Algorytmy i Struktury Danych, 8. ćwiczenia

2021-11-24

Contents

1 Klasówka 2016 (1), zadanie 2

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

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

2 Klasówka 2014 (1), zadanie 3

Dane są liczby całkowite dodatnie n, k, przy czym $k \le \sqrt{n}$. W tablicy $a[1..n]$ zapisano n liczb całkowitych o co najmniej k różnych wartościach. Należy zaprojektować algorytm, który stabilnie i w miejscu przemieści k parami różnych liczb na początek tablicy a i uporządkuje je rosnąco. Stabilność w tym przypadku oznacza, że kolejność występowania w tablicy liczb o tych samych wartościach zostaje zachowana. Twój algorytm powinien działać w czasie $O(n \log n)$.

Rozwiązanie: Możemy użyć następującego algorytmu:

Algorithm 1: Solution1(A, k)

```
Niech B oznacza blok A w którym będziemy gromadzić posortowane
  rosnąco różne elementy z A
Początkowo B jest pusty blokiem na samym początku A
foreach i ∈ 1, ..., n do
    if binarySearch(A[i], B) then
        // element A[i] jest już znany więc go ignorujemy
    else
        // element A[i] jest nowy i chcemy go dodać do B
        niech X oznacza blok zaczynający się za B i kończący na
          A[i − 1]
        Exchange(B, X)
        dodaj A[i] do B
        if |B| ≥ k then break
przenieś blok B na początek A
```

Analiza: Koszt $O(n \log n)$ ze względu na wykonywane $O(n)$ razy wyszukiwanie binarne. Pozostałe operacje zajmują $O(n)$ czasu:

- koszt dodawania nowych elementów to $O(k^2)$ czyli $O(\sqrt{n}^2) = O(n)$,
- koszt wszystkich operacji Exchange to $O(n)$ ponieważ $\sum_{j=1}^{p} |X_j| \le n$ (zauważmy, że wszystkie zbiory X_j są rozłączne), oraz $\sum_{j=1}^{p} |B_j| \le k^2 \le n$.

3 Klasówka 2013 (1), zadanie 3

Danych jest k uporządkowanych list o długościach będących parami różnymi potęgami dwójki. Zaproponuj wydajny algorytm scalenia tych list w jedną listę uporządkowaną. Uzasadnij poprawność swojego algorytmu i dokonaj analizy jego złożoności obliczeniowej ze względu na liczbę porównań wykonywanych podczas scalania.

Rozwiązanie: Uporządkuj listy rosnąco według długości i scalaj od najkrótszej do najdłuższej. Złożoność czasowa: $O(\sum |L_i|)$. Złożoność pamięciowa: $O(k)$ (na potrzeby uporządkowania list, nie potrzebujemy dodatkowej pamięci na scalanie bo operujemy na listach, które można scalać w pamięci $O(1)$).

4 Klasówka 2013 (1), zadanie 1

Zaprojektuj optymalny algorytm pod względem pesymistycznej liczby porównań, który znajduje dwa środkowe elementy w zbiorze czterech elementów. Dowiedź poprawności swojego rozwiązania.

Rozwiązanie: Algorytm używający 4 porównań:

- porównaj (a, b) i (c, d)
- porównaj $\min(a, b)$ i $\min(c, d)$
- porównaj $\max(a, b)$ i $\max(c, d)$
- w ten sposób wyznaczamy $\min(a, b, c, d)$ i $\max(a, b, c, d)$, pozostałe dwa elementy to poszukiwane środkowe wartości.

5 Klasówka 2012 (1), zadanie 3

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.

Algorithm 2: Sort(A)

```
if |A| ≥ 2 then
    (Z_l, O_l)=Sort(A[1..n/2])
    (Z_r, O_r)=Sort(A[n/2 + 1..n])
    Exchange(O_l, Z_r)
    return (Z_l + Z_r, O_l + O_r)
else
    return (A, ∅) (if A=[0]) or (∅, A) otherwise
```

Algorithm 3: Unpack(A)

if $|A| > 2$ **then**
  $l = \lfloor |A|/4 \rfloor$; $r = \lceil |A|/4 \rceil$;
  // zamień ciąg $0^{|A|/2}1^{|A|/2}$ na $0^l1^l0^r1^r$
  Exchange($A[(l+1)..2l], A[(2l+1)..(2l+r)]$)
  Unpack($A[1..2l]$)
  Unpack($A[(2l+1)..n]$)

6 Klasówka 2011 (1), zadanie 1

Danych jest n słów o takiej samej długości k, zbudowanych ze znaków n-elementowego, uporządkowanego alfabetu. Rozmiarem zadania w tym przypadku jest $R = nk$.

