

# AISD lista 7

Dominik Szczepaniak

August 30, 2024

## 1 Zadanie 1

Ponieważ w naszej operacji  $\text{Union}(A, B)$  przypisujemy  $A$  jako ojca  $B$ , to mamy  $O(1)$ . Jeśli wszystkie  $\text{Union}$  są przed  $\text{find}$  to najpierw wykonamy wszystkie  $\text{Union}$  w złożoności  $O(1)$ .

Teraz mamy same operacje  $\text{Find}(x)$ . Założmy, że elementów jest  $n$ . Założmy, że wszystkie wierzchołki są na początku odznaczone. Zaznaczamy wierzchołek tylko gdy przejdziemy przez niego jakimś  $\text{findem}$  (również gdy szliśmy z kogoś niżej do góry). W takim razie widać, że jeżeli nie ma żadnych  $\text{unionów}$  po wykonaniu  $\text{finda}$ , to jeśli wierzchołek będzie zaznaczony raz to będzie od razu podpięty pod ojca zbioru. Czyli każdy wierzchołek możemy zaznaczyć co najwyżej raz każdym  $\text{findem}$ , czyli mamy złożoność co najwyżej  $O(n)$ .

$$\text{Union} - O(n) + \text{Find } O(n) = O(2n) = O(n)$$

## 2 Zadanie 2

Możemy użyć drzewa Splay. Jeśli mamy  $\text{insert}$  to normalnie sobie  $\text{insertujemy}$ . Jeśli mamy  $\text{min}(i)$  to  $\text{splayujemy}$  pierwszą wartość większą niż  $i$ , i usuwamy lewe drzewo. Jak mamy  $\text{deletemin}$  to po prostu idziemy na  $\text{maxa}$  na lewo i usuwamy lewą wartość. Jeśli tutaj będziemy mieć długą ścieżkę w prawo to nam to nic nie wadzi, bo zawsze usuwamy minimalny element, więc usuniemy po prostu korzeń.

Problem - możemy usuwać i inserować ostatni element na zmianę i mamy ciągle złożoność  $O(n)$ , więc łącznie  $O(n^2)$

=====

Generalnie to jeżeli każda liczba może być dodana maksymalnie raz, to wszystkich liczb jest maksymalnie  $n$ .

W takim razie operacji deleteMin oraz Min nie może być więcej niż  $n$ , bo każda liczba jest dodana maksymalnie raz, a te operacje muszą usunąć przynajmniej jedną liczbę.

W takim razie możemy zrobić zwykłe drzewo AVL / CzerwonoCzarne które normalnie sobie insertuje liczby a później usuwa albo najmniejszą liczbę albo wszystkie liczby po kolei. Wtedy mamy złożoność  $O(n \log n)$ .

=====

Dla każdego zbioru chcemy sobie zachować minę który obowiązuje

Czyli dla  $\sigma_1$  min D

sigma 1 przypisujemy tego min

ale dla  $\sigma_1$  D min

nie przypisujemy min nikomu

jeśli poprzedni min był mniejszy niż obecny to jest spoko, a jeśli był większy to będziemy go i tak już dobrze pamiętać

Niech  $k$  będzie ilością deleteMin

Wtedy mamy ciąg  $\sigma = \sigma_1 E \sigma_2 E \sigma_3 \dots \sigma_k E \sigma_{k+1}$ , gdzie każda  $\sigma_j$   $1 \leq j \leq k+1$  to ciąg insertów.

Czyli ostatni zbiór trzymamy na "śmieci" - wartości już znalezione lub te których i tak nigdy nie będzie.

Za każdym razem gdy wyjmemy liczbę ze zbioru to możemy ją wrzucić do zbioru "śmieci"

Inicjalizujemy ciąg zbiorów dla union finda tak, żeby set nazywający się  $j$  zawierał liczby z  $\sigma_j$

Do tego trzymamy dwie tablice PRED i SUCC które tworzymy do zrobienia podwójnie skierowanej linked listy posortowanej dla tych wartości  $j$  dla których zbiory nazwane  $j$  istnieją.

