Algorytmy i Struktury Danych, 4. ćwiczenia, rozwiązania zadań z serii 3

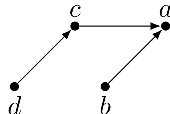
2022-10-26

Zadanie 3.1

a) Zaproponuj algorytm sortujący ciągi 5-elementowe, optymalny ze względu na porównania (wykonujący możliwie najmniej porównań w pesymistycznym przypadku). Udowodnij poprawność swojego rozwiązania.

Rozwiązanie: Niech $A = (a, b, c, d, e)$.

- $compare(a, b)$, (bez straty ogólności, niech $a < b$)
- $compare(c, d)$, (niech $c < d$)
- $compare(a, c)$, (niech $a < c$)



- teraz wsortowujemy, e pomiędzy a, c, d,
  if $(c > e)$ **then** $compare(e, a)$ **else** $compare(e, d)$
- możemy otrzymać jeden z następujących częściowych porządków:

  każdy z nich można posortować używając 2 porównań.

b) Zaproponuj optymalny ze względu na porównania algorytm sortujący 6 różnych liczb a, b, c, d, e, f, o których wiadomo, że $a < b$ oraz $c < d$.

Rozwiązanie: Wszystkich ciągów spełniających warunki zadania jest $6!/4 = 180$ (z każdej permutacji z $a < b$ i $c < d$ możemy wygenerować 3 inne zamieniając wartości a/b lub c/d lub a/b i c/d).

Algorytm sortujący: zastosuj schemat dla optymalnego sortowania 5-elementów (a, b, c, d, e) z pominięciem dwóch pierwszych porównań (znamy już ich wynik z założenia zadania) — 5 porównań. Następnie wsortuj f do uporządkowanego

ciągu 5-elementowego — 3 porównania. Razem wykonaliśmy 8 porównań, jest to optymalna liczba ponieważ $\lceil \log_2 180 \rceil = 8$.

c) Udowodnij, że do scalania dwóch ciągów uporządkowanych o długościach 2 i 5 potrzeba i wystarcza 5 porównań.

Rozwiązanie: Wszystkich permutacji spełniających warunki zadania jest $\binom{7}{2} = 21$ (spośród 7 liczb wybieramy 2 należące do krótszego ciągu).

Algorytm: Niech pierwszy ciąg to a_1, a_2 a drugi to b_1, b_2, b_3, b_4, b_5. Wsortowujemy a_1 do ciągu b rozpoczynając od porównania z b_2. Następnie wsortowujemy a_2 do pozostałej części ciągu b. W sumie potrzebujemy 5 porównań (jeśli $a_1 < b_2$ to 2+3 porównania, wpp, 3+2 porównania). Jest to również dolna granica, ponieważ $\lceil \log_2 21 \rceil = 5$.

Zadanie 3.2

W tym zadaniu badamy algorytmy (turnieje), które polegają na wykonaniu ciągu porównań na elementach danych. Każde takie porównanie nazywamy pojedynkiem, a o elemencie większym w pojedynku mówimy, że jest jego zwycięzcą.

a) Udowodnij, że każdy algorytm znajdujący przez porównania największy element w zbiorze n-elementowym, wykonuje w pesymistycznym przypadku co najmniej n-1 pojedynków

Rozwiązanie: Dla dowolnego algorytmu A, za każdym razem, gdy porównywane są dwa elementy, to łączymy je krawędzią. Jeśli A użył mniej niż $n-1$ porównań, to istnieją co najmniej dwie spójne składowe, więc i dwa elementy które nie są ze sobą porównywalne.

b) Udowodnij, że w każdym algorytmie wyznaczania elementu największego w zbiorze n-elementowym, element największy musi w pesymistycznym przypadku rozegrać co najmniej $\log n$ pojedynków.

c) Udowodnij, że optymalny algorytm wyznaczania 2-go elementu co do wielkości wykonuje w pesymistycznym przypadku co najmniej $n + \lceil \log n \rceil - 2$ pojedynków, $n > 1$.

