Korešpondenčný seminár z programovania
XXXIII. ročník, 2015/16
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. série zimnej časti

Matúš

1. Zúfalo málo miesta

(max. 7 b za popis, 3 b za program)

Základná myšlienka algoritmu pre hľadanie najlepšieho disku je veľmi jednoduchá. Budeme si pamätať najlepší disk, aký sme doteraz našli (na začiatku si zoberieme nejaký fiktívny, naozaj zlý disk), postupne prejdeme všetky disky a každý porovnáme s tým, ktorý bol doteraz najlepší. Vždy, keď je nejaký disk výhodnejší ako ten, čo vyhrával doteraz, stane sa najlepším on.

To, čo ostáva vymyslieť, je, ako porovnať dva disky, bez toho, aby sme spočítali samotnú cenu za gigabajt. Ak by sme mohli používať reálne čísla, cenu za gigabajt by sme porovnávali nasledovne:

$$\frac{cena\_aktualny}{kapacita\_aktualny} \gtreqless \frac{cena\_najlepsi}{kapacita\_najlepsi}$$

Delenie je však operácia, pri ktorej vznikajú desatinné čísla, preto sa chceme tejto operácii vyhnúť. Obe strany môžeme prenásobiť kapacitami diskov. Dostaneme:

$$cena\_aktualny \cdot kapacita\_najlepsi \gtreqless cena\_najlepsi \cdot kapacita\_aktualny$$

Výsledok tohto porovnania bude rovnaký, ako výsledok toho predošlého. Ak v pôvodnej nerovnosti platilo "<", ">" alebo "=", bude rovnaký vzťah platiť aj v druhej nerovnosti. Avšak, pri použití tohto porovnania sa nemusíme obávať vzniku desatinných čísel, lebo nepoužívame delenie.

Aká je zložitosť tohto algoritmu? Keďže na vstupe máme n diskov, ktoré musíme načítať, zložitosť bude aspoň $O(n)$. Pri porovnávaní diskov sa na každý pozrieme iba raz, keď ho porovnávame so zatiaľ najvýhodnejším. Každé porovnanie používa iba násobenie čísel s konštantnou zložitosťou a teda celková časová zložitosť bude $O(n)$. Pamäťová zložitosť je $O(1)$, pretože si stačí pamätať len doteraz najlepší disk. Na ostatné môžeme zabudnúť.

Niektorí z vás si zbytočne sťažovali život hľadaním najmenšieho spoločného násobku kapacít alebo cien diskov. To však algoritmu nepomôže a akurát ho to môže spomaliť, pretože hľadanie $nsn(a, b)$ trvá $O(\log b)$.

Listing programu (C++)

```
#include <cstdio>

int main(){
    int n, kapacita, cena;
    int best_kapacita = 0;      //Inicializujeme najlepsi najdeny disk na nieco,
    int best_cena = 1;          //co je zarucene nevyhodnejsie ako cokolvek ine
    scanf("%d",&n);
    for (int i = 0; i < n; i++){
        scanf("%d %d", &kapacita, &cena);
        if ( cena * best_kapacita <  best_cena * kapacita ){ //nase upravene kriterium
            best_kapacita = kapacita;
            best_cena = cena;
        }
    }
    printf("%d %d\n", best_kapacita, best_cena);
}
```

Listing programu (Pascal)

```
var i, n, best_kapacita, best_cena, kapacita, cena :integer;
begin
  best_cena := 1;
  best_kapacita := 0;
  readln(n);
  for i:=1 to n do begin
    readln(kapacita, cena);
    if ( cena * best_kapacita <  best_cena * kapacita ) then begin
      best_kapacita := kapacita;
      best_cena := cena;
    end;
  end;
  writeln(best_kapacita, ' ', best_cena);
end.
```

Askar

2. Zázračné platenie

(max. 5 b za popis, 5 b za program)

Úloha je pomerne priamočiara – je potrebné zrátať súčet cifier čísla na vstupe. Ako to spraviť? Na to existuje viacero rôznych prístupov a my si ukážeme dva z nich.

