

Algorytmy i Struktury Danych, 2. ćwiczenia

2025-10-15 (wersja 1.0.1)

Zadanie 2.1

Zaproponuj iteracyjny algorytm, który w czasie $O(\log n)$ oblicza $\lfloor \sqrt{n} \rfloor$ dla danej nieujemnej liczby całkowitej n . Udowodnij poprawność swojego algorytmu.

Algorytm 1: Sqrt(n)

Input: $n > 0$

Output: $\lfloor \sqrt{n} \rfloor$

$l = 1, r = n$

while $l < r$ **do**

 /* niezmiennik $l^2 \leq n$ i $(r+1)^2 > n$ */

$m = \lfloor \frac{l+r+1}{2} \rfloor$ // $l < m \leq r$

if $m^2 \leq n$ **then**

$l = m$

else

$r = m - 1$

return l

// $l = \lfloor \sqrt{n} \rfloor$

Zadanie 2.2

Algorytm ?

begin

 { $x \leq 100$ - liczba całkowita }

$y := x; z := 1;$

while $(y \leq 100)$ **or** $(z \neq 1)$ **do begin**

if $y \leq 100$ **then**

begin $y := y+11; z := z+1$ **end**

else

begin $y := y-10; z := z-1$ **end**

end

end

Udowodnij, że Algorytm ? ma własność stopu.:

Wskazówka: pokaż stan algorytmu na wykresie gdzie kolejne punkty wyznaczone są przez wartość (z, y) . Przeanalizuj jak zachowuje się algorytm w zakresie $90 \leq y \leq 111$.

```

Algorytm ?
begin
  { x <= 100 - liczba całkowita }
  y := x; z := 1;
  while (y <= 100) or (z != 1) do begin
    if 91 <= y < 100 then
      begin y := y + 1; end { złożenie dwóch kroków }
    else if y == 100 then
      begin y := 91; z := z - 1; end { złożenie trzech kroków }
    else if y <= 100 then
      begin y := y+11; z := z+1 end
    else
      begin y := y-10; z := z-1 end
    end
  end
end

```

Zadanie 2.3

Algorytm 2: Sortowanie Szymka R.

```

X := {x1, x2, ..., xn}
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 znajdą 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:

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

- 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 oczekiwania wartość #POP + #PUSH + #TOP jest rzędu $O(n^2)$.

Zadanie 2.4 - 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 2.5 - przesunięcie cykliczne

Funkcja `CyclicLeftShift( $a, k$ )`

```

 $n := len(a)$ 
Reverse( $a, 1, n - k$ )
Reverse( $a, n - k + 1, n$ )
Reverse( $a, 1, n$ )

```

Zadanie 2.6 - stabilne sortowanie ciągów 0/1

Algorytm 3: `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 $O(1)$).

Problem sortowania stabilnego i w miejscu dla ciągów binarnych, można rozwiązać w czasie $O(n)$ jednak takie rozwiązanie jest dosyć skomplikowane [2] <https://hjemmesider.diku.dk/~jyrki/Paper/KP1992bJ.pdf>.

Zadanie 2.7, 2.8 i 2.9 - merge w miejscu

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

Algorytm 4: 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 (dlugosci >= min(n1, |A|-n1))"""
    assert len(C) >= min(n1, len(A) - n1)
    n = len(A)
    if n1 > n - n1:
        # zamien 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

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

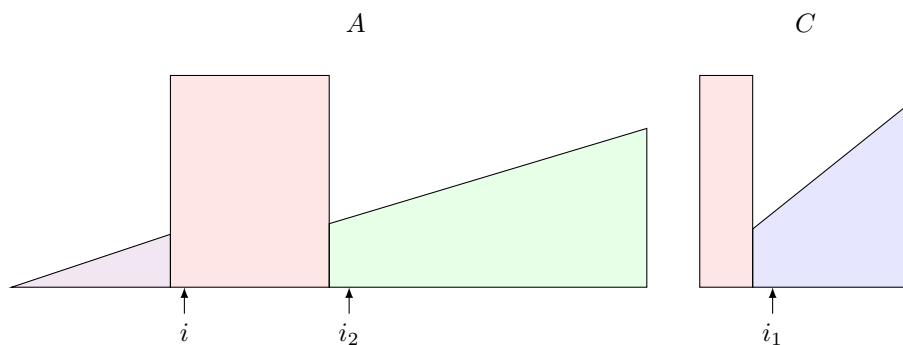
    i, i1, i2 = 0, 0, n1
    while i1 < n1:
        # fragment A[0:i) jest juz uporządkowany
        # pozostale elementy sa w C[i1:n1) i A[i2:n)
        if i2 == n or C[i1] <= A[i2]:
            A[i], C[i1] = C[i1], A[i]
            i1 += 1
        else:
            A[i], A[i2] = A[i2], A[i]
            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 odwracającą dany zakres.

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,



Rysunek 1: Przykładowy stan w trakcie uruchomienia algorytmu MERGE

- 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 5: 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).
- [2] Jyrki Katajainen i Tomi Pasanen. “Stable Minimum Space Partitioning in Linear Time”. W: *BIT* 32.4 (1992), s. 580–585. DOI: 10.1007/BF01994842. URL: <https://doi.org/10.1007/BF01994842>.