Korešpondenčný seminár z programovania
XLII. ročník, 2041/42
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola zimnej časti

Fipo
(max. 12 b za popis, 8 b za program)

1. Kto nájde diamanty?

Čo bolo našou úlohou

Našou úlohou bolo prejsť n riadkov stringu (reťazca) a spočítať diamanty v nich.

Ako sme to urobili

Priamočiare a najoptimálnejšie riešene je, že kontrolujme, či i-tý znak aktuálneho riadku je `/`, $i+1$ - tý znak je `\` a či v nasledujúcom riadku i-tý znak je `\` a $i+1$ - tý znak je `/`. Treba si dať pozor, že v tomto prípade neprechádzame posledný riadok, pretože pod ním nie je už žiaden potenciálny diamant a prehladávanie by nám len hodilo chybu, to isté platí aj pre posledný znak v riadku.

Časová zložitosť

Časová zložitosť je $O(nm)$, pretože prejdeme n riadkov dva krát (iba posledný riadok prejdeme raz) a v každom riadku prejdeme znak 2 krát (okrem posledného znaku v každom riadku). Keďže v každom riadku je m znakov a násobenie alebo pripočítavanie konštánt v časovej zložitosti zanedbávame, dostaneme $O(nm)$.

Pamäťová zložitosť

Tu si treba dať pozor, že optimálna pamäťová zložitosť je $O(m)$, pretože nám stačí pamätať si 2 riadky po m znakov, keďže postupne načítavame riadky a nahradzujeme ich novými.

Príklad

Pozrime sa, ako by sme si poradili s prvým príkladom zo zadania.

```
10 10
......./\.
......\/.
./\.......
./\.......
.\/.......
.\/.......
......./\..
/\....\/\.
\/....\//.
..........
```

Načítame si zo vstupu prvý a druhý riadok a kontrolujeme znaky: Znak na prvom mieste (index 0) v prvom riadku je `.` na druhom mieste tiež a to isté aj v druhom riadku, takže tu diamant nie je. Prvý diamant čo nájdeme, je stále v prvom a druhom riadku. Na 8. mieste v prvom riadku je `/` a 9. mieste je `\`. Keďže toto nám sedí, pozrieme sa na druhý riadok a tam tiež sedí, pretože na 8. mieste v prvom riadku je `\` a 9. mieste je `/`. Máme teda prvý diamant a postupujeme ďalej tak, že načítame tretí riadok a prvý zahodíme.

Listing programu (Python)

```
R, C = map(int, input().split())

res = 0
```

```
line1 = "." * C
for _ in range(R):
    line2 = input()
    for x in range(C - 1):
        if line1[x:x+2] == "/\\" and line2[x:x+2] == "\\/":
            res += 1
    line1 = line2

print(res)
```

Listing programu (C++)

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int n, m;
    cin >> n >> m;
    vector<string> map(n);

    getline(cin, map[0]);  // reads the rest of the line
    for (int r = 0; r < n; r++) {
        getline(cin, map[r]);
    }

    int count = 0;
    for (int r = 0; r < n-1; r++) {
        for (int c = 0; c < m-1; c++) {
            if (map[r][c] == '/' && map[r][c+1] == '\\' && map[r+1][c] == '\\' && map[r+1][c+1] ==
↪  '/') {
                count++;
            }
        }
    }

    cout << count << "\n";
    return 0;
}
```

Dušan
(max. 12 b za popis, 8 b za program)

2. Orať je zábava

Bruteforce

Naivné riešenie je vyskúšať všetky možné základy číselných sústav a pre každý základ vyskúšať všetky možné poradia cifier. Počet rôznych cifier na vstupe je maximálne 62, takže počet rôznych poradí cifier je $62! \approx 3 \cdot 10^{85}$. S kombináciou so všetkými možnými základmi číselných sústav dostávame veľmi veľa možností, ktoré musíme všetky vyskúšať.

Zaujímavá myšlienka

Čo si môžme uvedomiť je, že nám bude stačiť číselná sústava so základom rovnakým, ako je počet rôznych cifier v čísle na vstupe. Menší základ nemôže byť, lebo potom by 2 symboly museli reprezentovať rovnakú cifru,

čo nemôže nastať. Viacej to taktiež nemôže byť. Označme počet rôznych cifier n a počet cifier m. Ak by bol základ n, potom symbol úplne naľavo by mal hodnotu rádovo n^{m-1}. Ak by bol základ väčší, tak by mal hodnotu rádovo $(n+1)^{m-1} > n^{m-1}$.

Optimálne riešenie

Teraz keď už vieme, aký základ má Merlinova číselná sústava, stačí nám zistiť priradenie cifier. To spravíme jednoducho tak, že najľavejšej cifre priradíme najmenšiu hodnotu (hodnotu 1, lebo číslo sa nemôže začínať nulou). Potom nasledujúca cifra bude mať druhú najmenšiu hodnotu. Takže posledná cifra bude mať hodnotu $n-1$.

Listing programu (Python)

```python
T = int(input())
for t in range(T):
    sifra = input()
    kluc = {}

    ktore = 0
    for znak in sifra:
        if znak not in kluc:
            if ktore == 0:
                kluc[znak] = 1
            elif ktore == 1:
                kluc[znak] = 0
            else:
                kluc[znak] = ktore
            ktore += 1

    odpoved = 0
    dlzka_sifry = len(sifra)
    zaklad = max(2, len(kluc))
    for i in range(dlzka_sifry):
        odpoved += kluc[sifra[i]]*zaklad**(dlzka_sifry-i-1)
    print(odpoved)
```

Listing programu (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);

    int T;
    cin >> T;

    while (T--) {
        string sifra;
        cin >> sifra;

```

```
        int kolkate = 0;

        map<char, int> kluc;
        for (auto ch : sifra) {
            if (kluc.count(ch) == 0) {
                if (kolkate == 0)
                    kluc[ch] = 1;
                else if (kolkate == 1)
                    kluc[ch] = 0;
                else
                    kluc[ch] = kolkate;
                kolkate++;
            }
        }

        ll zaklad = max((ll)2, (ll)kluc.size());
        ll dlzka_sifry = sifra.length();
        vector<ll> pows = {1};
        for (int e = 1; e < dlzka_sifry; ++e)
            pows.push_back(pows.back() * zaklad);

        ll odpoved = 0;
        for (int i = 0; i < dlzka_sifry; i++)
            odpoved += kluc[sifra[i]] * pows[dlzka_sifry - i - 1];
        cout << odpoved << "\n";
    }
}
```

Vladko

3. Málo miesta na párty

(max. 12 b za popis, 8 b za program)

Síce vstup vyzerá strašidelne, ale to vôbec nevadí, načítame ho tak ako je. Keďže budeme hľadať obsahy miestností, potrebujeme presný plán budovy. Keďže každé prázdne políčko (`' '`) má obsah 1, tak nám pre každú miestnosť stačí zrátať, koľko je v nej takých prázdnych políčok. Nakoniec len vezmeme dve susedné miestnosti, ktoré nám po ich spojení dajú najväčší možný obsah. Jednoduché, však? Ale poďme na to postupne.

Hor sa prvú sadu!

Keďže v prvej sade sú všetky miestnosti na jednom riadku, môžeme postupne prechádzať druhým riadkom vstupu a zaradom počítať obsahy jednotlivých miestností. Takže, spravíme si pole, kam si budeme postupne tieto obsahy ukladať a prechádzame vstup – ak narazíme na stenu (`'|'`), pridáme do poľa s obsahmi 0. Ak prídeme na prázdne políčko (`' '`) pripočítame k poslednému číslu v poli 1. Inak povedané, kým sme v i-tej miestnosti, tak k i-temu číslu v poli prirátavame 1 za každé prázdne políčko. Keď narazíme na stenu, vieme že nám skončila miestnosť a bude nasledovať ďalšia. Akurát, na začiatku nemusí byť po `'#'` hneď stena, takže si na začiatok musíme dať "umelo" do poľa s obsahmi 0. Nakoniec už iba prejdeme toto pole obsahov, nájdeme 2 susedné miestnosti s najväčším súčtom a tento súčet vypíšeme.

Ostatné sady

Ako vyrátať obsah miestnosti?

Na vyrátanie obsahu nejakej miestnosti nám stačí začať prehľadávanie do hĺbky (DFS) z nejakého bodu vnútri tejto miestnosti a zrátať počet políčok, cez ktoré prejdeme. Toto budeme v najhoršom prípade musieť urobiť pre každý bod v budove, teda v čase $O(nm)$.

Pomalé riešenie

Postupne prejdeme všetkými zbúrateľnými stenami a vypočítame obsah miestnosti napravo a naľavo, resp.

hore a dole, podľa toho aký typ steny sme zbúrali. (!!!POZOR, treba ošetriť prípad, v ktorom je z oboch strán steny rovnaká miestnosť. Po zbúraní takejto steny ostane obsah miestnosti nezmenený!!!) Súčet týchto obsahov je obsah misetnosti, ktorá vznikne po zbúraní tejto steny. Doterajší najväčší obsah si budeme pamätať a po preskúmaní všetkých zbúrateľných stien vypíšeme najväčší obsah miestnosti po zbúraní nejakej steny. Časová zložitosť tohoto riešenia je $O((nm)^2)$, pretože pre každú zbúrateľnú stenu, ktorých je rádovo nm, sme rátali obsah susedných miestností, čo je rádovo taktiež nm.

Pamäťová zložitosť tohoto riešenia je $O(nm)$, pretože si potrebujeme pamätať vstup.

Optimálne riešenie

Všimnime si, že obsah nejakej miestnosti počítame toľkokrát, koľko má zbúrateľných stien, aj napriek tomu, že jej obsah sa za celý čas nezmenil.

Preto ešte predtým ako budeme iterovať zbúrateľnými stenami, si predpočítame obsahy daných miestností.

Potom, keď budeme iterovať cez všetky zbúrateľné steny, si vieme vyrátať obsah miestnosti po zbúraní steny v $O(1)$.

Predpočítať si obsahy miestností vieme znova pomocou DFS, no skôr ako začneme, musíme ešte: - Vytvoriť pole(1), do ktorého si budeme ukladať už vypočítané obsahy miestností. - Vytvoriť 2D pole(2), ktoré bude mať rozmery ako plánik budovy. V ňom si do každého bodu zapíšeme, na ktorom indexe v poli(1) je uložený obsah miestnosti, ktorá obsahuje tento bod.

Teraz už môžeme predpočítavať. Všimnime si, že skutočne prejdeme každým bodom v plániku nanajvýš raz, nakoľko nemusíme spúšťať DFS (hľadanie obsahu) z bodov, pre ktoré už v poli(2) existuje nejaký index. Preto bude mať táto časť algoritmu časovú zložitosť $O(nm)$.

Časová zložitosť tohoto riešenia je $O(nm)$, pretože sme najprv predpočítali obsahy miestností v $O(nm)$, nakoľko sme prešli celý obsah budovy, a potom pre každú zbúrateľnú stenu, ktorých je rádovo nm, sme rátali obsah susdených miestností v $O(1)$. Pamäťová zložitosť je $O(nm)$, pretože si potrebujeme pamätať vstup (nm), pole(1) (nanajvýš nm), pole(2) (nm).