Súčet cifier pomocou celočíselného delenia

Všimnime si, že poslednú cifru čísla n vieme získať ako zvyšok po delení 10 (operátor `%` v C++ a Pythone, `mod` v Pascale). Rovnako si všimnime, že celá časť čísla n po delení 10 (operátor `/` v C++, `//` v Python a `div` v Pascale) vyzerá ako pôvodné n, ale bez poslednej cifry. Zopakovaním tohto postupu sa vieme dostať aj ku predposlednej cifre čísla n. A potom k predpredposlednej. A tak ďalej.

Takýmto spôsobom vieme prejsť cez všetky cifry čísla n. Ak ich počas toho budeme aj sčítavať, tak získame náš želaný ciferný súčet – $digits(n)$.

Časová zložitosť tohto prístupu je úmerná počtu cifier čísla n, čo je zhruba hodnota $\log n$, preto dostávame zložitosť $O(\log n)$. Pamäťová zložitosť je konštantná.

Listing programu (C++)

```
#include <iostream>

using namespace std;

long long digits(long long n) {
    long long sum = 0;
    while( n > 0 ) {
        long long last_digit = n % 10;
        sum += last_digit;
        n = n / 10;
    }
    return sum;
}

int main(){
    long long n;
    cin >> n;
    cout << n - digits(n) << endl;
}
```

Súčet cifier pomocou načítania čísla ako reťazca

Trikovejší spôsob, ako vyriešiť úlohu, je prečítať číslo na vstupe ako reťazec – postupnosť znakov. Potom stačí každý znak skonvertovať na číslo a čísla sčítať. V jazyku Python sa toto robí obzvlášť pohodlne.

Listing programu (Python)

```
n = input()
print(int(n) - sum([int(x) for x in n]))
```

Jano

3. Zaujímavé kŕmenie strašidelnej príšery

(max. 6 b za popis, 4 b za program)

Každý pavúk môže stretnúť príšeru najviac v jednom stĺpci a to najviac raz. Vieme pomerne jednoducho spočítať, či pavúk vôbec dokáže stretnúť príšeru a ak áno, tak v ktorom stĺpci ju stretne.

Ak sa nám toto podarí zistiť, odpoveď na úlohu zistíme tak, že najprv vyrobíme pre každý stĺpec počítadlo (na začiatku s hodnotou 0) a následne cyklom prejdeme všetky pavúky. Pre každého pavúka vypočítame stĺpec, v ktorom pavúk stretne príšeru a ak taký stĺpec existuje, zvýšime si počítadlo daného stĺpca. Na konci len vypíšeme počítadlá pre každý stĺpec.

V programe budeme rozlišovať pavúky podľa toho, ktorým smerom sa hýbu. Pavúky, ktoré idú na juh (dolu) príšeru nestretnú nikdy, pretože sú vždy pred ňou. Pavúky, ktoré idú na sever (hore) stretnú príšeru v tom stĺpci, v ktorom sa nachádzajú.

Pavúky, ktoré idú na západ (vľavo), môžu stretnúť príšeru len vtedy, keď sa príšera ocitne v rovnakom riadku, pretože tieto pavúky idú stále po tom istom riadku. To, kedy to bude, vieme ľahko vypočítať: príšera sa do k-teho riadku dostane po k sekundách. Takže pavúk, ktorý ide na západ a začína na pozícii (k_i, s_i) môže stretnúť príšeru iba po k_i sekundách od začiatku pohybu. A vtedy sa pavúk bude nachádzať v stĺpci $s_i - k_i$ (alebo už bude schovaný pod podlahou, ak $s_i - k_i < 1$). Preto tento pavúk buď stretne príšeru v stĺpci $s_i - k_i$, alebo ju nestretne vôbec (ak $s_i - k_i < 1$).

Podobne pavúky, ktoré idu na východ (doprava) stretnú príšeru v stĺpci $s_i + k_i$ alebo vôbec, ak $s_i + k_i > s$.

Časová zložitosť riešenia je $O(p+s)$. Načítanie vstupu a výpočet pre každého pavúka zaberie $O(p)$ času. Výpis výstupu $O(s)$. Pamäťová zložitosť je $O(s)$.