Rozwiązanie:

- budujemy drzewo turniejowe (porównujemy sąsiednie elementy, dalej przechodzi wygrany) — ten krok zabiera $n-1$ porównań,
- niech S zbiór elementów które przegrały z liderem, $|S| = \lceil \log n \rceil$
- wybierz lidera wśród elementów S — ten krok zabiera $|S| - 1 = \lceil \log n \rceil - 1$ porównań.
- razem $n + \lceil \log n \rceil - 2$

Dowód, że algorytm jest optymalny. Knuth, tom III, 5.3.3. strona 221.

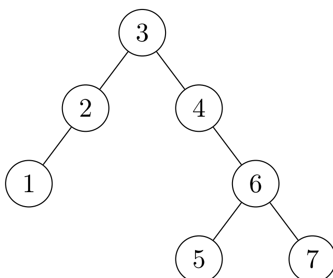
d) Zaproponuj metodę wyznaczania k-tego elementu co do wielkości ($1 < k \leq n/2$), w której w pesymistycznym przypadku wykonuje się co najwyżej $n - k + (k-1)\lceil \log(n-k+2) \rceil$ pojedynków. Wskazówka: zauważ, że element największy w podzbiorze (n-k+2)-elementowym nie może być k-tym co do wielkości.

Rozwiązanie:

- zbuduj drzewo turniejowe dla $A[1..n-(k-2)]$ elementów,
- dla każdego z elementu x z $A[(n-(k-2)+1)..n]$ ($k-2$ elementów), kolejno zastąp $\max(A[1..(n-(k-2))])$ przez x (to wymaga $\lceil \log(n-k+2) \rceil$ pojedynków)
- usuń $\max(A[1..(n-(k-2))])$ (znowu $\lceil \log(n-k+2) \rceil$ pojedynków)
- k-ty element to aktualnie największy element w drzewie turniejowym (w poprzednim kroku pozbyliśmy się $(k-1)$ największych elementów z A)

Zadanie 3.3 - Quicksort

Rozważmy rekurencyjny algorytm Quick Sort sortujący różnowartościowe ciągi liczbowe w taki sposób, że elementem dzielącym jest zawsze pierwszy element sortowanego ciągu, a względny porządek elementów w sortowanych rekurencyjnie podciągach jest taki sam, jak w całym ciągu. Dla ustalonego ciągu, sortowanie można przedstawić w postaci drzewa binarnego, w którego węzłach są zapisywane elementy dzielące, a dla każdego węzła lewe poddrzewo zawiera elementy mniejsze od elementu dzielącego, natomiast prawe poddrzewo - elementy większe. Takie drzewo nazywamy drzewem QS. Oto drzewo QS dla ciągu 3,4,2,1,6,7,5 (wysokość tego drzewa wynosi 3):



a) Ile jest permutacji liczb 1, 2, 3, 4, 5, dla których drzewo QS ma wysokość 3?

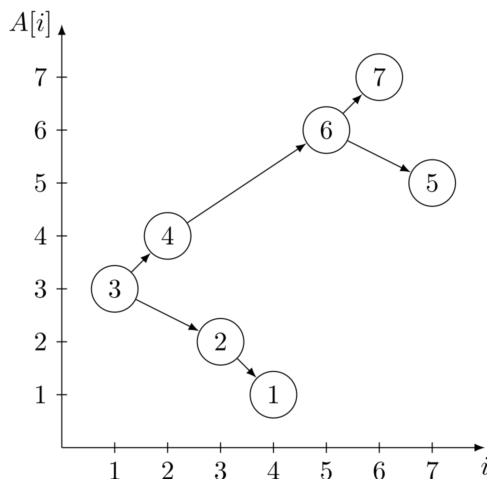
b) Zaprojektuj efektywny algorytm, który dla danych: dodatniej liczby całkowitej n, różnowartościowego ciągu liczb całkowitych o długości n oraz liczby naturalnej k, $1 \le k \le n$, wyznaczy liczbę porównań w algorytmie Quick Sort, w których bierze udział k-ty element ciągu. W przykładzie powyżej, 5-tym elementem ciągu jest 6 i bierze on udział w 4 porównaniach.
c) Zaprojektuj algorytm, który dla danej permutacji liczb $1, 2, \ldots, n$ obliczy w czasie liniowym liczbę porównań wykonywanych przez algorytm Quick Sort przy sortowaniu tej permutacji.