O algoritme DFS sa môžete dozvedieť viac tu: https://www.ksp.sk/kucharka/dfs/

Listing programu (Python)

```
# ulozim obsahy zistene dfs do pola group
# prejdem vsetky policka, a skusam rozbijat steny
# pozriem okolo nej a poscitavam obsahy, ak este neboli zapocitane

import sys

sys.setrecursionlimit(10**6)

n, m = map(int, input().split())
inp = [[j for j in input()] for _ in range(2 * n + 1)]

group = [[0 for _ in range(2 * m + 1)] for _ in range(2 * n + 1)]
curr_group = 1
areas = [0, 0]


def inside(i, j):
    return 0 <= i < 2 * n + 1 and 0 <= j < 2 * m + 1


def dfs(i, j):
    group[i][j] = curr_group
    areas[curr_group] += 1

    # vnor sa hlbsie, ked je to vnutri
    if inside(i + 2, j) and group[i + 2][j] == 0 and inp[i + 1][j] == ".":
```

```
        dfs(i + 2, j)
    if inside(i - 2, j) and group[i - 2][j] == 0 and inp[i - 1][j] == ".":
        dfs(i - 2, j)
    if inside(i, j + 2) and group[i][j + 2] == 0 and inp[i][j + 1] == ".":
        dfs(i, j + 2)
    if inside(i, j - 2) and group[i][j - 2] == 0 and inp[i][j - 1] == ".":
        dfs(i, j - 2)

    return


for i in range(1, 2 * n + 1, 2):
    for j in range(1, 2 * m + 1, 2):
        if inp[i][j] == " " and group[i][j] == 0:
            dfs(i, j)
            curr_group += 1
            areas.append(0)


ans = max(areas)
for i in range(1, 2 * n):
    for j in range(1, 2 * m):
        if inp[i][j] == "|":
            if group[i][j - 1] != group[i][j + 1]:
                ans = max(ans, areas[group[i][j - 1]] + areas[group[i][j + 1]])
        elif inp[i][j] == "-":
            if group[i - 1][j] != group[i + 1][j]:
                ans = max(ans, areas[group[i - 1][j]] + areas[group[i + 1][j]])

print(ans)
```

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    cin.tie(0)->sync_with_stdio(0);

    int N, M;
    cin >> N >> M;
    cin.ignore();

    int Vn = N * M;
    vector<vector<int>> Eex(Vn), Erm(Vn);
    auto convert = [M](int x, int y) { return y * M + x; };
    for (int y = 1; y < 2 * N + 1; y += 2) {
        string labove, lcurr;
        getline(cin, labove);
        getline(cin, lcurr);
        for (int x = 1; x < 2 * M + 1; x += 2) {
```

```
            int v = convert(x / 2, y / 2);
            int vx = convert(x / 2 - 1, y / 2);
            int vy = convert(x / 2, y / 2 - 1);
            if (x > 1) {
                if (lcurr[x - 1] == '.') {
                    Eex[v].push_back(vx);
                    Eex[vx].push_back(v);
                } else {
                    Erm[v].push_back(vx);
                    Erm[vx].push_back(v);
                }
            }
            if (y > 1) {
                if (labove[x] == '.') {
                    Eex[v].push_back(vy);
                    Eex[vy].push_back(v);
                } else {
                    Erm[v].push_back(vy);
                    Erm[vy].push_back(v);
                }
            }
        }
    }

    vector<int> C(Vn, -1);
    auto dfs = [&](auto&& dfs, int v, int c) -> void {
        C[v] = c;
        for (int u : Eex[v])
            if (C[u] == -1)
                dfs(dfs, u, c);
    };
    vector<int> Sz;
    for (int v = 0; v < Vn; ++v) {
        if (C[v] == -1) {
            dfs(dfs, v, Sz.size());
            Sz.push_back(0);
        }
        ++Sz[C[v]];
    }

    int res = *max_element(Sz.begin(), Sz.end());
    for (int v = 0; v < Vn; ++v)
        for (int u : Erm[v])
            if (C[v] != C[u])
                res = max(res, Sz[C[v]] + Sz[C[u]]);

    cout << res << '\n';
}
```

4. Búrka

Filip Siviček
(max. 12 b za popis, 8 b za program)

Bruteforce

Bruteforce môžeme spraviť tak, že si odsimulujeme tok vody z každého políčka. Pre každé políčko si zapamätáme, aké je jeho koncové políčko. Všetky políčka, čo stekajú na rovnaké miesto budú v rovnakej oblasti.

Keď toto spravíme pre všetky políčka, tak už nám iba stačí prideliť čísla oblastiam. Čísla oblastiam priraďujeme tak, že pri prechode poľom (zľava doprava, zhora dole) si pamätáme, na ktorú oblasť sme narazili ako prvú. Rôzne oblasti odlišujeme podľa toho, že políčka v nich stekajú na rôzne koncové miesta.

Priradiť koncové políčko jednému políčku vieme v čase $O(nm)$. Totiž, prinajhoršom budeme musieť prejsť pre toto políčko celé pole. To znamená, že časová zložitosť tohto riešenia je $O(n^2m^2)$.

Pre každé políčko si potrebujeme pamätať, aké je jeho koncové políčko. Taktiež si potrebujeme pre každé koncové políčko pamätať, aké je číslo oblasti, ktorá mu prináleží. Vieme však, že koncových políčok môže byť najviac nm To znamená, že toto riešenie má pamäťovú zložitosť $O(nm)$.

Zaujímavá myšlienka

Povedzme, že pri hľadaní koncového políčka pre políčko $CH[i][j]$ sme prešli políčkom $CH[k][l]$, pre ktoré už vieme, kam z neho steká voda. Koncové políčko pre $CH[i][j]$ musí teda byť rovnaké ako koncové políčko pre $CH[k][l]$. To kvôli tomu, že stekanie vody je jednoznačné - všetci susedia, ktorí stekajú na políčko $CH[k][l]$ budú z neho ďalej stekať na rovnaké miesto, ako steká ono samo.

Lepšie riešenie

Bruteforce by sme mohli vylepšiť tak, že by sme políčkami prechádzali od najnižšieho k najvyššiemu. Keď by sme sa potom dostali k nejakému políčku $CH[i][j]$, tak by mohli nastať 2 prípady: 1) Všetci jeho susedia sú vyšší. V tomto prípade z neho voda nemá kam stekať. To znamená, že jeho koncovým políčkom bude ono samo. 2) Nejaký jeho sused je od neho nižší. Keďže políčka riešime od najnižšieho, tak už máme vyriešených všetkých jeho nižších susedov. Teda tomuto políčku priradíme koncové políčko suseda, na ktoré steká ono samo.

Každé políčko vieme riešiť v konštantom čase (pozrieme sa na jeho 4 susedov). Keďže políčka riešime od najnižšieho k najvyššiemu, tak si potrebujeme políčka utriediť podľa výšky. Na konci potrebujeme ešte raz prejsť poľom, aby sme priradili číslo oblasti. To znamená, že časová zložitosť tohto riešenia je $O(nm \log(nm))$.

Potrebujeme si pamätať políčka zo vstupu a potrebujeme ďalšie pole, v ktorom si ich budeme pamätať utriedené podľa výšky. To znamená, že toto riešenie bude mať pamäťovú zložitosť $O(nm)$.

Optimálne riešenie

Náš bruteforce sa dá vylepšiť aj inak ako tak, že budeme prechádzať políčkami od najnižšieho po najvyššie. Povedzme, že potrebujeme zistiť, kam steká políčko $CH[i][j]$. Budeme to simulovať. Pri našej simulácii si však budeme v poli pamätať, ktoré všetky políčka sme navštívili, keď voda stekala z políčka $CH[i][j]$. Toto druhé pole si nazvime `tok`. Povedzme, že sme počas simulácie toku z políčka $CH[i][j]$ natrafili na políčko $CH[k][l]$, ktoré už máme vyriešené, alebo z neho nemá kam stekať voda. To znamená, že všetky políčka v simulovanom toku budú mať rovnaké koncové políčko ako $CH[k][l]$.

Prejsť celým poľom vieme v čase $O(nm)$. Ak by sme natrafili na políčko, ktoré máme už vyriešené (lebo bolo v toku iného políčka), tak ho môžeme preskočiť a ísť spracovávať ďalšie. To znamená, že toto riešenie má časovú zložitosť $O(nm)$.

Pre každé políčko si potrebujeme pamätať, aké je jeho koncové políčko. Taktiež si potrebujeme pamätať `tok`, ale ten môže mať dĺžku najviac nm. To znamená, že toto riešenie má pamäťovú zložitosť $O(nm)$.

Listing programu (Python)

```
n, m = [int(x) for x in input().split()]
p = [[0, 1], [1, 0], [0, -1], [-1, 0]]
inf = 1000000001
heights = [[inf for i in range(m + 2)] for j in range(n + 2)]

for i in range(1, n + 1):
    riadok = [int(x) for x in input().split()]
    for j in range(1, m + 1):
```

```
        heights[i][j] = riadok[j - 1]

odtoky = [[-1 for i in range(m + 1)] for j in range(n + 1)]

for i in range(1, n + 1):
    for j in range(1, m + 1):
        min = heights[i][j]
        smer = -1
        for k in range(4):
            if heights[i + p[k][0]][j + p[k][1]] <= min:
                smer = k
                min = heights[i + p[k][0]][j + p[k][1]]
        if min == heights[i][j]:
            smer = -1
        odtoky[i][j] = smer

ans = [[0 for i in range(m + 1)] for j in range(n + 1)]
kod_oblasti = 1

for i in range(1, n + 1):
    for j in range(1, m + 1):
        x_now = i
        y_now = j
        tok = []
        while odtoky[x_now][y_now] != -1 and ans[x_now][y_now] == 0:
            tok.append([x_now, y_now])
            x_next = x_now + p[odtoky[x_now][y_now]][0]
            y_next = y_now + p[odtoky[x_now][y_now]][1]
            x_now = x_next
            y_now = y_next
        tok.append([x_now, y_now])
        if not ans[x_now][y_now]:
            ans[x_now][y_now] = kod_oblasti
            kod_oblasti += 1
        fill = ans[x_now][y_now]
        for k in range(len(tok)):
            ans[tok[k][0]][tok[k][1]] = fill

for i in range(1, n + 1):
    for j in range(1, m):
        print(ans[i][j], end=" ")
    print(ans[i][m])
```

Listing programu (C++)

```
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;
typedef long long ll;
const ll inf = 1000000001;

```

```cpp
int main(){
    //ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
    int n, m; // n, m <= 5*10^2
    cin >> n >> m;
    // 0 <= hodnoty na vstupe <= 10^9;
    ll heights[n+2][m+2];
    int p[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}}; //zmenil som si poradie priorit: S, Z, J, V
    for (int i = 0; i < n+2; i++){
        heights[i][0] = inf;
        heights[i][m+1] = inf;
    }
    for (int i = 0; i < m+2; i++){
        heights[0][i] = inf;
        heights[n+1][i] = inf;
    }

    for (int i = 1; i <= n; i++){
        for (int j = 1; j <= m; j++){
            cin >> heights[i][j];
        }
    }

    int odtoky[n+1][m+1];
    for (int i = 1; i <= n; i++){
        for (int j = 1; j <= m; j++){
            ll min = heights[i][j];
            int smer = -1;
            for (int k = 0; k < 4; k++){
                if (heights[i+p[k][0]][j+p[k][1]] <= min){
                    smer = k;
                    min = heights[i+p[k][0]][j+p[k][1]];
                }
            }
            if (min == heights[i][j]){
                smer = -1;
            }
            odtoky[i][j] = smer;
        }
    }

    ll ans[n+1][m+1];
    for (int i = 0; i < n+1; i++){
        for (int j = 0; j < m+1; j++){
            ans[i][j] = 0;
        }
    }
    ll kod_oblasti = 1;
    for (int i = 1; i <= n; i++){
        for (int j = 1; j <= m; j++){
            int x_now = i;
            int y_now = j;
            vector<vector<int>> tok;
            while (odtoky[x_now][y_now] != -1 && (ans[x_now][y_now] == 0)){
```