Listing programu (C++)

```
#include<iostream>
#include<algorithm>
#include<vector>
using namespace std;
#define For(i, n) for(int i = 0; i<(n); ++i)

int S[500001];
int pocet,k,s,ki,si;
char c;

int main(){
    // Precitame pocet pavukov, riadkov a stlpcov
    cin >> pocet >> k >> s;
    For(i, pocet) {
        // Pre kazdeho pavuka nacitame jeho poziciu a smer
        cin >> ki >> si >> c;

        // Ak ide na juh, nestretne priseru, v ostatnych pripadoch spocitame,
        // kde ju stretne a zvysime pocitadlo daneho stlpca
        if (c == 'S') S[si]++;
        if (c == 'Z' && si-ki > 0) S[si-ki]++;
        if (c == 'V' && si+ki <= s) S[si+ki]++;
    }
    // Pre kazdy stlpec vypiseme pocitadlo. Pozor na medzeru za poslednym cislom.
    For(i, s) cout << S[i+1] << char((i==s-1)?'\n':' ');
}
```

4. Zašpinení programátori

výskumný tím Katedry Sprchovania a Plávania
(max. 9 b za popis, 6 b za program)

Poďme si princíp vysvetliť na príkade. Majme 7 sprchových gélov, ktoré boli na začiatku na kope v stave `0, 1, 2, 3, 4, 5, 6` a po jednom dni sprchovania v stave `0, 1, 3, 4, 6, 5, 2` (v týchto príkladoch za "spodok" kopy považujeme ľavú stranu riadku).

Najmenší možný počet

Keď sa programátor sprchuje, tak vytiahne svoj sprchový gél z ľubovoľnej pozície v kope a položí ho na vrch kopy. To nič nemení na kope pod ním ani nad ním, okrem najvyššej pozície. Preto, ak sa spodky počiatočnej aj výslednej kopy zhodujú, znamená to, že tieto gély neboli použité – v našom prípade gély `0, 1`. Použitý gél sa prejaví tak, že chýba vo výslednej postupnosti na svojom očakávanom mieste – gél `2` v druhej postupnosti chýba. Gély nad ním (ak nie sú použité) sú ale stále v nezmenenom poradí – gély `3, 4`. Ďalej je pôvodné poradie opäť narušené, lebo bol použitý gél `5`.

Vo výslednej postupnosti na pôvodných miestach teda chýbajú gély `2, 5`, a preto je v tomto príklade odpoveďou 2.

Najväčší možný počet

Tu je odpoveď celkom zrejmá. Každý programátor sa sprchuje najviac raz za deň. Nech výsledná kopa vyzerá akokoľvek, vždy ju vieme vytvoriť tak, že sa osprchujú všetci programátori práve v poradí, v akom sú gély vo výslednej kope. Teda, ak sa chce sprchovať čo najväčší počet programátorov, každý sa bude sprchovať práve raz. Odpoveď je teda n.

Poďme implementovať

Základná myšlienka algoritmu je, že máme 2 ukazovatele, jeden do každého z dvoch polí. Najskôr ukazujú na počiatočnú pozíciu. Máme tiež premennú, kde si pamätáme, koľko použitých šampónov sme našli.

Ak sa čísla šampónov, na ktoré ukazovatele ukazujú, zhodujú, oba sa posunú o jednu pozíciu doprava. Našli sme nepoužitý gél.

Ak sa niekde nezhodujú,[1] zvýšime počet osprchovaných programátorov a preskočíme použitý gél v pôvodnej postupnosti (posunieme ukazovateľ o jedno doprava). Postupnosti sa budú zhodovať, až kým nenájdeme ďalší použitý gél.

Postup opakujeme, kým neprejdeme celú pôvodnú postupnosť. Na konci budeme mať najmenší možný počet osprchovaných programátorov.

[1] Tak hurá! Niekto prekonal svoju averziu k vode a osprchoval sa!