- Zaproponuj algorytm, który dla danego i, $1 \leq i \leq k$, obliczy w czasie $O(R)$ liczbę wszystkich par słów, które różnią się tylko na i-tej pozycji.
- Zaproponuj algorytm, który obliczy w czasie $O(R)$ liczbę wszystkich par słów, które różnią się tylko na dokładnie jednej pozycji.

Rozwiązanie: Zakładamy że wszystkie słowa na wejściu są różne (możemy to łatwo sprawdzić).

Dla dowolnego i, j przez $pref(i, j)$ oznaczamy kod prefiksu słowa w_i długości j, chcemy żeby kody były liczbami z zakresu 1..n takimi, że, $w_i[1..j] = w_q[1..j]$ wtw $pref(i, j) = pref(q, j)$ (czyli mogą służyć do porównywania prefiksów ustalonej długości)

Analogicznie definiujemy dla sufiksów: $suf(i, j)$.

Rozwiązujemy w czasie $O(n)$ każdy problem z osobna dla $j \in 1..k$ (w tym kroku będziemy liczyć pary słów które różnią się dokładnie na j-tej pozycji)

P = $\emptyset$
for *i:=1..n* **do**
  for *j:=1..k* **do**
    P+= $(pref(i, j-1), suf(i, k-j), i)$ (czyli zapisujemy kod słowa bez j-tego znaku)
posortuj leksykograficznie trójki z P
ile:=0
foreach *grupy G trójek o tych samych wartościach pierwszych dwóch elementów* **do**
  // *dowolna para słów z G różni się jedynie na j-tej pozycji*
  $ile+ = |G| * (|G| - 1)/2$

Warto zauważyć, że jeśli jakieś dwa słowa różnią się na dokładnie jednej pozycji to istnieje tylko jedna wartość j w której zostaną zliczone

Ponieważ każda faza zajmuje czas $O(n)$ i faz jest k więc cały algorytm zajmuje $O(nk)$.

Pozostaje jeszcze powiedzieć jak obliczyć pref/suf - robimy to podobnie jak w izomorfizmie drzew, trzeba po prostu kompresować kody:

```
for i:=1 to n do
    pref(i, 1) = w_i[1]
for j:=2 to k do
    P = ∅
    for i:=1 to n do
        P += (pref(i, j − 1), w_i[j], i)
    posortuj leksykograficznie trójki z P
    zgrupuj trójki o tych samych wartościach pierwszych dwóch
     elementów w G_1, G_2, ..G_p
    for t:=1 to p do
        foreach (p, q, i) ∈ G do
            pref(i, j) = t
```

7 Klasówka 2008 (1), zadanie 2

Zaproponuj wzbogacenie kopca zupełnego w taki sposób, żeby efektywnie w czasie zamortyzowanym wykonywane były operacje: Min, DeleteMin, Insert, CountMin. Ostatnia operacja polega na podaniu aktualnej liczby elementów w kopcu o wartości równej Min. Przeprowadź analizę kosztu zamortyzowanego wykonania poszczególnych operacji.

Rozwiązanie: Wzbogacamy węzły kopca o atrybut `countEq` oznaczającą liczbę węzłów **w poddrzewie** zawierających identyczną wartość co ten zapisany w kluczu. Uwaga! atrybut ten nie ma znaczenia globalnego (bo trudno byłoby aktualizować jego wartości) tylko lokalne i dotyczy tylko poddrzewa danego węzła.

