Algorytmy i Struktury Danych, 2. ćwiczenia

2022-10-12 (wersja 1.01)

Zadanie 2.1

Algorithm 1: Sortowanie Szymka R.

$X := \{x_1, x_2, \ldots, x_n\}$
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 $\{x_1, \ldots, x_n\}$ jako permutację $\{1, \ldots, n\}$.

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 zredukownay 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 minimalny trafi na pierwszą pozycję stosu

  Niech Y to zmienna losowa oznaczająca liczbę losowań w algorytmie do momentu w którym zostanie wylosowane MIN. 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 prawodpodobieństwo wylosowania MIN 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. $E[Y'] = 1/p$

  $$E[Y'] = p + 2(1-p)p + 3(1-p)^2p + \dots$$
  $$(1-p)E[Y'] = (1-p)p + 2(1-p)^2p + 3(1-p)^3p + \dots$$

  po odjęciu stronami:

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

- 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 2.2 - sortowanie ciągów k-dobrych

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

a) [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.

b) [5 punktów] zaproponuj efektywny czasowo i pamięciowo algorytm, który sprawdza czy ciąg liczb $a[1], \ldots, a[n]$ 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: istnieje co najmniej $(k!)^{n/k}$ różnych permutacji n-elementowych, które są k-dobre. Stąd algorytm sortujący musi wykonać co najmniej $\log((k!)^{n/k})$ porównań, czyli $\Omega(n/k \cdot k \log k) = \Omega(n \log k)$. Konstrukcja tej rodziny permutacji: podziel liczby na bloki $B_1 = 1..k$, $B_2 = k + 1..2k$, itd. permutacja otrzymana z dowolnego przemieszania bloków $p(B_1) + p(B_2) + p(B_3) \ldots$ jest k-dobra.

Algorytm: sortuj kolejno bloki długości $2k$ zaczynające się na pozycjach $1, k + 1, 2k + 1, \ldots$.

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

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

Zadanie 2.3 - sortowanie przy pomocy funkcji ile

a) potrzebujemy co najmniej $n-1$ wywołań ile, żeby wyznaczyć minimalny element z a
b) algorytm sortujący

Algorithm 2: IleSort($a[1,\ldots,n]$)

znajdź $j = argmin(a_1,\ldots,a_n)$
Zamień(a[1], a[j])
$i = 2$
while $i \leq n$ **do**
  $k = Ile(a[1], a[i])$
  if $k \neq i$ **then**
    Zamień($a[i], a[k]$)
  else
    $i = i + 1$

Zadanie 2.4

Wskazówka: jeśli ciąg jest zarówno 7-uporządkowany jak i 11-uporządkowany to dowolne dwa elementy w odległości ≥ 77 są już uporządkowane.

Lemat 1. *Niech $1 < p < q$, takie, że $gcd(p,q) = 1$. Dla $i \geq pq$ istnieje rozwiązanie równania $ap + bq = i$, takie, że $a, b \geq 0$.*

Proof. Łatwo zauważyć, że dla dowolnego $j < q$ (np. dla $j = i \mod q$) istnieje rozwiązanie równania $ap \equiv j(\mod q)$ ($a \equiv j \cdot (p^{-1}(\mod q))$). □

Przykład: Dla $p = 7, q = 11, i = 50$ mamy:

- $j = 50(\mod 11) = 6$,
- $7^{-1}(\mod 11) = 8$,
- $a = 6 \cdot 8(\mod 11) = 4$,
- $b = (50 - 4 \cdot 7)/11 = 2$.
- $ap + bq = 4 \cdot 7 + 2 \cdot 11 = 50$.

Zadanie 2.5 - przesunięcie cykliczne

Function 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

Algorithm 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 = StableMergeSort01($A[1, \ldots, m]$)
  c_2 = StableMergeSort01($A[m+1, \ldots, n]$)
  if $c_1 < m$ **and** $c_2 > 0$ **then**
    Swap($A[c_1 + 1, \ldots, m]$, $A[m+1, \ldots, m+c_2]$) – 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 pamieci $O(1)$).

Zadanie 2.7 - jak sprawdzić czy dwa elementy będą porównane?

Algorithm 4: CzyBędąPorównane(A, x, y)

zlokalizuj pierwszy poziom rekurencji w których w różnych połówkach tablicy rozważane są x i y
niech $x \in X$, $y \in Y$ (X, Y - fragmenty tablicy A)
niech $X' = \{e \in X : e \leq \min(x, y)\}$
niech $Y' = \{e \in Y : e \leq \min(x, y)\}$
return $min(X') = x$ ***and*** $min(Y') = y$

Złożoność czasowa $O(n)$.

Zadanie 2.8, 2.9 i 2.10 - merge w miejscu

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

Algorithm 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 użycia bufora:

```
def merge(A, n1, C):
    """scalanie uporzadkowanych A[0..n1) i A[n1..]
```

```
    przy uzyciu 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 uporzadkowany
        # 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 odwracająca dany zakres.

Scalanie w miejscu

Knuth, Tom III, strona 698.

- podziel tablicę na bloki rozmiaru $\lceil\sqrt{n}\rceil$, — Z_1, Z_2, ..., 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, ..., 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, ..., Z_m używając Z_{m+1} jako bufora pomocniczego,

Algorithm 6: Z-Merge(Z)

foreach $i \in 1, \ldots, m-1$ **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 $Z_1, .., Z_{m+1}$

- 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.