Rozwiązanie: Załóżmy, że dane wejściowe zostały zapisane w tablicy $A[1, \ldots, n]$. Ponieważ zakładamy, że wejściowy ciąg jest permutacją $1, \ldots, n$ stąd możemy zdefiniować A^{-1}:

$$A^{-1}[j] = \{i : A[i] = j\}$$

Czyli A^{-1} dla każdej wartości zawiera odpowiadający jej indeks w tablicy A.

Zanim przejdziemy do rozwiązania, rozrysujmy drzewa QS (dla przykładowej permutacji) w trochę inny sposób:



Możemy zasymulować algorytm i wygenerować drzewo QS używając następującego algorytmu:

Algorithm 1: Calc-QS-Tree($A[1,\ldots,n]$)

Function CALC(S)**:**
- $pivot = L[1]$
- $L = \{x : x \in S[2,\ldots,|S|] \text{ oraz } x < pivot\}$
- $R = \{x : x \in S[2,\ldots,|S|] \text{ oraz } x > pivot\}$
- **if** $L \neq \emptyset$ **then**
  - CALC(L)
  - $parent[L[1]] = pivot$
- **if** $R \neq \emptyset$ **then**
  - CALC(R)
  - $parent[R[1]] = pivot$

$parent(v) = \text{NULL}$ (dla $v \in 1,\ldots,n$)
CALC($A[1,\ldots,n]$)

Niestety takie rozwiązanie ma dokładnie taką samą złożoność jak Quick-Sort czyli pesymistycznie kwadratową (np. dla posortowanego ciągu).

Rozwiązanie brutalne:

Skorzystajmy z następującego lematu (więcej szczegółów można przeczytać na stronie `https://www.mimuw.edu.pl/~rytter/TEACHING/JAO/radoszwski_delta.pdf`):

Lemat 1. *Dla zadanej tablicy liczb całkowitych $X[1,\ldots,n]$, po liniowym preprocessingu, możemy odpowiadać w czasie stałym na dowolne zapytania postaci:*

$$\text{RMQ}(X, l, r) = \min(X[l,\ldots,r])$$

Zauważmy, że wartości rozważane w CALC(S) zawsze są spójnymi przedziałami zakresu $1,\ldots,n$, stąd możemy przepisać nasz algorytm w następujący sposób:

Algorithm 2: Calc-QS-Tree-2($A[1,\ldots,n]$)

```
Input: A: permutacja liczb 1,...,n
Function CALC(l, r):
    pivot = RMQ(A^{-1}, l, r)
    if l < pivot then
        lp = CALC(l, pivot − 1)
        parent[lp] = pivot
    if pivot < r then
        rp = CALC(pivot + 1, r)
        parent[rp] = pivot
    return pivot
parent(v) = NULL (dla v ∈ 1,...,n)
przygotuj tablicę A^{-1} do zapytań RMQ CALC(1, n)
```

