AISD lista 1

Dominik Szczepaniak

July 10, 2024

1 Zadanie 1

Zakładam, że T jest podane jako

struct Node

Node l = null;

Node r = null;

int value;

;

wpp. powyzsze zadanie nie ma troche sensu

a)

Chcemy to zrobic rekurencyjnie wiec:

```
int ile_wierzcholkow(Node t){
    if(t == null){
        return 1;
    }
    return ile_wierzcholkow(t.l) + ile_wierzcholkow(t.r) + 1;
}
```

Dlaczego to działa? Musimy wziąć siebie oraz wynik dla naszych synów. Jeśli nasi synowie są liścmi to zwrócimy po prostu 1, a w.p.p obliczymy rekurencyjnie wyniki dla naszych synów oraz jeszcze dodamy do tego siebie samego (+1).

b)

Jakie mamy możliwości najdłuższych odległości?

1. Najdłuższa droga to najgłębszy syn z lewego poddrzewa i najgłębszy syn z prawego poddrzewa od obecnego wierzchołka. Wtedy odległość to suma głębokości.

Dlaczego? Jeśli tak nie jest to inaczej najdłuższa droga będzie po prostu ścieżka poziom po poziomie w dół. Więc w szczególności będzie to też dawać odpowiedź, bo możemy wziąć wierzchołek w którym byśmy zaczęli tą ścieżkę i obliczyli $max_depth(t.l)$, a $max_depth(t.r) = 0$. Ale! Niech d będzie długością ścieżki. Jeśli $max_depth(t.l) ==$ d, a $max_depth(t.r) > 0$, to mamy najdłuższą drogę w lewym i prawym poddrzewie zakładanego wierzchołka.

```
int max_depth(Node t){
    if(t == null){
        return 0
    }
    left = max_depth(t.l)
    right = max_depth(t.r)
    return max(left, right) + 1;
}

int najwieksza_odleglosc(Node t){
    if(t == null){
        return 0;
    }
    lewo = max_depth(t.l)
    prawo = max_depth(t.r)
    wynik = lewo + prawo + 1;
    return max(wynik, max(najwieksza_odleglosc(t.l), najwieksza_odleglos
}
```

2 Zadanie 3

Dobra, jak robimy w ogóle topology sort? Jakie są znane algorytmy?

1. Algorytm Kahn'a

L <- lista po topo sorcie

S <- zbiór wierzchołków bez wchodzącej krawędzi

while S is not empty:

remove node n from S

add n to L

foreach node m with edge e from n to m:

remove edge e from graph

if m has no incoming edges:
add m to S
if graph has edges:
return error
else
return L

Jak zmodyfikować, żeby dawał porządek topologiczny? Wystarczy z S zawsze brać najmniejszy wartościowo Node aby zmniejszyć porządek topologiczny.

Czyli nasz S musi być zbiorem który wyrzuca liczby minimalne na samą górę. Można to po prostu zrobić kopcem, ale to też zależy od implementacji zbioru w różnych językach np. w C++ bazowo set jest już kopcem minimalnym, więc wystarczy brać pierwszy element z S i jest to minimalny element.

No ale nie działamy w języku, więc po prostu niech S będzie kopcem minimalnym. Szybkość operacji się nie zmienia, po prostu jest dalej log(n)

Złożoność O(nlogn + m)

Czy da się lepiej? Nie. Druga opcja na topo sort to DFS, który dizała w O(n + m), ale usimy pamiętać wierzchołki też za pomocą kopca (multiset/priority queue), więc będzie ta sama złożoność.

3 Zadanie 4

Dobra pierwsze co mozna zrobic takiego podstawowego to jest dijkstra z wierzchołka v który znajdzie nam najkrótszą trase do kazdego innego wierzcholka.

Pozniej robimy sobie dp z bfs jednoczesnie:

fn dfs(u: vertex)
visited[u] = true;
for x: vecter in u:
if D[x] < D[u]:
if !visited[x]:
dfs(x);
dp[u] <- dp[u] + dp[x];
dp[v] = 1;
dfs(u)
println!("", dp[u]);

O(E * log(V) + V+E)

4 Zadanie 5

dp[node] - (długość najdłuższej scieżki która zaczyna się w node, ojciec)

długość ścieżki albo można przedłużyć z sąsiada albo jest już najdłuższa

dla przykladu:

a -> 3 -> 2

|

4 - 5 - 6 - 7

|

1

tutaj a wykona dfs na 3 i 4, te wykonaja glebiej. po skonczonym dfs wezmiemy najdluzsza odnoge i zaznaczymy ze ojcem tej odnogi jest a

```
use std::io::{self, BufRead};
use std::cmp::max;

fn dfs(node: usize, dp: &mut Vec<(u32, u32)>, adj: &Vec<Vec<usize>>, vis
    visited[node] = true;
    for &i in &adj[node] {
        if !visited[i] {
            dfs(i, dp, adj, visited);
            if 1 + dp[i].0 > dp[node].0{
                dp[node].0 = 1 + dp[i].0;
                dp[node].1 = i as u32;
            }
        }
    }
}

fn main() {
    let stdin = io::stdin();
    let mut handle = stdin.lock();
    let mut input = String::new();
    handle.read_line(&mut input).unwrap();
    let mut iter = input.split_whitespace();
    let n: usize = iter.next().unwrap().parse().unwrap();
```

```
    let m: usize = iter.next().unwrap().parse().unwrap();
    let mut visited = vec![false; n + 1];
    let mut adj = vec![vec![]; n + 1];
    let mut dp: Vec<(u32, u32)> = vec![(0, 0); n + 1];
    for _i in 1..=m {
        input.clear();
        handle.read_line(&mut input).unwrap();
        let mut iter = input.split_whitespace();
        let a: usize = iter.next().unwrap().parse().unwrap();
        let b: usize = iter.next().unwrap().parse().unwrap();
        adj[a].push(b);
    }

    for i in 1..=n {
        if !visited[i] {
            dfs(i, &mut dp, &adj, &mut visited);
        }
    }
    let mut ans = 0;
    let mut ktory = 0;
    for i in 1..=n {
        if dp[i].0 > ans {
            ans = dp[i].0;
            ktory = i;
        }
    }
    println!("{}", ans);
    let mut res = vec![];
    while ktory != 0 {
        res.push(ktory);
        ktory = dp[ktory].1 as usize;
    }
    res = res.into_iter().rev().collect();
    for i in (0..res.len()).rev() {
        print!("{} ", res[i]);
    }
}
```

Dlaczego działa?

Jeśli mamy wszystkich synów jakiegoś Node 1, to chcemy wybrać ten który ma najdłuższą drogę. Zapisujemy najdłuższą drogę w dp, więc wystarczy sprawdzić dp[syn] + 1 (odległość z 1 do syna).

najdłuższą ścieżke przypisujemy do drugiej wartości, zeby pamietac dokad sie szlo (na koncu to odwrocimy)

5 Zadanie 6

Mozemy usuwac tylko parami, wiec w najlepszej opcji usuniemy dokladnie polowe par z tablicy.

Zauwazmy tez ze mamy nastepujaca obserwacje:

Jesli a_i moze wykreslic a_j, to a_i moze wykreslic a_{j+1}, a_{j+2}, ..., a_n - bo tablica jest posortowana.

W takim razie zrobmy sobie dwa wskazniki. Niech jeden wskazuje poczatek tablicy, a drugi pierwszy element po srodku.

Przeprowadzamy nastepujacy algorytm:

Jesli mozna wykreslic wartosc z 1 wskaznika i drugiego to wykreslamy obie (dodajemy 1 do wyniku i zwiekszamy wskazniki o 1)

Jesli nie mozna wykreslic, bo wartosc 2 wskaznika jest za mala, to przesuwamy drugi wskaznik.

Dlaczego dostaniemy najlepsza mozliwa odpowiedz?

Jesli a_i nie moze wykreslic a_j, to w szczegolnosci a_{i+1} nie moze wykreslic a_j (bo jest wieksze), wiec a_j nie moze wykreslic nic, co juz nie zostalo wykreslone. (Jesli np. a_{i-1} moglo wykreslic a_j, ale wykreslilo a_{j-1}, to a_i nie wykresli a_{j-1}, wiec wynik jest ten sam).

W takim razie wykreslimy najlepszy mozliwy wynik.

Załóżmy, że jest para <a, b> która nie zostanie wykreślona, a może być (czyli b >= 2a)

Mamy takie przypadki:

1. a i b są w lewej tablicy:

Skoro b jest w lewej tablicy to w prawej sa liczby wieksze od b czyli tez mozliwe do wykreslenia z a - czyli a zostanie wykreslona

2. a w lewej b w prawej:

jeśli na lewo od a jest wiecej liczb niz na lewo od b to b zostanie wykreslone (bo liczby po lewo od a sa mniejsze, wiec b moze byc z nimi wykreslone)

jeśli na lewo od a jest mniej liczb niż na lewo od b to wykreślimy albo a albo b albo oba - czyli nie będzie tej pary

// 1 3 4 4 4 4 4 6 - oba

// a b

// 1 2 3 4 5 6 7 8 - wykreślone a z nie-b

// a b

// 1 5

// 2 6

// 5 6 7 8| 9 9 14 15 - wykreślone b z nie-a

// a b

3. oba w prawej:

wszystkie liczby z lewej tablicy mozna wykreslic z b, wiec b zostanie wykreslone

z każdego dostajemy sprzeczność, więc dowód działa

6 Zadanie 7

1. Liczymy floyem warshallem wszystkie najkrotsze sciezki dla grafu bez wszystkich v1, v2, ..., vk O(

$$n^3$$

)

disty na inf

disty do samych siebie na 0

i jazda to:

For k = 0 to n − 1

For i = 0 to n − 1

For j = 0 to n − 1

Distance[i, j] = min(Distance[i, j], Distance[i, k] + Distance[k, j])

where i = source Node, j = Destination Node, k = Intermediate Node

Mozemy isc od tylu i zamiast usuwac bedziemy dodawac.