Zložitosti

Z popisu algoritmu je zrejmé, že časová zložitosť bude $O(n)$, lebo každú postupnosť prejdeme nanajvýš raz. Lepšia časová zložitosť sa dosiahnuť nedá, lebo musíme načítať celý vstup. Pamäťová zložitosť bude tiež $O(n)$. Lepšie to opäť nejde, lebo keď postupnosti prechádzame, musíme si aspoň jednu pamätať celú.

Listing programu (C++)

```
#include<cstdio>

#define For(i, n) for(int i = 0; i < n; i++)

int n;
int povodna[500042], vysledna[500042];

int main() {
    scanf("␣%d", &n);
    For(i, n) scanf("␣%d", &povodna[i]);
    For(i, n) scanf("␣%d", &vysledna[i]);

    int a = 0, b = 0, pouzite = 0;

    while(a < n){
        if(povodna[a] == vysledna[b]) {
            a++;
            b++;
        } else {
            a++;
            pouzite++;
        }
    }

    printf("%d\n%d\n", pouzite, n);
    return 0;
}
```

Listing programu (Python)

```
input()                 # n nas nezaujima :P
povodna_kopa = input().split()
vysledna_kopa = input().split()

b = 0
vysledok = 0

for gel in povodna_kopa:
    if gel == vysledna_kopa[b]:
        b += 1          # Kopy sa zhoduju, ideme dalej
    else:
        vysledok += 1   # Niekto sa osprchoval

print('{0}\n{1}'.format(vysledok, len(povodna_kopa)))
```

Peťa

5. O Vlejdovom bicykli

(max. 9 b za popis, 6 b za program)

Na získanie 7-8 bodov za tento príklad stačilo bruteforce riešenie *"vyskúšame všetky trojice"* s časovou zložitosťou $O(n^3)$. (Samozrejme, aj tu si treba dať pozor na to, že ak tri rúrky majú tvoriť trojuholník, musí pre ne platiť, že súčet dĺžok ľubovoľných dvoch rúrok je väčší než dĺžka tretej rúrky).

Riešenie vieme zlepšiť využitím informácie o obvode trojuholníka. Namiesto zmýšľania v štýle *"vyskúšame všetky trojice, či náhodou nebudú mať súčet d,"*, použijeme štýl *"vyskúšame dvojice, dopočítame si dĺžku tretej strany trojuholníka a overíme či máme tyčku správnej dĺžky."* Overenie sa dá spraviť napríklad binárnym vyhľadávaním, čím by sme dosiahli časovú zložitosť $O(n^2 \log n)$. Aby sme vedeli binárne vyhľadávať, musíme na začiatku algoritmu pole utriediť, ale to nie je žiaden problém, pretože zložitosť triedenia je $O(n \log n)$.

Ešte lepšie riešenie vieme dosiahnuť prístupom: *"Vyskúšame všetky možnosti pre najdlhšiu rúrku, jej dĺžku označíme r_i, a pokúsime sa zistiť, či sa medzi paličkami nachádzajú dve so súčtom dĺžok $d - r_i$, pričom obe sú nanajvýš také dlhé ako r_i."*

Prečo nám toto pomôže? Pretože v utriedenom poli čísel vieme nájsť dve také, ktoré majú súčet x, v lineárnom čase. Stačí použiť dvoch bežcov: dva pointre ukazujúce do poľa, na začiatku inicializované na začiatok a koniec poľa. Pokiaľ je súčet prvkov, na ktoré ukazujú, menší ako x, posunieme ľavý pointer doprava. Pokiaľ je súčet prvkov, na ktoré ukazujú, väčší ako x, posunieme pravý pointer doľava. Ak je súčet rovný x, našli sme prvky, ktoré sme hľadali. Nuž a pokiaľ sa stane, že ľavý pointer je napravo od pravého pointra, v poli sa nenachádzajú také dva prvky, ktorých súčet by bol x.

Prehoďme pár slov o tom, prečo to funguje a prečo to má lineárnu časovú zložitosť. Pokúšame sa zistiť, či vo vzostupne utriedenom poli a_0 až a_k, sú dve čísla a_i, a_j také že $a_i + a_j = x$.