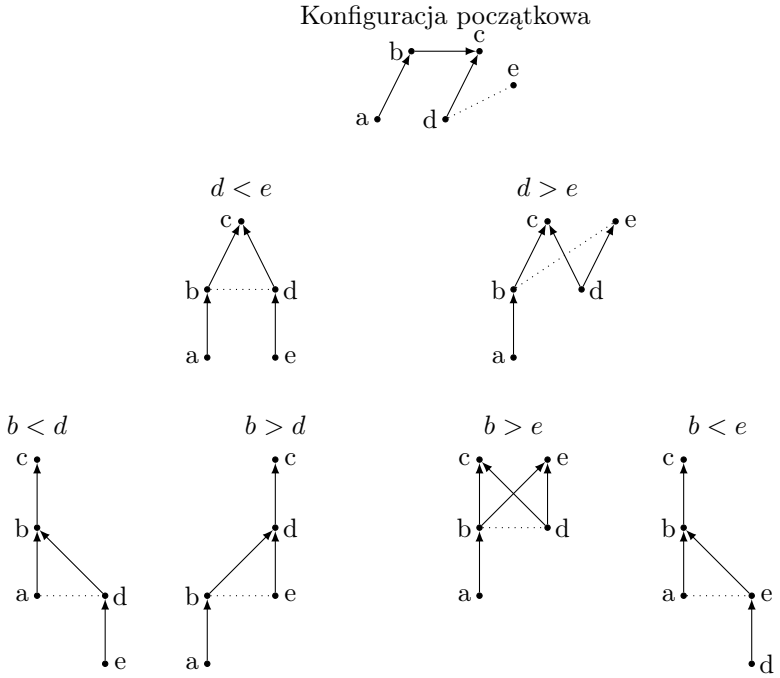
Dzięki takiemu atrybutowi CountMin jest operacją trywialną. Możemy też aktualizować wartość tego atrybutu przy wszystkich operacjach kopcowych.

8 Klasówka 2010 (1), zadanie 2

Wykaż, że każdy algorytm znajdujący medianę w zbiorze 5-elementowym wykona w pesymistycznym przypadku co najmniej 5 porównań. Zaproponuj algorytm dokonujący tego za pomocą co najwyżej 6 porównań.

Rozwiązanie: Dolna granica: dzielimy wszystkie permutacje $\{1, \ldots, 5\}$ na klasy abstrakcji: (pozycja mediany, zbiór pozycji elementów mniejszych od mediany). Na przykład permutacja $(5, 1, 4, 3, 2)$ należy do klasy abstrakcji $(4, \{2, 5\})$. Takich klas abstrakcji jest $5 \cdot \binom{4}{2} = 30$. Dowolne drzewo porównań które rozróżnia wszystkie klasy abstrakcji musi mieć wysokość $h \geq \log_2 30 > 4$. Zauważmy, że jeśli algorytm utożsamia jakieś dwie klasy abstrakcji to możemy skonstruować dane dla których udzieli nieprawidłowej odpowiedzi.

Algorytm wykonujący 6 porównań. Porównaj a i b, porównaj c i d, porównaj $\max(a, b)$ i $\max(c, d)$. Bez utraty ogólności $a > b > c$ i $d > c$. Następnie:



9 Izomorfizm drzew

W tym zadaniu rozważamy drzewa ukorzenione o n wierzchołkach $1, 2, \ldots n$. Korzeniem drzewa jest zawsze wierzchołek 1.

W takim drzewie jest jednoznacznie określona funkcja $f : \{1, 2, \ldots n\} \to \{0, 1, 2, \ldots n\}$, która każdemu wierzchołkowi i, różnemu od korzenia, przyporządkowuje wierzchołek $f(i) =$ ojciec i w drzewie; $f(1)$ jest zawsze równe 0.

Dwa n-wierzchołkowe drzewa T_1 i T_2, zadane odpowiednio przez funkcje f_1 i f_2, są izomorficzne wtedy i tylko wtedy, gdy istnieje taka różnowartościowa funkcja $g : \{1, 2, \ldots, n\} \to \{1, 2, \ldots, n\}$, że dla każdego $i = 1, 2, \ldots, n$, $g(f_1(i)) = f_2(g(i))$.

Zaproponuj algorytm, który w czasie $O(n)$ sprawdzi, czy dwa ukorzenione drzewa T_1 i T_2 są izomorficzne.

Rozwiązanie:

Algorithm 4: TreeIsomorphism($T_1, T_2, depth$)

```
if T1.height > depth then
    return T1.height == T2.height;
if not TreeIsomorphism(T1, T2, depth + 1) then
    return false;
foreach v ∈ T1.nodes[depth + 1] ∪ T2.nodes[depth + 1] do
    (w porządku rosnących etykiet)
    dodaj value(v) do listy wierzchołka parent(v)
posortuj leksykograficznie listy value(v) dla v ∈ T1.nodes[depth]
posortuj leksykograficznie listy value(v) dla v ∈ T2.nodes[depth]
porównaj czy listy są identyczne, jeśli nie to return false
zamień etykiety value(v) na liczby z zakresu 1, ..., n
return true;
```