Na początku  $PRED[j] = j-1$  dla  $1 \leq j \leq k+1$  i  $SUCC[j] = j+1$  dla  $0 \leq j \leq k$ .

for  $i = 1$  until  $n$  do:

$j = \text{find}(i)$

if  $j \neq m+1$ :

$\text{extracted}[j] = i$

let  $l$  be smallest value greater than  $j$  for which set  $K_l$  exists

$K_l = \text{union}(K_j, K_l)$  i usuwamy  $K_j$

Dowód:

Założmy, że `extracted` jest nieprawidłowe. Niech  $x = \text{extracted}[j]$  będzie najmniejszą wartością dla której `extracted[j]` jest nieprawidłowe. Niech prawidłowa wartość będzie w tablicy "correct" i niech ta wartość nazywa się  $y$ . Mamy dwie możliwości -  $x < y$  oraz  $x > y$ .

a) Niech  $x > y$ . Wtedy  $y$  nie może wystąpić w `extracted`, bo inaczej byłoby najmniejszą wartością dla której wynik jest nieprawidłowy. Ponieważ już przepracowaliśmy  $y$  przed przepracowaniem  $x$  musi być w zbiorze  $m+1$ . Ale jeśli `correct[j] = y` to  $y$  musi być gdzieś w  $K_i$  gdzie  $i < j$ . Ponieważ `extracted[j]` nie miało wartości gdy było procesowane  $y$ , to nie moglibyśmy wrzucić wartości  $y$  do zbioru  $m+1$ , bo łączymy się tylko z setami większymi od nas i nie zrobiliśmy uniona z  $K_j$ , więc  $x > y$  nie może zachodzić.

b) Założmy, że  $x < y$ . Mówimy, że element  $x$  musi wystąpić w tablicy `correct`. Oczywiście  $x$  musi wystąpić przed  $j$ -tym wyjęciem w oryginalnym wejściu ponieważ nasz algorithm nigdy nie przesuwa zbiorów elementów wstecz. Jeśli nie wyodrębniliśmy  $x$  do  $j$ -tej selekcji to optymalne rozwiązanie powinno wybrać  $x$  zamiast  $y$  dla  $j$ -tej selekcji, ponieważ  $x$  jest mniejszy. Dlatego optymalne rozwiązanie musiało wyodrębnić  $x$  dla jakiegoś  $i < j$ . Ale to oznacza, że `extracted[i]` trzyma jakieś  $z > x$ . Z podobnych powodów jak powyżej nie mogliśmy przesunąć  $x$  poza zbiór  $K_i$ , ponieważ `extracted[i]` byłoby puste w momencie w którym  $x$  został wybrany. Więc ponieważ łączymy się tylko ze zbiorami powyżej nas i  $K_i$  nie został jeszcze połączony nie możemy umieścić  $x$  w `extracted[j]` przed `extracted[i]`, dlatego nie możemy mieć  $x < y$ .

W takim razie `extracted[j] = correct[j]`, więc algos jest poprawny.

Złożoność:

Robimy  $n$  setów make-set

Robimy uniony  $n-m$  razy (gdzie  $m$  to liczba deleteMin)

Dla każdego seta trzymamy 3 wartości - `number`, `prev`, `next` (trzymamy je każdy, ale potrzebuje je trzymać tylko przedstawiciel)

`number` mówi o  $j$  w  $K_j$  i może być łatwo ustawiony na samym początku tworzenia setów

`prev` wskazuje na poprzedniego reprezentanta (reprezentanta  $K_{j-1}$ ), `next` następnego reprezentanta

Gdy robimy union na dwóch setach  $j$  i  $l$ , gdzie  $l$  to zbiór który dalej istnieje po  $j$ , to ustawiamy `number` na maximum `number` z dwóch reprezentantów, czyli  $l$ . `Prev` ustawiamy na `prev` z setu  $j$ , a `next` z `prev` z setu  $j$  ustawiamy na nowy `number`, `next` na `next` z setu  $l$  i `prev` z `next` z setu  $l$  na nowy reprezentant

Mamy  $n$  razy find-set -  $O(n \log^* n)$

Później "let  $l$  be smallest value greater than  $j$  for which set  $K_l$  exists" to wystarczy pójść do next dla  $j$   $O(1)$

Później mamy Union() co najwyżej  $m$  razy, czyli łącznie mamy  $n$  unionów  
Mamy więc  $O(n \log^* n)$  razy.