```
                tok.push_back({x_now, y_now});
                int x_next = x_now + p[odtoky[x_now][y_now]][0];
                int y_next = y_now + p[odtoky[x_now][y_now]][1];
                x_now = x_next;
                y_now = y_next;
            }
            tok.push_back({x_now, y_now});
            if (!ans[x_now][y_now]){
                ans[x_now][y_now] = kod_oblasti;
                kod_oblasti++;
            }
            ll fill = ans[x_now][y_now];
            for (ll k = 0; k < tok.size(); k++){
                ans[tok[k][0]][tok[k][1]] = fill;
            }
        }
    }
    for (int i = 1; i <= n; i++){
        for (int j = 1; j < m; j++){
            cout << ans[i][j] << " ";
        }
        cout << ans[i][m] << "\n";
    }
    //vrati to tu mapu ako jeden string
}
```

danza

5. Analýza kombajnových úrodičov

(max. 12 b za popis, 8 b za program)

V tejto úlohe sme dostali pole n čísiel. Našou úlohou bolo vypočítať súčet rozdielov maximálnej a minimálnej hodnoty cez všetky neprázdne úseky tohto poľa.

Priamočiare riešenie

Najjednoduchším spôsobom, ako sa dostať k výsledku, je postupne prejsť všetky neprázdne úseky. V každom úseku nájdeme maximum a minimum, odčítame ich a pripočítame ku výsledku.

Koľko úsekov existuje? Úsek môže mať n rôznych začiatkov a ku každému začiatku rádovo n možných koncov. Úsekov je teda $O(n^2)$. Pre každý úsek musíme nájsť maximum a minimum, čo vieme spraviť jednoduchým prechodom cez celý úsek. Celková časová zložitosť tohto riešenia je teda $O(n^3)$.

Pamätať si potrebujeme vstupné pole veľkosti n. Pamäťová zložitosť je teda $O(n)$.

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;

int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i];
```

```
    }

    ll res = 0;
    for (int left = 0; left < n; left++) {
        for(int right = left; right < n; right++) {
            ll mn = 1000000000, mx = -1;
            for (int i = left; i <= right; i++) {
                mn = min(mn, (ll)v[i]);
                mx = max(mx, (ll)v[i]);
            }
            res += mx - mn;
        }
    }

    cout << res << '\n';

    return 0;
}
```

Listing programu (Python)

```
def main():
    n = int(input())
    v = list(map(int, input().split()))

    res = 0
    for left in range(n):
        for right in range(left, n):
            mn = float('inf')
            mx = -1
            for i in range(left, right + 1):
                mn = min(mn, v[i])
                mx = max(mx, v[i])
            res += mx - mn

    print(res)


main()
```

Drobné vylepšenie

Predchádzajúce riešenie vieme ľahko zlepšiť tak, aby malo časovú zložitosť $O(n^2)$. Nebudeme prechádzať každý úsek od začiatku – ak už totiž poznáme minimum a maximum v úseku $[i, j]$, vieme v čase $O(1)$ zistiť minimum a maximum v úseku $[i, j+1]$.

Týmto spôsobom dostávame riešenie, ktoré má rovnakú zložitosť ako je počet všetkých úsekov, teda $O(n^2)$. Ak ale chceme lepšiu časovú zložitosť, nebudeme už môcť prechádzať všetky úseky…

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;
```

```cpp

typedef long long ll;

int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }

    ll res = 0;
    for (int left = 0; left < n; left++) {
        ll mn = 1000000000, mx = -1;
        for(int right = left; right < n; right++) {
            mn = min(mn, (ll)v[right]);
            mx = max(mx, (ll)v[right]);
            res += mx - mn;
        }
    }

    cout << res << '\n';

    return 0;
}
```

Listing programu (Python)

```python
def main():
    n = int(input())
    v = list(map(int, input().split()))

    res = 0
    for left in range(n):
        mn = 1000000000
        mx = -1
        for right in range(left, n):
            mn = min(mn, v[right])
            mx = max(mx, v[right])
            res += mx - mn

    print(res)


main()
```

Vzorové riešenie

Prvým pozorovaním je, že výsledok vieme získať ako súčet maxím cez všetky úseky mínus súčet miním cez všetky úseky. Budeme sa teda ďalej snažiť vypočítať tieto dve hodnoty. Ak vieme počítať súčet maxím, súčet miním budeme vedieť počítať analogicky. Pozrime sa teda na to, ako vieme efektívne vypočítať súčet maxím cez všetky úseky.

Často používanou technikou, ktorá bude užitočná aj pre nás, je technika *príspevku k súčtu*. Spočíva v tom, že pre každý prvok zistíme, koľko on sám prispeje ku výsledku. V našom prípade by sme pre každý prvok chceli

vedieť, v koľkých úsekoch poľa bude práve on maximom. Ak je prvok v_i maximom v k rôznych úsekoch, tak ku celkovému súčtu maxím cez všetky úseky prispeje hodnotou $k \cdot v_i$. Pozrime sa teda na to, ako pre každý prvok zistiť, v koľkých úsekoch je maximom.

Ale ešte predtým sa trochu zamyslime. Prvky poľa sa môžu opakovať. Môže teda nastať situácia, že nejaký úsek bude mať viacero maximálnych prvkov. Nemôže to spôsobiť, že pre daný úsek pripočítame všetky tieto maximá? Skutočne by sa nám to mohlo stať. Potrebujeme teda jednoznačne určiť, ktorý maximálny prvok budeme považovať za správny. Preformulujme teda to, čo hľadáme. Nebudeme pre každý prvok zisťovať, v koľkých úsekoch je maximom, ale v koľkých úsekoch je práve on tým **najľavejším** maximom.

Na to nám stačí vedieť dve veci:

- Ako ďaleko od daného prvku môžeme ísť **doľava**, kým narazíme na **väčší, alebo rovnaký** prvok.
- Ako ďaleko od daného prvku môžeme ísť **doprava**, kým narazíme na **ostro väčší** prvok.

To už nám stačí na výpočet počtu úsekov. Totiž, ak od prvku v_i vieme týmto spôsobom ísť doľava o l_i a doprava o r_i políčok, tak daný prvok bude najľavejším maximom v každom úseku, ktorý začína niekde na políčkach $i - l_i$ až i a končí niekde na políčkach i až $i + r_i$. Týchto úsekov je zjavne $(l_i + 1) \cdot (r_i + 1)$. Náš prvok teda k celkovému súčtu maxím prispeje hodnotou $v_i \cdot (l_i + 1) \cdot (r_i + 1)$.

Zostáva vypočítať hodnoty l_i a r_i. Keďže ich počítanie je analogické (až na ostrosť nerovnosti), ukážeme si iba ako spočítať hodnoty l_i. Stačí nám na to jeden zásobník. Prvky poľa budeme prechádzať zľava doprava. Pozrime sa, ako spracujeme i-ty prvok:

Kým je na vrchu zásobníka číslo j také, že $v_j < v_i$, tak ho zo zásobníka odstránime. Tým zo zásobníka postupne odstránime niekoľko (možno nula) prvkov. Ak sme zásobník úplne vyprázdnili, znamená to, že všetky prvky naľavo od i-teho boli menšie ako on. Ak sme zásobník nevyprázdnili, na jeho vrchu zostala najbližšia pozícia naľavo od i, kde sa nachádza prvok väčší alebo rovný v_i. Z tohto vieme odčítať hľadanú hodnotu l_i. Následne na vrch zásobníka pridáme i a posunieme sa na spracovanie ďalšieho prvku rovnakým spôsobom.

Nemôže byť problematické, že sme zo zásobníka niektoré prvky odstránili? Nie! Na zásobník sme totiž po spracovaní i-teho prvku pridali číslo i. Ak pri spracovaní nejakého ďalšieho prvku zo zásobníka odstránime i, tak odstraňujeme aj všetky prvky s menšími hodnotami, než v_i. Takže všetky čísla, ktoré sme odstránili pri spracovaní i-teho prvku, by sme odstránili tak či tak. Problematické to teda byť nemôže.

Celý algoritmus na vypočítanie l_i potrebuje iba jeden prechod cez pole. Každý prvok bude najviac raz pridaný na zásobník a najviac raz zo zásobníka odstránený. Časová zložitosť tohto prechodu je teda $O(n)$. My potrebujeme vypočítať hodnoty l_i aj r_i pre zistenie súčtu maxím a následne pre zistenie súčtu miním. To sú 4 takéto prechody cez pole. Celková časová zložitosť tohto riešenia je teda $O(n)$.

Pamätať si potrebujeme iba vstupné pole a zásobník. Pamäťová zložitosť riešenia je preto tiež $O(n)$.

Listing programu (C++)

```cpp
#include <bits/stdc++.h>

using namespace std;

// Pre prvok chceme zratat, pokial dolava a doprava mozeme od neho ist, aby on bol NAJLAVEJSIM minimom (resp. maximom)
// Preto ak ideme zlava, chceme pouzit ostru rovnost, ale ked ideme sprava, tak neostru.
bool should_pop(const string& side, const string& function, int a, int b) {
    if (side == "left") {
        if (function == "min") {
            return a > b;
        } else {
            return a < b;
        }
    } else {
        if (function == "min") {
            return a >= b;
        } else {
            return a <= b;
```

```
        }
    }
}

vector<int> calc_boundaries(vector<int> v, const string& side, const string& function) {
    int n = v.size();
    vector<int> boundary(n);

    // Ak chceme prave hranice, tak si otocime vstupne pole a zratame lave hranice
    // Toto je maly trik, aby sme neprogramovali takmer to iste dvakrat
    if (side == "right") {
        reverse(v.begin(), v.end());
    }

    // Prechadzajme polom zlava doprava
    // Pouzime stack na zratanie toho, po ktory index nalavo bude najmensim (resp. najvacsim)
    ↪ prvkom
    stack<int> s;
    for(int i = 0; i < v.size(); i++) {
        while(!s.empty() && should_pop(side, function, v[s.top()], v[i])) {
            s.pop();
        }
        if(s.empty()) {
            boundary[i] = -1;
        } else {
            boundary[i] = s.top();
        }
        s.push(i);
    }

    // Ak sme pocitali prave hranice, tak sme na zaciatku otocili pole
    // Musime teda "zrkadlovo" otocit aj vysledok
    if (side == "right") {
        for (int i = 0; i < n; i++) {
            boundary[i] = n - 1 - boundary[i];
        }
        reverse(boundary.begin(), boundary.end());
    }

    return boundary;
}

int main() {
    int n;
    cin >> n;
    vector<int> v(n);