Rozwiązanie eleganckie:
Dla ustalonej tablicy $A[1,\ldots,n]$ zdefinujmy następujące wartości:

$$LPred[i] = \max\{x : x \in A[1,\ldots,(i-1)] \text{ oraz } x < A[i]\}$$

$$LSucc[i] = \min\{x : x \in A[1,\ldots,(i-1)] \text{ oraz } x > A[i]\}$$

Dla pewnych indeksów niektóre wartości mogą nie być zdefiniowane – gdy np. obliczamy min lub max po pustych zbiorach.
Przykład:

i	1	2	3	4	5	6	7
$A[i]$	3	4	2	1	6	7	5
$LPred[i]$	-	3	-	-	4	6	4
$LSucc[i]$	-	-	3	2	-	-	6

Wartości $LPred$ (i analogicznie $LSucc$) możemy obliczyć w czasie $O(n)$ korzystając z bardzo prostych struktur danych:

Algorithm 3: obliczanie $LPred$

```
S = pusty stos
foreach j ∈ {1,...,n} do
    // niezmiennik: stos zawiera indeksy i_1,...,i_k (Top(S)=i_k)
    // i_1 < i_2 < ... < i_k oraz A[i_1] < A[i_2] < ... < A[i_k]
    i = A^{-1}[j]
    while |S| > 0 and Top(S) > i do
        Pop(S)
    if |S| > 0 then
        LPred[i] = A[Top(S)]
    else
        LPred[i] = -
    Push(i)
```

Lemat 2. *Niech $A[i] = j$, $LPred[i] = p$, $LSucc = s$, wartość $parent[j]$ jest równa:*

- *jeśli* $p \neq$ - *i* $s \neq$ - *to odpowiedzią jest wartość, która jest bardziej na prawo, czyli: jeśli* $A^{-1}[p] > A^{-1}[s]$ *to* $parent[j] = p$ *w przeciwnym przypadku* $parent[j] = s$,
- *jeśli* $p \neq$ - *i* $s =$ - *to* $parent[j] = p$
- *jeśli* $p =$ - *i* $s \neq$ - *to* $parent[j] = s$
- *jeśli* $p =$ - *i* $s =$ - *to* $parent[j] = NULL$.

Jeszcze ładniejsz rozwiązanie:

Drzewo QS ma kształ dokładnie taki jak Cartesian Tree (`https://en.wikipedia.org/wiki/Cartesian_tree`) tablicy A^{-1}. A takie drzewo można zbudować w czasie $O(n)$.

Zadanie 3.4

Powiemy, że dwa napisy są podobne wtedy i tylko wtedy, gdy zawierają jednakowe liczby wystąpień tych samych znaków. Danych jest n napisów nad alfabetem m-znakowym $\{1, 2, \ldots, m\}$. Zaproponuj algorytm, który stwierdza, ile jest wśród nich różnych klas napisów podobnych. Twój algorytm powinien działać w czasie $O(R + m)$, gdzie R jest sumą długości wszystkich napisów.

Rozwiązanie: Podstawowa idea:

- dla każdego słowa w_i oblicz jego kod $code(w_i) = sorted(w_i)$, gdzie $sorted(w)$ oznacza słowo w z uporządkowanymi niemalejącymi znakami (np. $sorted(adbacab) = aaabbcd$)
- posortuj słowa $code(w_1), \ldots, code(w_n)$ używając algorytmu z ćwiczeń (sortowanie leksykograficzne słów różnej długości)
- usuń duplikaty z posortowanej listy.

Kroki drugi i trzeci w uczywisty sposób zajmą czas $O(R + m)$ Niestety jeśli pierwszy krok tego algorytmu zaimplementujemy naiwnie, to może się okazać, że obliczenie $code(w_i)$ zajmie nam czas $O(|w_i| + m)$, co w sumie może dać $O(R + nm)$.

