ) """
    assert len( $C$ )  $\geq$  min( $n1$ , len( $A$ ) -  $n1$ )
     $n =$  len( $A$ )
    if  $n1 > n - n1$ :
        # zamień  $A[0..n1)$  i  $A[n1..)$  - wersja uproszczona!
        reverse = lambda arr: arr[::-1] # to nie jest  $O(1)$ 
         $A[:n1]$ ,  $A[n1:] =$  reverse( $A[:n1]$ ), reverse( $A[n1:]$ )
         $A[:n] =$  reverse( $A[:n]$ )
         $n1 = n - n1$ 

    # zamień  $A[0..n1)$  i  $C$ 
    for  $i$  in range( $n1$ ):
         $A[i]$ ,  $C[i] = C[i]$ ,  $A[i]$ 

     $i$ ,  $i1$ ,  $i2 = 0$ ,  $0$ ,  $n1$ 
    while  $i1 < n1$ :
        # fragment  $A[0:i)$  jest już uporządkowany
        # pozostałe elementy są w  $C[i1:n1)$  i  $A[i2:n)$ 
        if  $i2 == n$  or  $C[i1] \leq A[i2]$ :
             $A[i]$ ,  $C[i1] = C[i1]$ ,  $A[i]$ 
             $i1 += 1$ 
        else:
             $A[i]$ ,  $A[i2] = A[i2]$ ,  $A[i]$ 
             $i2 += 1$ 
```

$i += 1$

Uwaga! W tym kodzie dla uproszczenia zapisu użyłem do odwracania listy konstrukcji  $[::-1]$ , która używa dodatkowej pamięci, jednak łatwo to zastąpić prostą pętlą odwracająca dany zakres.



Rysunek 1: Przykładowy stan w trakcie uruchomienia algorytmu MERGE

## Scalanie w miejscu

Knuth, Tom III, strona 698.

- podziel tablicę na bloki rozmiaru  $\lceil \sqrt{n} \rceil$ , —  $Z_1, Z_2, \dots, Z_{m+2}$ , (blok  $Z_{m+2}$  może być mniejszy,
- zamień blok leżący na połączeniu dwóch ciągów, z blokiem  $Z_{m+1}$ , teraz każdy z bloków  $Z_1, \dots, Z_m$  jest uporządkowany,
- posortuj używając selection-sort bloki, wg. pierwszego elementu z bloków (jeśli dwa bloki mają ten sam element początkowy, to porównaj elementy końcowe)
- scal  $Z_1, \dots, Z_m$  używając  $Z_{m+1}$  jako bufora pomocniczego,

---

### Algorytm 6: Z-Merge( $Z$ )

---

```
foreach  $i \in 1, \dots, m - 1$  do  
  SimpleMerge( $Z_i, Z_{i+1}, Z_{m+1}$ )
```

---

(należy jeszcze pokazać, że taka procedura daje dobre uporządkowanie) — wskazówka: przed tym krokiem każdy element jest w inwersji z co najwyżej  $\sqrt{n}$  innymi elementami bloków  $Z_1, \dots, Z_{m+1}$

- dzielimy tablicę na trzy części:  $A, B, C$ ,  $|B| = |C| = 2\lceil \sqrt{n} \rceil$
- posortuj ostatnie  $4 \cdot \lceil \sqrt{n} \rceil$  elementów (bloki  $B, C$ ) używając InsertionSort (w rezultacie w bloku  $C$  znajdują się największe elementy w tablicy)
- scal bloki  $A$  i  $B$  używając  $C$  jako bufora

- posortuj blok  $C$  używając InsertionSort

Ćwiczenie: dlaczego używając selection-sort trzeba uwzględniać początki i końce bloków?

Na przykład dla ciągów (111,123),(111,145) (rozmiar bloku 3), sortując jedynie po początkach moglibyśmy otrzymać: (123,145,111,111), który przy scalaniu metodą opisaną w algorytmie nie da uporządkowanego ciągu.

Inne wyniki: istnieje również trochę bardziej skomplikowany algorytm, który pozwala na scalanie dwóch uporządkowanych ciągów w czasie liniowym, w stałej pamięci i dodatkowo **stabilnie** ([1]).

## Bibliografia

- [1] Viliam Geffert, Jyrki Katajainen i Tomi Pasanen. “Asymptotically efficient in-place merging”. W: *Theor. Comput. Sci.* 237.1-2 (2000), s. 159–181. DOI: 10.1016/S0304-3975(98)00162-5. URL: [https://doi.org/10.1016/S0304-3975\(98\)00162-5](https://doi.org/10.1016/S0304-3975(98)00162-5).