    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }

    vector<string> functions = {"min", "max"};
    long long result = 0;
```

```
    for (auto &function: functions) {
        vector<int> left = calc_boundaries(v, "left", function);
        vector<int> right = calc_boundaries(v, "right", function);
        for (int i = 0; i < n; i++) {
            long long elems_to_the_left = i - left[i] - 1;
            long long elems_to_the_right = right[i] - i - 1;
            long long this_elem_subsegments = (elems_to_the_left + 1) * (elems_to_the_right + 1);
            long long sum_values_for_segments = v[i] * this_elem_subsegments;
            if (function == "max") {
                result += sum_values_for_segments;
            } else {
                result -= sum_values_for_segments;
            }
        }
    }

    cout << result << '\n';

    return 0;
}
```

Listing programu (Python)

```
from typing import List


def should_pop(side: str, function: str, a: int, b: int) -> bool:
    if side == "left":
        if function == "min":
            return a > b
        else:
            return a < b
    else:
        if function == "min":
            return a >= b
        else:
            return a <= b


def calc_boundaries(v: List[int], side: str, function: str) -> List[int]:
    n = len(v)
    boundary = [0] * n

    if side == "right":
        v = v[::-1]

    s = []
    for i in range(n):
        while s and should_pop(side, function, v[s[-1]], v[i]):
            s.pop()
        if not s:
            boundary[i] = -1
```

```
        else:
            boundary[i] = s[-1]
        s.append(i)

    if side == "right":
        boundary = [n - 1 - x for x in boundary]
        boundary = boundary[::-1]

    return boundary


def main() -> None:
    n = int(input())
    v = list(map(int, input().split()))

    functions = ["min", "max"]
    result = 0

    for function in functions:
        left = calc_boundaries(v, "left", function)
        right = calc_boundaries(v, "right", function)
        for i in range(n):
            elems_to_the_left = i - left[i] - 1
            elems_to_the_right = right[i] - i - 1
            this_elem_subsegments = (elems_to_the_left + 1) * (elems_to_the_right + 1)
            sum_values_for_segments = v[i] * this_elem_subsegments
            if function == "max":
                result += sum_values_for_segments
            else:
                result -= sum_values_for_segments

    print(result)


main()
```

Janči

6. Just načas

(max. 12 b za popis, 8 b za program)

Čo chceme zistiť

Úloha sa nás pýta, koľko najmenej sypaní Janči nestihne. To je celkom zvláštna formulácia otázky, ktorá sa ale vlastne pýta len to, koľko najviac sypaní stihne – keďže poznáme celkový počet sypaní.

Ako to zistíme

Keďže deň je naozaj dlhý, nestihneme simulovať, kde sa traktor môže nachádzať v každom možnom čase. To nám však nevadí, pretože väčšina časov je pre nás úplne nezaujímavá – jediné podstatné sú tie, v ktorých nejaký kombajnista vysype náklad.

To, či v danom čase stihneme náklad zobrať, ale záleží na tom, kde sa traktor nachádzal predtým. Dopredu pritom nevieme, či sa nám náklad oplatí zobrať, alebo ísť počas tej doby radšej niekam inam.

Na takýto typ problému sa teda hodí použiť dynamické programovanie – nebudeme sa snažiť zistiť, ktorú postupnosť sypaní je najlepšie stihnúť, ale pre každé sypanie sa pozrieme, koľko najviac sypaní by sa dalo stihnúť, ak by sme sa rozhodli stihnúť aj to aktuálne. Nakoniec sa pozrieme na celkovo najlepší výsledok zo všetkých možných sypaní.

V čom to spočíva?

Aby dynamické programovanie fungovalo, musíme si vedieť v každej časti úlohy – stave – pomôcť nejakou časťou, ktorú už sme predtým vyriešili. Keďže sa na sypania pozeráme chronologicky, vieme, že pri riešení konkrétneho sypania sme už všetky predošlé vyriešili optimálne, a tie budúce nás zatiaľ zaujímať nemusia, lebo každé z nich budeme riešiť zvlášť neskôr. Preto nám stačí zistiť, po ktorých z predošlých sypaní sa dá stihnúť to aktuálne a z nich vybrať také, kde sme predtým stihli najviac. K tomuto najlepšiemu výsledku pripočítame 1 (že sme stihli aj aktuálne) a uložíme si to ako najlepší výsledok pre aktuálne sypanie. Ukladať si počty stihnutých sypaní budeme do samostatného poľa, v ktorom bude pre každé sypanie výsledok na takom indexe, aký majú súradnice daného sypania v poli súradníc a časy v poli časov. Vďaka tomu sa jednoducho dostaneme ku všetkým informáciam o každom sypaní.

Ako zistiť, či je jedno sypanie stihnuteľné pred druhým? Potrebujeme, aby sa traktor vedel dostať z jedného miesta na druhé v čase najviac tak dlhom, ako je rozdiel časov sypaní. Keďže traktor sa pohybuje po štvorcovej mriežke, používame takzvanú Manhattanskú metriku – medzi každými dvoma miestami sa dá dostať tak, že najskôr ideme len horizontálne a potom len vertikálne. Takáto cesta je zároveň rovnako dobrá, ako ktorákoľvek, ktorá sa nevracia (opačným smerom, než už predtým šla), a je zároveň najlepšia možná. Vzdialenosť teda spočítame ako absolútnu hodnotu rozdielov jednej súradnice plus absolútnu hodnotu rozdielov druhej súradnice jednotlivých sypaní.

Koľko to trvá?

Celý deň. Ahá, myslíte spočítať. Pardón. Pre každé sypanie potrebujeme preskúmať všetky predošlé sypania (lebo to zatiaľ najlepšie mohlo byť veľmi dávno), pričom zistiť pre sypanie, či sa dalo stihnúť a či je lepšie, než sme našli doteraz, zvládneme v konštantnom čase, takže to dokopy trvá $O(N^2)$. To nie je úplne zlé, ale prechádza to len na polovicu bodov!

Dá sa to zlepšiť?

Ako pri každej správnej šifre sa skúsme pozrieť, aké informácie sme ešte nepoužili. V zadaní sa píše o maximálnej ploche poľa, ktorá môže byť až 250 000 hektárov. To je až 250 000 štvorcov 100 krát 100 metrov. Pole je ale určite štvorcové, takže maximálny rozmer jednej jeho strany je 500 štvorcov. Všetky súradncice, na ktoré narazíme, tak možu mať len jednu z 500 rôznych hodnôt. To nám veľmi nepomôže, keďže súradnice si ukladáme ako čísla a problém s pamäťou nemáme.

Ale mohlo by nám pomôcť niečo, čo z toho vyplýva, a síce, že žiadne dva body nemajú vzdialenosť väčšiu, než $2 \cdot 500$ štvorcov. Takže sa najneskôr za 1000 minút dostaneme na ľubovoľné iné miesto na poli – alebo ešte skôr, záleží na veľkosti poľa, ktorú si označme r. To znamená, že keď sa pri aktuálnom sypaní pozeráme na sypania predošlé, nepotrebujeme nikdy ísť viac, než $2r$ minút do minulosti. Teda vždy sa stačí pozerať najviac na posledných $2r$ sypaní, keďže žiadne 2 neprebiehajú naraz.

Ako to použijeme?

Budeme si teda pamätať separátne pole zatiaľ celkovo najlepších výsledkov. Vždy, keď nájdeme najlepší výsledok pre dané sypanie, porovnáme ho s celkovo najlepším výsledkom doteraz a lepší výsledok si pridáme do tohto nového poľa. Vďaka tomu bude toto pole obsahovať neklesajúcu postupnosť čísel – takže najlepší výsledok dosiahnutý do nejakého sypania bude v tomto poli na indexe daného sypania.

Keď potom budeme pri každom aktuálnom sypaní prehľadávať sypania minulé, budeme to robiť len dovtedy, kým nenarazíme na nejaké, ktoré prebehlo od aktuálneho skôr o viac, než $2r$ minút. V momente, keď také nájdeme, už totiž vieme, že všetky skoršie vieme dosiahnuť, a zaujíma nás len to, ktoré z nich je najlepšie. Túto hodnotu však už poznáme – je v novom poli na indexe sypania, na ktoré sme práve narazili.

Pomôže to?

Rozhodne áno. Namiesto toho, aby sme museli pre každé sypanie prejsť všetky predošlé sypania (ktorých môže byť až n), stačí prejsť najviac r predošlých sypaní. Vieme, že $r = \sqrt{h}$. Časová zložitosť sa tak zlepší na $O(n * r)$, teda $O(n * \sqrt{h})$.

Pamäťová zložitosť sa nezhorší, pamätáme si len jedno pole naviac k tomu, čo sme si museli pamätať doteraz. To boli tiež len polia s časmi, súradnicami a najlepšími výsledkami pre jednotlivé sypania, takže celková pamäťová zložitosť je $O(n)$.

Listing programu (Python)

```python
def solve():
    h, n = map(int, input().split())
    r = int(h**(1 / 2)) # Spočítame dĺžku strany poľa - r

    # events sú jednotlivé časy a súradnice sypaní,
    # best_here je najlepší počet dosiahnuteľný, ak Janči
    # skončí na danom mieste v danom čase,
    # best_anywhere je najlepší počet,
    # ktorý v danom čase vieme dosiahnuť na ľubovoľnom mieste.
    # Všade pridáme aj začiatočnú pozíciu:
    # v čase 0 na súradniciach (0, 0) s 0 stihnutými kombajnistami.
    best_here = [0 for i in range(n + 1)]
    best_anywhere = [0 for i in range(n + 1)]
    events = [(0, 0, 0)]

    # Načítame časy a súradnice.
    for i in range(1, n + 1):
        time, x, y = map(int, input().split())
        events.append((time, x // 100, y // 100))

    # Pre každé sypanie sa pozeráme postupne na predošlé,
    # dokým nie je jasné, že vieme dosiahnuť všetky,
    # potom použijeme best_anywhere.
    # Inak sa snažíme použiť predošlé best_here,
    # ak sa dá dosiahnuť a oplatí sa to.
    for i in range(1, n + 1): # i je index aktuálneho sypania,
                              # ktoré sa práve snažíme vyriešiť.
        for j in range(i - 1, -1, -1): # j je postupne index všetkých
                                       # predošlých sypaní.
            if events[i][0] - events[j][0] > 2 * r:
                # Vieme dosiahnuť všetky, nemá zmysel hľadať ďalej.
                best_here[i] = max(best_here[i], best_anywhere[j] + 1)
                break
            if best_here[j] + 1 > best_here[i]:
                # Zatiaľ najlepšia možnosť...
                if (abs(events[i][1] - events[j][1]) +
                        abs(events[i][2] - events[j][2]) <=
                        events[i][0] - events[j][0]):
                        # ...a vieme ju dosiahnuť z aktuálnej pozície.
                    best_here[i] = best_here[j] + 1

        # Ak sme vedeli nejaký počet stihnúť predtým, vieme aj teraz.
        best_anywhere[i] = max(best_anywhere[i - 1], best_here[i])

    # Na konci nám je jedno, kde skončíme,
    # nemusíme pritom stihnúť posledné sypanie.
    print(n - best_anywhere[n])

solve()
```

Listing programu (C++)