Jak dodajemy nowy wierzcholek u to musimy obliczyc odleglosci (v, u) dla kazdego v. Teraz mozemy po prostu zaktualizowac odleglosci do starych wierzcholkow jako:

(x, y) = (x, u) + (u, y)

Odleglosci zapisujemy w 2 wymiarowym wektorze

Obliczanie odleglosci (v, u) jest w O(n+m)

Dla kzdego wierzcholka to jest O(n*(n+m))

Pozniej liczenie nowych odleglosci bedzie O(n^2) dla kazdego czyli O(n^3)

Liczenie D_j też O(n^2) i dla kazdego wierzcholka n razy czyli O(n^3)

Takze mamy O(n^3)

1. Liczymy floyem warshallem wszystkie najkrotsze sciezki dla grafu bez wszystkich v1, v2, ..., vk O(n^3)
2. Obliczanie odleglosci (v, u) dla kazdego nowego v O(n*(n+m))
3. Update odległosci O(n^3)
4. Liczenie D_j O(n^3)

7 Zadanie 8

Idea:

Trzymamy kopiec elementów minimalnych oraz zmienną która trzyma maxa. Na samym początku wrzucamy pierwszy element z każdej listy. Następnie idziemy z kopca wyciągamy element najmniejszy i go wyrzucamy, a na jego miejsce dodajemy kolejny element z listy z której element wyrzuciliśmy. Za każdym elementem sprawdzamy czy wartość max - min jest najmniejsza.

Pseudokod:

```
min_kopiec = utworz-kopiec()
miejsca = [0] * k
maxW = L[0][0]
for i in range(k):
    min_kopiec.dodaj(L[i][0], i)
    maxW = max(maxW, L[i][0])

wynik = maxW - min_kopiec.min()
while(True):
    if(miejsca[min_kopiec.min().second] == len(L[min_kopiec.min().second
        break #konczymy gdy skoncza sie liczby w jakiejs liscie
    k = min_kopiec.min().second
    min_kopiec.usun()
    miejsca[k]+=1
    min_kopiec.dodaj(L[k][miejsca[k]], k)
    maxW = max(maxW, L[k][miejsca[k]])
    wynik = min(wynik, maxW - min_kopiec.min().first)
return wynik
```

Dowód:

Lemat1: Jeśli breakujemy to nie będzie lepszego wyniku

Dowód lematu1: Jeśli zbreakowaliśmy to skończyły się liczby w jakiejś liście, więc obecna branie kolejnych liczb będzie tylko możliwie zwiększać max, wiec różnica max-min będzie się zwiększać, ponieważ min się nie zmienia, gdyż jest on ostatnią liczbą z jakiejś listy. Możemy więc zakończyć tą pętlę.

Lemat2: Jeśli mamy minimalny wynik to zawiera od najmniejsze liczby ze swojej listy, większe od minimum z tego wyniku.

Dowód: Załóżmy, nie wprost, że są w wyniku liczby które w swojej liście nie są pierwszym większym elementem od x. Wtedy jeśli zamienilibyśmy tą liczbę na pierwszą większą od x, to nasz wynik byłby potencjalnie mniejszy, ponieważ minimum mogłoby zmaleć. Jeśli mamy dwa elementy o takiej samej wartości to równie dobrze możemy go potraktować jako jeden o tej wartości.

W takim razie jeśli mamy wynik to musi on zawierać najmniejsze liczby ze swojej listy, większe od minimum z tego wyniku.

Na samym początku mamy wszystkie najmniejsze elementy, przez co pierwsza możliwość max - min jest obecnie rozpatrzona. Następnie przechodząc do kolejnych stanów będziemy usuwać najmniejszą liczbę, aby zwiększyć mina i potencjalnie przybliżyć go do maxa, a jednocześnie będziemy dodawać maxa.

Dlaczego jest to optymalne?

Załóżmy, że mamy lepszy wynik "B", od naszego wyniku "A". Czyli istnieje taki zbiór k liczb z każdej listy, którego różnica między min a maxem jest mniejsza niż nasza różnica. Oznaczmy tego mina przez x, a maxa przez y. Załóżmy, że jesteśmy w sytuacji w naszym algorytmie gdzie $min_k opiec$.min().first $==$ x. Wtedy z konstukcji algorytmu mamy, że każdy inny element w kopcu, oznaczmy go jako a, b, gdzie a to wartość i b to lista, jest minimalnym elementem w swojej liście który jest większy od x. Jest tak, ponieważ jeśli byłby jakiś element z który $x < z < a$, to z musiałby być minimem i zostać już usuniętym z kopca, a tak być nie może ponieważ x jest mniejszy od z. Więc a jest najmniejszym elementem w swojej liście, który jest większy od x. W takim razie jeśli w wyniku B było minimum x oraz jakieś jeszcze elementy, to z lematu 2 musiały być one pierwszymi liczbami z list większymi od x. W takim razie nasz algorytm jest optymalny, bo znajduje takie liczby.