### 3 Zadanie 3

Na początku każde drzewo ma jeden element. Każdy wierzchołek trzyma wagę którą początkowo jest równa 0.

Dodatkowo każdy korzeń (drzewo) pamięta ile wierzchołków w sobie przechowuje.

Jeśli mamy ścieżkę  $v_1, v_2, v_3, \dots, v_k$ , gdzie  $v_k$  to korzeń a  $v_1$  to  $v$ .

$\text{Depth}(v)$ :

Trzymamy niezmiennik że  $\text{Depth}(v)$  = suma wag od  $v$  do korzenia drzewa w którym jest  $v$ , czyli  $\text{Depth}(v) = \sum_{i=1}^k v_i$

Wykonujemy kompresję ścieżki w następujący sposób:

1.  $\forall v_i$  podczepiamy  $v_i$  pod  $r$

2.  $\forall v_i$  ustalamy  $\text{weight}(v_i)$  na  $\sum_{j=i}^{k-1} \text{weight}(v_j)$

$\text{Link}(v, r)$ :

Niech  $T_r$  - drzewo posiadające  $r$ , gdzie  $r'$  to jego korzeń (z treści)

Niech  $T_v$  - drzewo posiadające  $v$ , i  $v'$  korzeń tego drzewa

Rozważamy przypadki:

1.  $\text{Count}(T_r) \leq \text{Count}(T_v)$ :

Podczepiamy  $r'$  pod  $v'$  i wykonujemy  $\text{Depth}(v)$ .

Zaktualizujemy wagę  $r'$  na:

$$\text{weight}(r') = \text{weight}(r') - \text{weight}(v') + \text{depth}(v) + 1$$

$\text{weight}(r') - \text{weight}(v')$  z podpiecia  $r'$  pod  $v'$ , a  $\text{depth}(v) + 1$  z tego że  $r$  podpinamy pod  $v$

Na koniec aktualizujemy county więc:

$$\text{Count}(T_v) + = \text{count}(T_r)$$

2.  $\text{Count}(T_r) > \text{Count}(T_v)$

Podczepiamy  $v'$  pod  $r'$ .

Aktualizujemy  $r'$  oraz  $v'$ :

$$\text{weight}(r') = \text{weight}(r') + \text{depth}(v) + 1 \text{ z podpiecia } r \text{ pod } v$$

$$\text{weight}(v') = \text{weight}(v') - \text{weight}(r')$$

Na koniec:

$$\text{Count}(T_R) + = \text{Count}(T_v)$$

## 4 Zadanie 4

No generalnie to jeżeli zwiększymy stałą przy operacji find to zwiększymy złożoność. Tutaj nam się dodaje kolejny log, a złożoność i tak była  $\log^* = \log(\log(\log(\dots)))$  czyli dodanie jednego loga generalnie nie będzie nas jakoś bardzo boleć, bo i tak mamy  $\log^*$ . Czyli możemy użyć tego samego rozumowania, ponieważ tutaj dochodzi tylko jeden logarytm, stąd trzeba zastosować inną stałą do operacji find w tym zadaniu.

## 5 Zadanie 5

Dostajemy wierzchołek  $v$  i dla każdego wierzchołka chcemy umieć odpowiedzieć ile krawędzi początkowych musimy usunąć aby ten wierzchołek  $i$  i  $v$  nie było połączone żadną krawędzią.

Idziemy od tyłu i dodajemy krawędzie tworząc sobie drzewa w union findzie. Za każdym razem gdy zrobimy union to chcemy zrobić finda na jednym z dwóch wierzchołków które dodaliśmy (żeby drzewa były postaci ojciec i synowie i nie było głębokości 2). Jeśli nagle trafiamy w union do jakiejś krawędzi w  $v$  to robimy tak, że jeśli mamy  $\text{Union}(v, w)$  to jeśli przed unionem w  $\text{find}(w) = \text{find}(v)$  to nie robimy nic, a w.p.p jeśli  $w$  było ojcem swojego uniona to dla każdego syna aktualizujemy  $v$  na maxa z obecnych, a jeśli nie było korzeniem swojego uniona to idziemy do ojca i robimy to z ojcem.