```cpp
#include <vector>
#include <algorithm>
#include <iostream>
#include <cmath>

using namespace::std;

int main(){
    // times sú jednotlivé časy, coords súradnice sypaní,
    // best_here je najlepší počet dosiahnuteľný, ak Janči
    // skončí na danom mieste v danom čase,
    // best_anywhere je najlepší počet,
    // ktorý v danom čase vieme dosiahnuť na ľubovoľnom mieste.
    vector<int> times, best_here, best_anywhere;
    vector<pair<int, int>> coords;

    int h, n, x, y;
    cin >> h >> n;
    // Spočítame dĺžku strany poľa - r.
    int r = sqrt(h);

    // Všade pridáme aj začiatočnú pozíciu:
    // v čase 0 na súradniciach (0, 0) s 0 stihnutými kombajnistami.
    times = best_here = best_anywhere = vector<int>(n + 1, 0);
    coords = vector<pair<int,int>>(n + 1, 0);

    // Načítame časy a súradnice.
    for (int i = 1; i <= n; i++){
        cin >> times[i] >> x >> y;
        coords[i] = make_pair(x / 100, y / 100);
    }

    // Pre každé sypanie sa pozeráme postupne na predošlé,
    // dokým nie je jasné, že vieme dosiahnuť všetky,
    // potom použijeme best_anywhere.
    // Inak sa snažíme použiť predošlé best_here,
    // ak sa dá dosiahnuť a oplatí sa to.
    for (int i = 1; i <= n; i++){ // i je index aktuálneho sypania,
                                  // ktoré sa práve snažíme vyriešiť.
        for (int j = i - 1; j >= 0; j--){ // j je postupne index všetkých
                                          // predošlých sypaní.
            if (times[i] - times[j] > 2 * r){
                // Vieme dosiahnuť všetky, nemá zmysel hľadať ďalej.
                best_here[i] = max(best_here[i], best_anywhere[j] + 1);
                break;
            }
            if (best_here[j] + 1 > best_here[i]) // Zatiaľ najlepšia možnosť...
                if (abs(coords[i].first - coords[j].first) +
                    abs(coords[i].second - coords[j].second) <=
                    times[i] - times[j]) { // ...a vieme ju dosiahnuť z aktuálnej pozície.
                        best_here[i] = best_here[j] + 1;
```

```
                }
            }
            // Ak sme vedeli nejaký počet stihnúť predtým, vieme aj teraz.
            best_anywhere[i] = max(best_anywhere[i - 1], best_here[i]);
        }
        // Na konci nám je jedno, kde skončíme, nemusíme pritom stihnúť posledné sypanie.
        cout << n - best_anywhere[n] << endl;
}
```

Merlin

7. Novodobý farmár

(max. 12 b za popis, 8 b za program)

Keďže sa na vstupe dokopy nachádza $O((n+q)D)$ znakov, tak časová zložitosť načítavania vstupu bude pri všetkých riešeniach $O((n+q)D)$, preto ju už ďalej pri odhade časovej zložitosti nebudeme spomínať.

Ako si to vôbec vieme predstaviť?

Celú úlohy si vieme reprezentovať ako hľadanie najkratšej cesty v nejakom neorientovanom grafe. Vrcholy tohto grafu budú jednotlivé odrody a hrany budú medzi odrodami a a b práve vtedy, ak sa dá zmeniť odroda a za odrodu b zaplatením 1 trojkenu. Hrany v tomto grafe si vieme rozdeliť na 2 skupiny: hrany, ktoré vznikli kvôli barterovým ponukám a hrany, ktoré vznikli kvôli laboratóriu.

Hrany, ktoré vznikli kvôli laboratóriu (označme si ich *stromové*) sa v rôznych vstupoch nelíšia. A čo viac, keď sa pozrieme na graf, ktorý obsahuje len tieto hrany, tak dostaneme binárny strom, ktorý je zakorenený vo vrchole reprezentujúcom prázdny genóm. Ľavý syn nejakého vrchola reprezentuje genóm, ktorý vznikol pridaním `L` na koniec aktuálneho genómu a pravý syn reprezentuje genóm, ktorý vznikol pridaním `R`.

BFS

Najprv si môžeme všimnúť, že sa nikdy neoplatí mať odrodu, ktorej genóm má dĺžku väčšiu ako D. Teda nám stačí vygenerovať graf obsahujúci všetkých $O(2^D)$ vrcholov. Keďže je najdlhšia dĺžka reťazca v prvej podúlohe pomerne malá, tak nám stačí si ho celý zapamätať a pre každú otázku pomocou prehľadávania do šírky nájsť najkratšiu cestu.

Náš graf má $O(2^D)$ vrcholov a dokopy $O(2^D+n)$ hrán, teda celková časová zložitosť pre jednu otázku bude $O(2^D+n)$.

Časová zložitosť predpočitania bude $O(nD\log n)$ alebo $O(2^D)$ v závislosti od implementácie.

Zbytočné a zaujímavé vrcholy

Ak by sa v úlohe nenachádzali žiadne barterové ponuky, tak cesta medzi ľubovoľnými vrcholmi a a b by vyzerala pomerne jednoducho. Najprv pôjdeme z vrcholu a smerom hore do ich najnižšieho spoločného predka[1] (alebo aj v skratke LCA) a potom z tohto predka smerom dole do vrcholu b.

Ako bude vyzerať nejaká cesta z vrcholu a do vrcholu b v grafe, kde sú aj hrany reprezentujúce barterové ponuky? Povedzme, že postupne použijeme barterové hrany $b_1, b_2, ..., b_k$. Potom medzi koncom jednej a začiatkom ďalšej hrany bude cesta prechádzať len cez stromové hrany, teda pôjde hore do ich najnižšieho spoločného predka a potom dole.

Označme si teda vrcholy, ktoré sú konce nejakej barterovej hrany alebo sú súčasťou nejakej otázky ako zaujímavé. Ako sme si vyššie ukázali, tak nám stačí medzi každou dvojicou zaujímavých vrcholov vedieť zistiť najkratšiu cestu, ktorá prechádza len cez stromové hrany. Táto cesta pôjde cez ich najmenšieho spoločného predka. Keďže najnižší spoločný predok dvoch zaujímavých vrcholov sa nachádza niekde na ceste z jedného vrchola do koreňa, tak optimálna cesta medzi zaujímavými vrcholmi bude určite prechádzať len cez vrcholy, ktoré sú predkami aspoň jedného z nich.

Tu sa nám črtá trošku lepšie riešenie. Budeme sa pozerať na graf, ktorý obsahuje len zaujímavé vrcholy a vrcholy, ktoré sú predkami nejakého zaujímavého vrchola. Koľko ich bude? Každý vrchol je v hĺbke najviac D, teda má nad sebou najviac D vrcholov. Takýto graf teda bude mať $O((n+q)D)$ vrcholov a $O((n+q)D)$ hrán. Ak v ňom budeme zas hľadať cesty pomocou prehľadávania do šírky, tak na jednu otázku vieme odpovedať v čase $O((n+q)D)$ s predpočítaním v čase $O(nD)$.

[1] https://www.ksp.sk/kucharka/lca/

Ešte ich vieme nejaké odstrániť

Povedzme, že si medzi zaujímavé vrcholy pridáme LCA všetkých dvojíc na začiatku zaujímavých vrcholov. Keďže najkratšie cesty po stromových hranách idú len hore po LCA, tak pre každý zaujímavý vrchol nám stačí len pridať hranu medzi ním a prvým zaujímavým vrcholom, ktorý je nad ním. Takto sa vieme dostať z jedného zaujímavého vrcholu do druhého po stromových hranách rovnakou cestou, ako predtým, teda sme týmto nič nepokazili.

Koľko vrcholov ale musíme pridať? Predstavme si, že máme na začiatku v poli k zaujímavých vrcholov. Zoberme si teraz 2 vrcholy a a b, ktorých LCA (označme si ho L) sa nachádza najhlbšie v strome. V podstrome vrcholu L sa nemôže nachádzať žiaden iný vrchol (okrem a a $b) z nášho poľa, lebo by to bolo v spore s tým, že L je najhlbšie LCA vrcholov z poľa. Preto pre ľubovoľný iný vrchol c bude platiť, že $LCA(c, a) = LCA(c, b) = LCA(c, L)$. Preto môžeme a a b z poľa vymazať a vložiť tam L. V každom takomto kroku sa zníži počet prvkov v poli o 1, teda nových zaujímavých vrcholov môžeme pridať najviac $O(k)$. V asymptotickej časovej zložitosti sa nám teda nič nezhorší tým, keď tieto vrcholy pridáme medzi zaujímavé!

Vieme teda, že zaujímavých vrcholov bude spolu najviac $O(n + q)$ a hrán medzi nimi tiež $O(n + q)$. Keď sa nám už nejak podarí vytvoriť tento graf, tak pre každú otázku len stačí pomocou Dijkstry (lebo hrany teraz nemajú všetky dĺžku 1) zistiť vzdialenosť konca a začiatku v čase $O((n + q) \log (n + q)$.

Predpočítanie

Vrcholy budeme vkladať do písmenkového stromu (trie). Začneme so všetkými genómami v koreni. Keď sme v nejakom vrchole trie s nejakou množinou genómov, tak si ju vieme rozdeliť v lineárnom čase od ich počtu na tie, ktoré končia v tomto vrchole, ktoré idú do ľavého podstromu a ktoré do pravého. Kedy je vrchol trie zaujímavý? Ak sa vo vrchole končí nejaký genóm, tak určite je zaujímavý, lebo je to koniec nejakej hrany alebo súvisí z nejakou otázkou. Ak nejaké vrcholy patria do jeho ľavého aj pravého podstromu, tak bude tiež zaujímavý, keďže bude LCA nejakej dvojice vrcholov. Inak je nezaujímavý. Keď si takto rekurzívne spočítame všetky zaujímavé vrcholy, tak sa dá aj jednoducho zistiť, ako vyzerajú hrany medzi nimi. Vždy si len stačí do rekurzívneho volania posielať aj rodiča aktuálneho vrchola a jeho hĺbku.

Toto celé vieme predpočítať v čase $O((n + q)D)$, lebo v každej z D vrstiev trie sa nachádza v súčte najviac $n + q$ prvkov a v každej úrovni vykonáme prácu priamo úmernú počtu prvkov v nej.

Rýchlejšie otázky

V predošlom riešení sme si medzi zaujímavé vrcholy pridali všetkých $O(q)$ vrcholov, ktoré sa vzťahujú na otázky. Každý z nich ale využijeme len v tej jednej otázke, ku ktorej sa vzťahujú a inak sú zbytočné. Skúsme teda vrcholy z_i a k_i pridať medzi zaujímavé len počas danej otázky a potom ich odstrániť.

Ako teraz zistiť odpovede na jednotlivé otázky? Keď budeme hľadať nejaký genóm x v našej trie zaujímavých vrcholov, tak niekedy narazíme na hranu, na ktorej sa prestane v nejakom znaku zhodovať (ak taká neexistuje, tak to znamená, že x je v trii a netreba riešiť žiadne špeciálne prípady). Nech táto hrana ide z vrcholu a do vrcholu b. Teraz nám stačí pridať hrany z x do a a z x do b so správnou dĺžkou a máme graf, ktorý správne reprezentuje aktuálnu situáciu. Keď toto spravíme pre z_i a k_i, tak len znova stačí pomocou Dijkstry zistiť ich vzdialenosť.

Keďže graf zaujímavých vrcholov teraz bude mať len $O(n)$ vrcholov, tak odpovedať na otázku vieme v čase $O(n \log n)$. Predpočítanie sa nám nijak nezhoršilo. Finálna časová zložitosť je teda $O(nD)$ na predpočítanie a $O(n \log n)$ na otázku.

Listing programu (C++)