Pokiaľ $a_0 + a_k > x$, tak pre všetky $i \geq 0$ platí, že $a_i + a_k > x$ a teda a_k nemôže byť a_j. Preto môžeme a_k zahodiť. Analogicky, keď $a_0 + a_k < x$, môžeme zahodiť a_0. Po zahodení dostaneme o jedna kratšie pole, na ktorom riešime rovnakú úlohu.

Keďže zahadzujeme len tie prvky, ktoré nemôžu byť hľadanými a_i, a_j, tak buď tieto prvky časom nájdeme, alebo zistíme, že sme všetko zahodili a prvky sa v poli nenachádzajú. V každom kroku, v ktorom sme nenašli odpoveď, sme postupnosť skrátili, takže časová zložitosť je $O(k)$, kde k je počiatočná dĺžka postupnosti.

Takže celý algoritmus bude fungovať takto: pole dĺžok rúrok si utriedime, postupne si každú rúrku (jej dĺžku označíme r_i) zvolíme za "najdlhšiu rúrku trojuholníka". Pokiaľ $2r_i \geq d$, rovno vieme, že to, čo by sme poskladali, nebude trojuholník. V opačnom prípade sa pokúsime ku nej nájsť (v časti poľa, v ktorej sú len kratšie rúrky) dve také rúrky, ktoré majú súčet dĺžok rovný $d - r_i$. Ak sa podarilo, vypíšeme `DA SA`, ak sa to nepodarí pre žiadnu "najdlhšiu rúrku trojuholníka", vypíšeme `NEDA SA`. Časová zložitosť algoritmu pre jednu testovaciu sadu je $O(n^2)$. Jeho pamäťová zložitosť je $O(n)$.

Listing programu (C++)

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

using namespace std;