Na szczęście możemy wygenerować kody słów w efektywniejszy sposób. Każdy znak z $w_1, \ldots, w_n$ zastępujemy przez trójkę (c, i, j) oznaczającą że $w_i[j] = c$. Sortujemy wszystkie trójki w jednym kroku. Teraz dzięki tej posortowanej liście mamy uporządkowane wszystkie litery z całego zbioru słów i możemy je kolejno dopisywać do kodów słów:

Input: lista słów $w_1, \ldots, w_n$
T:=pusta lista
foreach $w_i \in w_1, \ldots, w_n$ **do**
  foreach $j \in 1, \ldots, |w_i|$ **do**
    dodaj $(w_i[j], i, j)$ do T

posortuj leksykograficznie trójki z T
foreach $i \in 1, \ldots, n$ **do**
  $code[w_i] = \epsilon$ (pusty napis)

foreach $(c, i, j) \in T$ **do**
  $code[w_i] += c$

Dzięki "zbiorczemu" sortowaniu listy T udało się obliczyć kody wszystkich słów w w czasie $O(R + m)$.
Przykład:

```
w_1 = aba
w_2 = ba
w_3 = caa
w_4 = ab

T = [
  (a, 1, 1), (b, 1, 2), (a, 1, 3),
  (b, 2, 1), (a, 2, 2),
  (c, 3, 1), (a, 3, 2), (a, 3, 3),
  (a, 4, 1), (b, 4, 2)
]

posortowane T = [
  (a, 1, 1), (a, 1, 3), (a, 2, 2), (a, 3, 2), (a, 3, 3), (a, 4, 1),
  (b, 1, 2), (b, 2, 1), (b, 4, 2),
  (c, 3, 1)
]

code(w_1) = aab
code(w_2) = ab
code(w_3) = aac
code(w_4) = ab
```

Zadanie 3.5

Dana jest tablica liczb całkowitych a[1..n], o której wiadomo, że dla każdego $i = 1, 2, \ldots, n$,

$$|\{j : |a[j] - a[i]| \leq n\}| > n/2020.$$

Zaproponuj liniowy algorytm sortowania tablicy a.

Rozwiązanie:

Algorithm 4: Sort(a)

$wynik = \emptyset$
while $a \neq \emptyset$ **do**
  $m = min(a)$
  wybierz elementy z zakresu $[m, m+2n] \in a$ i zapisz je w tablicy b
  usuń elementy należące do b z a
  posortuj b używając algorytmu CountingSort
  dodaj posortowaną tablicę b na koniec tablicy $wynik$

Możemy zauważyć, że dla tablic spełniających warunki zadania, pętla **while** wykona się $O(1)$ razy.

Niech $adj(x) = \{j : |x - a[j]| \leq n\}$, z założenia zadania wiemy, że dla $x \in a$, mamy $|adj(x)| > n/2020$.

Niech $m_1, \ldots, m_k$ to ciąg minimów wybranych przez algorytm. Z warunku wyboru tablicy b mamy $|m_i - m_j| > 2n$ dla $i \neq j$. Stąd jeśli popatrzymy na zbioru $adj(m_i)$ i $adj(m_j)$ są one parami rozłączne. Czyli, jeśli algorytm wykona k iteracji, to $|\bigcup_{1 \leq i \leq k} adj(m_i)| > \frac{kn}{2020}$.

Zadanie 3.6

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 5: TreeIsomorphism($T_1, T_2, depth$)

if $T_1.height > depth$ **then**
  return $T_1.height == T_2.height$;
if not *TreeIsomorphism*($T_1, T_2, depth + 1$) **then**
  return *false*;
foreach $v \in T_1.nodes[depth + 1] \cup T_2.nodes[depth + 1]$ **do**
  (w porządku rosnących etykiet)
  dodaj $value(v)$ do listy wierzchołka $parent(v)$
posortuj leksykograficznie listy $value(v)$ dla $v \in T_1.nodes[depth]$
posortuj leksykograficznie listy $value(v)$ dla $v \in T_2.nodes[depth]$
porównaj czy listy są identyczne, jeśli nie to **return** false
zamień etykiety $value(v)$ na liczby z zakresu $1, \ldots, n$
return *true*;