```cpp
#include <bits/stdc++.h>
using namespace std;
#define f first
#define s second

vector<string> zaujimave;           // zoznam povodnych zaujimavych genomov
vector<vector<pair<int, int>>> sus; // zoznam susedov vrchola v tvare dvojica {id, dlzka}
                                    // nove vrcholy dostanu id postupne pocas behu programu

struct node {
    node *L, *R;
```

```
12      string L_edge = "L", R_edge = "R";
13      int id;
14      node(int ID = -1) {
15          L = R = nullptr;
16          id = ID;
17      }
18  };
19
20  // Funkcia, ktora spocita vzdialenost genomov A a B
21  // |A| + |B| - 2 LCP(A,B)
22  int tree_dist(string& a, string& b) {
23      int ans = a.size() + b.size();
24      int i = 0;
25      while (i < a.size() && i < b.size() && a[i] == b[i++])
26          ans -= 2;
27      return ans;
28  }
29
30  // Bucket sort
31  // Roztriedi vrcholy z pola cur podla toho, ci maju na dalsom mieste 'L', 'R' alebo uz koncia
32  void b_sort(vector<int>& cur, int d, vector<int>& L, vector<int>& R, vector<int>& koniec) {
33      for (int i : cur) {
34          if (zaujimave[i].size() <= d)
35              koniec.push_back(i);
36          else if (zaujimave[i][d] == 'L')
37              L.push_back(i);
38          else
39              R.push_back(i);
40      }
41  }
42
43  // Zisti, ktore treba pridat navyse a vytvory z nich ohodnoteny graf
44  // cur su indexy aktualnych zaujimavych genomov, ktore sa nachadzaju v podstrome aktualneho vrchola
45  // edge je postupnost L/R, ktore su na hrane z aktualneho vrchola do jeho otca
46  // d je aktualna hlbka
47  // par je index otca
48  // d_par je hlbka otca
49  node* generate_graph(vector<int>& cur, string& edge, int d, int par, int d_par) {
50      // ak sa v podstrome aktualneho vrchola nenachadzaju ziadne zaujimave genomy,
51      // tak tento vrchol nepotrebujeme
52      if (cur.size() == 0) {
53          edge = "";
54          return nullptr;
55      }
56
57      node* ret = new node();
58
59      // Budeme prechadzat po strome az kym nenarazime na zaujimavy vrchol
60      while (1) {
61          vector<int> L, R, koniec;
62
63          // Rozdelime si zaujimave vrcholy podla toho, do ktoreho podstromu patria
64          b_sort(cur, d, L, R, koniec);
```

```
65
66          // Pridame nulove hrany medzi rovnake koncove vrcholy
67          for (int i = 1; i < koniec.size(); i++) {
68              sus[koniec[i - 1]].push_back({koniec[i], 0});
69              sus[koniec[i]].push_back({koniec[i - 1], 0});
70          }
71
72          // Ak sa niektory zo zaujimavych vrcholov konci v tomto vrchole alebo je tento
73          // vrchol LCA nejakej dvojice, tak je zaujimavy
74          if ((L.size() && R.size())  koniec.size()) {
75              // spocitame si index aktualneho vrchola
76              int moj_ind = koniec.size() ? koniec[0] : sus.size();
77              ret->id = moj_ind;
78
79              // Ak pridavame novy vrchol
80              if (koniec.size() == 0)
81                  sus.push_back({});
82
83              // Pridame hranu medzi aktualnym vrcholom a otcom s dlzkou d - d_par
84              sus[moj_ind].push_back({par, d - d_par});
85              sus[par].push_back({moj_ind, d - d_par});
86
87              // Rekurzivne vytvorime pravy a lavy podstrom aktualneho vrchola
88              ret->L = generate_graph(L, ret->L_edge, d + 1, moj_ind, d);
89              ret->R = generate_graph(R, ret->R_edge, d + 1, moj_ind, d);
90              return ret;
91          }
92
93          // Aktualny vrchol nie je zaujimavy, teda treba zvysit hlbku a pridat na koniec hrany do
94          // otca spravny znak
95          if (L.size())
96              edge += 'L';
97          else
98              edge += 'R';
99          d++;
100     }
101 }
102
103 // Funkcia, ktora najde hranu, na ktorej sa genom s oddeluje z trie. Vracia
104 // dvojicu {{id vrchneho, vzdialenost k vrchnemu}, {id spodneho, vzdialenost k spodnemu}}
105 pair<pair<int, int>, pair<int, int>> find(string& s, node* cur, int d = 0) {
106     string match;
107     node* next;
108
109     // Zistime, do ktoreho podstromu patri genom s
110     if (s[d] == 'L') {
111         next = cur->L;
112         match = cur->L_edge;
113     } else {
114         next = cur->R;
115         match = cur->R_edge;
116     }
117
```

```
118     // Postupne skontrolujeme znaky na hrane do tohto podstromu
119     for (int i = 0; i < match.size(); i++) {
120         // Genom s moze skoncit niekde v strede tejto hrany
121         if (s.size() <= d + i) {
122             // Vzdialenost k vrchnemu vrcholu je pocet prejdenych znakov (i)
123             // Vzdialenost k spodnemu vrcholu je dlzka hrany - pocet prejdenych znakov
124             return {{cur->id, i}, {next->id, match.size() - i}};
125         }
126         // Genom sa moze prestat zhodovat niekde v strede hrany
127         if (match[i] != s[d + i]) {
128             // Vzdialenost od konca genomu s k vrcholu, v ktorom sa prestane zhodovat s triou
129             int dist = s.size() - d - i;
130             // Vzdialenosti k vrchnemu a spodnemu vrcholu spocitame rovnako, len pridame
131             // vzdialenost k vrcholu, kde sa s oddeluje od trie
132             return {{cur->id, dist + i}, {next->id, dist + match.size() - i}};
133         }
134     }
135 
136     // Neexistuje dalsi vrchol, genom s sa teda prestal zhodovat s triou v nejakom liste
137     if (next == nullptr)
138         return {{cur->id, s.size() - d - match.size()}, {0, 1 << 30}};
139 
140     // Rekurzivne sa zavolame na dalsi vrchol
141     return find(s, next, d + match.size());
142 }
143 
144 int main() {
145     cin.tie(0)->sync_with_stdio(0);
146     int n, q;
147     cin >> n >> q;
148     sus.resize(2 * n);
149     for (int i = 0; i < n; i++) {
150         string a, b;
151         cin >> a >> b;
152         zaujimave.push_back(a);
153         zaujimave.push_back(b);
154 
155         // Pridame barterove hrany
156         sus[2 * i].push_back({2 * i + 1, 1});
157         sus[2 * i + 1].push_back({2 * i, 1});
158     }
159 
160     vector<int> cur(2 * n);
161     // iota naplni vektor cur postupne cislami 0, 1, 2, ...
162     iota(cur.begin(), cur.end(), 0);
163 
164     // Vytvorime si koren trie
165     node* tr = new node();
166 
167     vector<int> R, L, koniec;
168     b_sort(cur, 0, L, R, koniec);
169     tr->id = sus.size();
170     sus.push_back({});
```

```
171
172     // Vytvorime lavy a pravy podstrom korena
173     tr->L = generate_graph(L, tr->L_edge, 1, tr->id, 0);
174     tr->R = generate_graph(R, tr->R_edge, 1, tr->id, 0);
175
176     // Pre kazdu otazku spustime Dijkstru na vytvorenom grafe
177     for (int i = 0; i < q; i++) {
178         string a, b;
179         cin >> a >> b;
180         // Najdeme hranu, kde hrany, kde sa a a b v trii prestavaju zhodovat
181         auto a_edge = find(a, tr), b_edge = find(b, tr);
182
183         // Pole vzdialenosti, -1 znamena, ze sme este vrchol nepreskumali
184         vector<int> dist(sus.size(), -1);
185
186         priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> que;
187
188         // Do prioritnej fronty pridame horny aj spodny vrchol hrany, v ktorom
189         // sa vrchol a prestava zhodovat
190         que.push({a_edge.f.s, a_edge.f.f});
191         que.push({a_edge.s.s, a_edge.s.f});
192
193         // Obycajny Dijkstrov algoritmus
194         while (!que.empty()) {
195             int d, a;
196             tie(d, a) = que.top();
197             que.pop();
198             if (dist[a] != -1)
199                 continue;
200             dist[a] = d;
201             for (auto u : sus[a]) {
202                 if (dist[u.f] == -1)
203                     que.push({u.s + d, u.f});
204             }
205         }
206
207         // Odpoved je minimum zo vzdialenosti a, b a najkratsej cestu v strome
208         int ans = tree_dist(a, b);
209
210         // Skusime najprv horny vrchol hrany, kde sa odpaja b a potom spodny vrchol tejto hrany
211         ans = min(ans, dist[b_edge.f.f] + b_edge.f.s);
212         ans = min(ans, dist[b_edge.s.f] + b_edge.s.s);
213
214         cout << ans << "\n";
215     }
216 }
```

Ešte rýchlejšie otázky

K zrýchleniu otázok nám stačí jednoduché pozorovanie. Keďže náš strom genómov má hĺbku najviac D, tak maximálna možná vzdialenosť dvoch vrcholov bude $2D$. Pri Dijkstre teda nemusíme používať prioritnú frontu, stačí nám pole obyčajných front veľkosti $2D$. Vrcholy budeme vyberať zo začiatku prvej neprázdnej fronty a keď vkladáme nový vrchol vzdialený od počiatočného i, tak ho vložíme do i-tej fronty. Takýmto spôsobom prejdeme vrcholy od najbližších po najvzdialenejšie (teda rovnako, ako pri použití prioritnej fronty). Navyše vkladať aj

vyberať[2] z tejto dátovej štruktúry vieme v konštantnom čase.

Pomocou tejto optimalizácie vieme odpovedať na otázky v čase $O(n)$ a časová zložitosť predpočítania sa nijak nezhoršila. Výsledná časová zložitosť je teda $O((n+q)D+qn)$.

8. Yucatán na polia!

paulinia
(max. 12 b za popis, 8 b za program)

Skúšame všetky možnosti

Možnosť ktorú (skoro) vždy ide skúsiť, je vyskúšať všetky možnosti. Všimnime si, že pre pole s n políčkami, po každom zožatí existuje 2^n možností ako pole práve vyzerá – každé políčko môže byť celé zožaté alebo celé nezožaté. Keďže každé žatie by malo zožať aspoň jedno políčko, počet nezožatých políčok sa nám postupne zmenšuje.

Na tejto myšlienke vieme podstaviť riešenie pomocou *bitmaskovej dynamiky*. Ak ste o tomto princípe ešte nepočuli, jedná sa v princípe o reprezentovanie stavov – polí boolov, ako celé čísla. V našom prípade bity s hodnotou 1 reprezentujú políčka, ktoré sú ešte nezožaté. Takže stavy sú reprezentované celými číslami od 0 po 2^n-1.

Taktiež si všimnime, že ak $a<b$ sú možné stavy, potom sa nikde nevieme dostať z a po b – žatím sa reprezentácia poľa nikdy nezvýši.

Na tomto princípe teraz postavíme dynamiku: postupne si prejdeme všetky stavy a pre každý zrátame, koľko šťastia vieme bohom zabezpečiť od dosiahnutia tohto stavu až po zožatie celého poľa. Stavy prejdeme v poradí od 0 po 2^n-1. Toto nám zaručí, že keď spracúvame nejaký stav, všetky stavy z neho dosiahnuteľné už budú spracované. Pre každý stav si pozrieme všetky možnosti ktoré vieme zožať. Naoko ich je až $O(n^2)$, avšak môžete si všimnúť, že sa vždy oplatí žať *maximálny* úsek kukurice - teda taký úsek, ktorý nevieme rozšíriť na žiadnu stranu, tak aby sme zožali viac (rovnakého) kusu kukurice.

Oplatí sa iba maximálny úsek

Toto tvrdenie si vieme zdôvodniť nasledovne. Bez ujmy na všeobecnosti sa môžeme tváriť, že žiadne už zožaté políčka ešte neexistujú. Predstavte si, že na začiatku zožneme úsek kukurice dlhý a, hoci by sme ho mohli rozšíriť (bez ujmy na všeobecnosti doprava) o ďalšie políčko (povedzme na pozícii i). Bez ujmy na všeobecnosti sa všetky žatia môžu začínať aj končiť nezožatým políčkom.Takto sa teda žiadna žatva pred zožatím i-teho políčka nemôže spoliehať na to, že a políčok priamo naľavo od i-teho (pôvodne s rovnakou odrodou ako i-te) je už zožatých. Teda vieme ich žatvu posunúť, a zožať ich spolu s i-tym políčkom. Keďže $(a+b)^2>a^2+b^2$ pre $a,b>0$ takáto zmena vždy zvýši šťastie.

Ozbrojení s týmto pozorovaním, si teda môžme všimnúť, že existuje $O(n)$ možných úsekov ktoré sa oplatí skúsiť: prejdeme si pole a tak ho rozdelíme na súvislé úseky, tak že každý úsek obsahuje len jeden typ kukurice, a každé dva vedľajšie úseky majú rôzne typy kukurice. Keďže každé políčko je v najviac jednom úseku, úsekov bude najviac n, a žiadny z nich sa nedá rozšíriť o viac nezožatých políčok kukurice. Takže nám pre každý možný stav poľa stačí skúsiť najviac n možností na ďalšiu žatvu. Toto nám teda dá riešenie s časovou zložitosťou $O(n2^n)$, a s pamäťovou zložitosťou $O(2^n)$ ktoré stačí na 2 body.

Na záver si všimnime, že na zrekonštruovanie riešenia, si nám stačí pamätať pre každý stav aký ďalší úsek máme zožať, takže toto nám nezničí pamäťovú zložitosť. Riešenie vieme z toho zrekonštruovať v lineárnom čase.

Listing programu (C++)

```cpp
#include <bits/stdc++.h>

using namespace std;

#define FOR(i, n) for (int i = 0; i < (int)n; i++)
#define ii pair<int, int>
#define MP make_pair
#define SIZE(x) int(x.size())
```

[2]Keďže sú dĺžky hrán kladné, tak sa index prvej neprázdnej fronty bude len zvyšovať. Keď si budeme pamätať ukazovateľ na prvú neprázdnu frontu a postupne ho budeme posúvať doprava, tak ho posunieme $O(D)$ krát, teda vyberanie z tejto dátovej štruktúry bude v amortizovanom konštantnom čase.

```
9
10  ii next(int a, int i, vector<ii>& starts) {
11      if (!(a & (1 << i)))
12          return {a, 0};
13      int j = i;
14      int total = 0;
15      int b = a;
16      while (j < starts.size() - 1 && (starts[j].second == starts[i].second  (!(a & (1 << j))))) {
17          if (a & (1 << j)) {
18              total += starts[j + 1].first - starts[j].first;
19              b ^= (1 << j);
20          }
21          j++;
22      }
23      return {b, total};
24  }
25
26  int main() {
27      cin.tie(0)->sync_with_stdio(0);
28      int t;
29      cin >> t;
30      while (t--) {
31          int n;
32          cin >> n;
33          vector<int> pole;
34          vector<ii> starts;
35          FOR(i, n) {
36              int a;
37              cin >> a;
38              pole.push_back(a);
39              if (i) {
40                  if (pole[i] != pole[i - 1]) {
41                      starts.push_back({i, pole[i]});
42                  }
43              } else {
44                  starts.push_back({i, pole[i]});
45              }
46          }
47
48          int m = starts.size();
49          starts.push_back({n, -1});
50
51          vector<ii> dp(1 << m, {0, -1});
52
53          FOR(a, 1 << m) {
54              if (!a)
55                  continue;
56              FOR(i, m) {
57                  if (a & (1 << i)) {
58                      ii prel = next(a, i, starts);
59                      dp[a] = max(dp[a], MP(dp[prel.first].first + prel.second * prel.second, i));
60                  }
61              }
```

```
        }

        vector<ii> instructions;
        int a = (1 << m) - 1;
        while (a) {
            int zac = dp[a].second;
            int end = zac;
            while (end < SIZE(starts) &&
                   (starts[zac].second == starts[end].second  (!(a & (1 << end))))) {
                if (a & (1 << end))
                    a ^= (1 << end);
                end++;
            }
            instructions.push_back({starts[zac].first + 1, (end < m ? starts[end].first : n)});
        }

        cout << dp[(1 << m) - 1].first << " " << SIZE(instructions) << "\n";

        for (ii a : instructions) {
            cout << a.first << " " << a.second << "\n";
        }
    }
}
```

(Skoro) vzorák

Poďme sa zamyslieť nad nejakým polynomiálnym riešením. Ako pri hrubej sile, aj teraz sa bude jednať o dynamické programovanie. Problém na ktorý narazíme, ak sa pokúsime priamočiaro zlepšiť riešenie hrubou silou, je že ak zožneme nejaký úsek kukurice, riešenie sa nám *nerozpadne* na viacero nezávislých podproblémov - aj keď sú kusy poľa rozdelené už zožatým úsekom, vieme stále žať cez tento prázdny úsek. A pretože za jedno žatie získame *štvorec* šťastia, treba na to ísť múdrejšie. Totiž kontribúcie z oboch strán sú kombinované netriviálnym spôsobom – teda ak získame a z jednej, a b z druhej strany prázdneho úseku v jednom žatí, šťastie z toho žatia je $(a+b)^2 = a^2 + b^2 + 2ab$.

Pozrime sa na prvé políčko kukurice. Keďže chceme dožať celé pole, niekedy v optimálnom riešení zožneme prvé políčko. Buď s ním nezožneme žiadne ďalšie políčko, alebo povedzme že v tom žatí zožneme na prvý raz aj nejaké ďalšie políčka. Povedzme že i-te je najľavejšie z nich. V tom prípade musíme pred týmto žatím zožať všetky políčka na pozíciach $2, \ldots, i-1$ (ak i je druhé políčko, potom netreba predtým zožať nič). Navyše tento interval políčok musíme zožať iba žatiami vnútri intervalu (inak by sme zožali políčko 1 alebo i skôr ako chceme).

Takto nám teda vlastne vznikne *nezávislý podproblém*: Daný je interval $[z, k]$, aké najväčšie šťastie vieme dosiahnuť ak žneme len kukurice na tomto intervale?

Avšak, toto nie je jediný problém, ktorý nám ostane. Zostane nám ešte otázka, ako maximalizovať šťastie z intervalu $[i, n]$, ak vieme že nám naľavo "odstáva" jedno nezožaté políčko kukurice rovnakého typu ako na políčku i (keďže najskôr zožneme interval $[2, i-1]$, nezáleží nám kde presne sa to "odstávajúce políčko" nachádza (stačí nám vedieť že je naľavo od začiatku úseku).

Tak by sa mohlo zdať, že nám stačí vedieť vyriešiť dva podproblémy:

- Pre daný podinterval poľa, aké najväčšie šťastie vedia dosiahnuť žatím len na tomto úseku?
- Pre daný podinterval poľa, aké najväčšie šťastie vedia dosiahnuť žatím len tohto úseku ak je naľavo priamo dosiahnuteľné políčko s rovnakým typom kukurice ako najľavejšie políčko v úseku?

Avšak keď skúsime rekurzívne riešiť druhý problém, zistíme že sa nám kumulujú kukurice rovnakého typu: povedzme že riešime podproblém s úsekom $[z, k]$ a jednou "odstrávajúcou" kukuricou. Potom ak chceme najskôr zožať úsek $[z+1, i]$, tak nám v zostávajúcom úseku $[i+1, k]$ zústávajú 2 odstávajúce nezožaté kukurice naľavo! Chceme teda vyriešiť trochu viac všeobecný podproblém:

- Pre daný podinterval poľa, aké najväčšie šťastie vedia dosiahnuť žatím len tohto úseku ak je naľavo priamo dosiahnuteľných s políčok s rovnakým typom kukurice ako najľavejšie políčko v úseku?

Môžeme si všimnúť, že ak $s = 0$, potom toto zodpovedá prvému podproblému. Takže podproblém ktorý riešime je nasledovný:

Pre daný interval [z, k] a číslo $s \geq 0$, aké najväčšie šťastie vedia Mayovia dosiahnuť zožatím intervalu $[z, k]$ ak majú navyše s kukuríc rovnakého typu ako kukurica na políčku z dostupných na zožatie naľavo od úseku.

Nazvime riešenie tohto problému $dp[z][k][s]$. Potom si všimnime, že ak vieme riešenie pre všetky menšie intervaly, vieme spočítať túto hodnotu ako maximum z:

- $dp[z+1][k][0] + (s+1)^2$ – toto je prípad, že keď zožneme interval s z-tým políčkom, práve z bude posledné zožaté políčko
- Pre každé políčko i pre ktoré platí $z < i \leq k$ a na ktorom rastie rovnaká odroda kukurice ako na políčku z, $dp[z+1][i-1][0] + dp[i][k][s+1]$

Samozrejme, pre prázdne intervaly, teda kde $z = k$, riešenie je s^2 (zožneme zostávajúce políčka). Všimnite si, že všetky zaujímavé hodnoty s sú medzi 0 a n. Takto teda dostaneme teda dynamické programovanie s $O(n^3)$ stavmi - to bude pamäťová zložitosť, a keďže potrebujeme $O(n)$ operácií na spočítanie jednej hodnoty, časová zložitosť riešenia je $O(n^4)$.

Aj na konštantách záleží: nerobme prácu navyše

A toto je optimálna časová zložitosť - lenže len asymptoticky. Čo to znamená? Väčšinou, keď riešime časové zložitosti sa nám všetky konštanty skryjú pod O – teda či program spraví $100n^4$ alebo $n^4/10$ operácií, stále má časovú zložitosť $O(n^4)$. Lenže v druhom prípade bude bežať o dosť rýchlejšie, a pre vstupy na hranici časového limitu nám na tom bude veru záležať.

$O(n^4)$ dynamika ktorú sme opísali hore má dosť zlú konštantu: ráta veľa vecí ktoré nemusí. Pri riešení hrubou silou sme si uvedomili, že ak vieme žací interval rozšíriť (o ďalšie políčka rovnakého kukuričného typu), vždy sa to oplatí. Takže napríklad, kedykoľvek keď máme viac kukuríc rovnakého typu za sebou v poli, vieme, že ich vždy budeme kosiť spolu. Takouto myšlienkou vieme spraviť prvú optimalizáciu riešenia: skomprimujme si pole tak, že si budeme pamätať dĺžku úsekov s rovnakou kukuricou a o aký typ sa jedná.

Napríklad z poľa $1, 1, 3, 3, 1$ by nám vzniklo pole $(1, 2), (3, 2), (1, 1)$. Na takomto skomprimovanom poli vieme robiť dynamiku ako hore, len namiesto pripočítania jedného políčka, budeme pripočítať dĺžku úseku na ktorom sme. Takáto úprava síce v najhoršom prípade[3] riešenie nezlepší, ale na väčšine vstupov takáto základná optimalizácia zlepší čas behu niekoľkonásobne.

Na ďalšie pozorovanie si najskôr upravme notáciu. Pre i-ty typ kukurice majme c_i – počet políčok na ktorých sa pestuje. Všimnime si, že teda existuje najviac c_i možných začiatkov úsekov s touto farbou. Ak by j-te políčko typu i bolo aj začiatok úseku, potom sa oplatí uvažovať iba možné s (počet odstávajúcich nezožatých naľavo od úseku) medzi 0 a $j-1$. Taktiež existuje nanajvýš $c_i - j + 1$ možných úsekov ktorými sa oplatí zaoberať. Keďže j-te políčko s odrodou i musí byť na aspoň j-tej pozícii v poli ako takom[4], takže počet možných koncov úsekov je najviac $n - j + 2$. V najhorošom prípade (keďže robíme horný odhad), je každé políčko tejto farby začiatok úseku, teda dostaneme, že v dynamike potrebujeme urobiť najviac

$$\sum_{i}^{n} \sum_{j=1}^{c_i} j(c_i - j + 1)(n - j + 2) \leq \sum_{i}^{n} \frac{n(c_i + 2)^3}{6}$$

iterácií[5] (na spočítanie konečného výsledku – tým myslíme počet iterácií, ktoré naše forcykly v dynamike musia vykonať, resp. v rekurzívnej implementácii koľko volaní musíme spraviť). Všimnime si, dokopy je súčet všetkých c_i rovný n. Akonáhle neexistuje typ kukurice pestovaný na nadpolovičnej väčšine políčok, vieme si ukázať, že v tejto situácii je suma maximalizovaná ak máme len dve odrody kukurice, a každá zaberá polovicu políčok. Vtedy dostaneme horný odhad približne $n^4/24$.

Čo ak je nejakej odrody priveľa? Bez ujmy na všeobecnosti nech je kukurice typu 1 najviac. Vtedy si všimnite, že akonáhle $c_1 > n/2$ (nejaký typ kukurice pestujeme na nadpolovičnej väčšine políčok), potom spravíme operácií vlastne o dosť menej. Keďže úseky rovnakého typu kukurice musia byť oddelené iným typom kukurice, tak celkovo úsekov nebude dokopy viac ako $2(n - c_1) + 1$. Teda tento najväčší typ kukurice nemôže zaberať viac ako $n - c_1 + 1$ úsekov. Spravme si druhý horný odhad práce, ktorú musíme v tomto prípade urobiť. Existuje najviac $2(n - c_1) + 1$ možných koncov (keďže sa neoplatí počítať dynamiku pre konce, ktoré nie sú konce úsekov). Teda, keby bolo j-te políčko s kukuricou typu i začiatkom úseku, na spočítanie dynamík s týmto

[3] aký je najhorší prípad? Zamyslite sa

[4] tu poznamenám, že v praxi toto bude viac, keďže medzi jednotlivými úsekmi rovnakej odrody bude rásť aspoň jedno políčko s inou odrodou

[5] úpravu za nás spraví WolframAlpha

začiatkom by sme potrebovali najviac $j(c_1 - j + 1) \cdot (2(n - c_1) + 1)$ iterácií (tu sme zrecyklovali predchádzajúci argument). Všimnime si, že toto číslo je maximalizované pre $j = (c_1 + 1)/2$. Keďže nebudeme mať viac ako $n - c_1 + 1$ začiatkov ktoré nás zaujímajú, dostaneme že pre túto odrodu budeme musieť spraviť najviac

$$\frac{(n - c_1 + 1)(2(n - c_1) + 1)(c_i + 1)^2}{4} \approx \frac{(n - c_1)^2 c_i^2}{2}$$

A teda celkovo, berúc do úvahy aj ostatné kukurice, pomocou prvého odhadu dostaneme, že nespravíme viac ako približne $n(n - c_1)^3/6 + (n - c_1)^2 c_1^2/2$ iterácií. Môžeme si všimnúť[6], že na intervale $[n/2, n]$ táto funkcia klesá, a teda vieme počet iterácií zas odhadnúť pomocou $c_i = n/2$. Vyjde nám tak, že nám stačí približne $n^4(1/48 + 1/32) = 5n^4/96 \leq n^4/19$, čo je len o niečo horšie ako predchádzajúci odhad[7]

Pre porovnanie, pre maximálne n v tejto úlohe, tento odhad vyjde na niečo pod 10^8 čo je zhruba na hrane toho, čo testovať ešte zvládne. Ak vám riešenie stále neprechádza, treba sa zamyslieť nad ďalšími faktormi – akú dátovú štruktúru používate na dynamiku? Robíte dynamiku alebo rekurziu s memoizáciou (hoci tá druhá naozaj navštívi len relevantné stavy, je v praxi pomalšia).

Rekonštrukcia riešenia

Vo vzoráku som zatiaľ nespomenula ako rekonštruovať riešenia. Je to hlavne záležitosť správnej implementácie. Ako zvyčajne, keď potrebujeme v dynamike rekonštruovať riešenie (nielen výsledok) pamätáme si pre každý stav nielen maximálne hodnoty šťastia, ale aj ako sme ho vedeli získať. Následne riešenie rekonštruujeme pomocou zásobníka, v čase lineárnom od veľkosti poľa (maximálny počet žatí). Postupujeme od konca, a treba si dobre rozmyslieť v akom poradí dávame žatia do stacku: pamätajme si že ak chceme prvé políčko úseku spojiť s nejakým ktoré nie je tesne napravo, musíme najskôr vyžať úsek medzi nimi.

Poznámka na záver - $O(n^5)$ riešenie

Existujú aj iné dynamiky v polynomiálnom čase. Napríklad existuje $O(n^5)$ riešenie ktoré stačí na 4 body, ale je (podľa autorkinho názoru) o niečo komplikovanejšie než vzorák. Ak sa chcete zamyslieť, uvažujte dynamické programovanie so stavmi: začiatok úseku, koniec úseku, a typ kukurice t, ktorý chceme dosiahnuť na konci, a množstvo x koľko jej chceme dosiahnuť. Výsledok pre stav je maximálne šťastie, ktoré vieme dosiahnuť nejakým žatím po ktorom ostane na danom úseku x políčok s kukuricou typu t. Všimnite si, že síce má toto riešenie stavov len $O(n^3)$, rovnako ako $O(n^4)$ riešenie, avšak, z každého stavu máme až $O(n^2)$ možností kam sa pohnúť ďalej.

Listing programu (C++)

```cpp
#include <bits/stdc++.h>

using namespace std;

#define FOR(i, n) for (int i = 0; i < (int)n; i++)
#define ii pair<int, int>

int main() {
    cin.tie(0)->sync_with_stdio(0);
    int t;
    cin >> t;

    while (t--) {
        int n;
        cin >> n;
        /* kedze sa nam vzdy uplati zacinat/koncit zatia za zaciatkoch/koncoch rovnakych kukuric,
         * pri nacitavani vstupu si vstup rozdelime na suvisle useky.
         * useky[c] je pole kde si pamatame ktore useky patria kukurici typu c.
         * counts[c][i] je pocet kukuric typu c ktore sme videli pred i-tym usekom typu c */
```

[6] odporúčam použiť napríklad Geogebru

[7] s trochou práce navyše to viete znížiť na približne $n^4/20$.

```
        vector<vector<int>> useky(n), counts(n);
        /* starts[i] =
         * (kde zacina i-ty usek, (typ kukurice na nom, na akom indexe ho najdeme v 'useky')) */
        vector<pair<int, ii>> starts;
        vector<int> count(n, 0);
        vector<int> pole;
        int maxc = 0;
        FOR(i, n) {
            int kuk;
            cin >> kuk;
            pole.push_back(--kuk);
            if (i) {
                if (kuk != pole[i - 1]) {
                    counts[kuk].push_back(count[kuk]);
                    useky[kuk].push_back(size(starts));
                    starts.push_back({i, {kuk, size(useky[kuk]) - 1}});
                }
            } else {
                counts[kuk].push_back(count[kuk]);
                useky[kuk].push_back(0);
                starts.push_back({i, {kuk, 0}});
            }
            count[kuk]++;
            maxc = max(maxc, count[kuk]);
        }

        int m = size(starts);
        starts.push_back({n, {-1, -1}});
        /* dp[z][k][p] = (vysledok od z-teho do k-tehu useku
         * ak nalavo je p nezozatych kukuric rovnakeho typu ako z-ty usek,
         * s ktorym najblizsim usekom napravo zozneme z-ty usek)
         */
        vector<vector<vector<ii>>> dp(
            m + 1, vector<vector<ii>>(m + 1, vector<ii>(maxc + 1, {0, -1})));
        int cnt = 0;
        for (int k = 0; k <= m; k++) {
            FOR(i, maxc + 1) dp[k][k][i].first = i * i;
            for (int z = k - 1; z >= 0; z--) {
                int typ = starts[z].second.first; // typ kukurice
                int index_useky = starts[z].second.second;
                FOR(p, counts[typ][index_useky] + 1) {
                    int len = starts[z + 1].first - starts[z].first + p;
                    ii& best = dp[z][k][p];
                    best = {len * len + dp[z + 1][k][0].first, k};
                    for (int i = index_useky + 1; i < size(useky[typ]) && useky[typ][i] < k; i++) {
                        int u = useky[typ][i];
                        best = max(best, {dp[z + 1][u][0].first + dp[u][k][len].first, u});
                        cnt++;
                    }
                }
            }
        }

```

```
73          vector<ii> instructions;
74          stack<pair<ii, ii>> to_resolve;
75          to_resolve.push({{0, m}, {0, 0}});
76          while (to_resolve.size()) {
77              auto top = to_resolve.top();
78              to_resolve.pop();
79              int z = top.first.first;
80              int k = top.first.second;
81              int p = top.second.first;
82              int from = top.second.second;
83              if (z >= k) {
84                  if (p > 0)
85                      instructions.push_back({starts[from].first + 1, starts[z].first});
86                  continue;
87              }
88              int instr = dp[z][k][p].second;
89              if (instr == k) {
90                  instructions.push_back({starts[from].first + 1, starts[z + 1].first});
91                  to_resolve.push({{z + 1, k}, {0, z + 1}});
92              } else {
93                  to_resolve.push({{instr, k}, {p + starts[z + 1].first - starts[z].first, from}});
94                  to_resolve.push({{z + 1, instr}, {0, z + 1}});
95              }
96          }
97
98          cout << dp[0][m][0].first << " " << size(instructions) << "\n";
99          for (ii a : instructions) {
100             cout << a.first << " " << a.second << "\n";
101         }
102     }
103 }
```