int main() {
    int t, d, n, najdlhsia, a, b;
    long long sucet;
    bool bicykel;
    cin >> t;
    for (int i = 0; i < t; ++i) {
        cin >> n >> d;
        vector<int> r(n);
        for (int j = 0; j < n; ++j) cin >> r[j];
        bicykel = false;
        sort(r.begin(), r.end());
        najdlhsia = 2;
        // Skusame vsetky rurky, ci by mohli byt najdlhsou rurkou trojuholnika
        while (najdlhsia < n && r[najdlhsia]*2 < d && !bicykel) {
            a = 0;
            b = najdlhsia - 1;
            // Hladame dve zvysne rurky pasujuce k vybranej najdlhsej
            while (a < b && !bicykel) {
                sucet = (long long)r[a] + (long long)r[b] + r[najdlhsia];
                if (sucet > d) b--;
                else if (sucet < d) a++;
                else bicykel = true;
            }
            najdlhsia++;
        }
        if (bicykel) cout << "DA SA" << endl;
        else cout << "NEDA SA" << endl;
    }
}
```

Kubo

6. Obchodnícke obtiaže

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

Najskôr sa pozrieme na jednoduchšie algoritmy, ktoré vás mohli pri riešení tejto úlohy napadnúť, následne si predstavíme dynamické programovanie a nakoniec v ňom nájdeme malý trik ku vzorovému riešeniu.

Jednoduché algoritmy

Prvá vec, ktorá nás napadne pri ľubovoľnej úlohe, je brute force. Ten však bude príliš pomalý. Navyše pri tejto úlohe je aj jeho implementácia netriviálna, takže je jednoduchšie sa zamyslieť nad rýchlejším riešením.

Ďalší jednoduchý prístup sú greedy algoritmy. No či už vyberáme pre zákazníka najlacnejšiu alebo najdrahšiu knihu, dá sa nájsť protipríklad, na ktorom nezvolíme optimálnu stratégiu. Poďme sa teda pozrieť na myšlienkovo náročnejšie riešenie, ktoré ale už bude korektné.

Dynamické programovanie

Táto úloha sa dá celkom príjemne riešiť pomocou dynamického programovania – dynamiky. Dynamika je algoritmus, v ktorom zo stavov, ktoré už poznáme, postupne počítame ďalšie, až kým sa nedostaneme ku výsledku.

V tejto úlohe bude naším stavom počet už predaných kníh a vrchné tri knihy na kope. Informácia, ktorú pre tento stav chceme vedieť, je maximum peňazí ktoré vieme mať vtedy zarobených. Medzi týmito stavmi sa presúvame predajom kníh. Pri každom predaji predáme jednu knihu (konkrétneho zákazníka vieme určiť z počtu

predaných kníh) a namiesto nej do trojice známych kníh dáme ďalšiu knihu z kopy. Týmto nám vznikol nový stav. Ak už sme taký stav niekedy dosiahli, tak si porovnáme zarobené peniaze a zapamätáme si maximum.

Implementovať to budeme pomocou štvorrozmerného poľa, kde jedna súradnica bude počet predaných kníh, ďalšími tromi budú vrchné tri knihy na kope, a hodnota jedného políčka budú peniaze zarobené v danom stave.

Samotný algoritmus bude postupne prechádzať všetky políčka (stavy). Pre dané políčko si vypočíta stav, ktorý mu vznikne predajom prvej knihy, a na políčko tohto nového stavu skúsi uložiť peniaze, ktoré by takto zarobil. No čo ak sa na toto políčko už dostal predtým? Preto sa najskôr pozrie, či už na danom políčku nie je uložená väčšia hodnota, ako by zapisoval. Ak áno, tak nespraví nič, pretože by prepisoval optimálnejšie riešenie. V opačnom prípade zapíše novú hodnotu. To isté zopakuje pre druhú a tretiu knihu.

Keďže si pamätáme štvorrozmerné pole, kde každá zo súradníc môže mať veľkosť až n, pamäťová zložitosť bude $O(n^4)$. A keďže všetky tieto políčka počas výpočtu prechádzame, tak aj časová zložitosť bude $O(n^4)$.

Už máme skoro vzorové riešenie, ale n^4 je predsa len príliš pomalé a na najväčších vstupoch nestíha. Skúsme teda naše riešenie trochu zoptimalizovať.

Zlepšenie zložitosti

Pozrime sa bližšie na predaj jednej knihy. Zákazník príde, kúpi si jednu z vrchných troch kníh a odíde. Keďže sa jedna z vrchných troch kníh zobrala, štvrtá kniha z kopy sa "presunula" na tretie miesto. Toto sa deje pri každej kúpe knihy[2]. Treťou knihou na kope bude teda po ľubovoľných k nákupoch vždy $k+3$-tia v pôvodnej kope.

Nám si teda stačí pamätať len túto tretiu knihu a celkový počet kníh si z nej budeme vedieť ľahko. Celkovo teda máme tri vnorené cykly, čo nám dáva časovú a aj pamäťovú zložitosť $O(n^3)$.

Listing programu (C++)

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

using namespace std;

#define MIN_INFTY -56789

int n, B[542], H[542];
int data[542][542][542];

int main(){

    cin >> n;   //načítame vstup
    for(int i = 0; i<n; ++i) cin >> H[i];
    for(int i = 0; i<n; ++i) cin >> B[i];

    H[n] = H[n+1] = H[n+2] = 0;
    B[n] = B[n+1] = H[n+2] = 0;
    //vytvoríme si sentinely, t.j. hranice na okraji našich hodnôt, aby sme
    //nemuseli ošetrovať hraničné prípady. V tomto prípade doplníme hodnoty o
    //tri bezcenné knihy na spodku kopy a troch švorc kupcov na konci. Je
    //ľahké vidieť, že v každom optimálnom riešení si títo traja kupci vezmú
    //tieto tri knihy, a nám to ušetrí kopu if-ov

    for(int i = 0; i<2; ++i) {  //inicializujeme si pole
        for(int j = 0; j<n+2; ++j) {
            for(int k = 0; k<(n+2); ++k) {
                data[i][j][k] = MIN_INFTY;
            }
        }
    }

    data[0][1][2] = 0; //počiatočný stav

    for(int i=0; i<n+1; i++){   //prejdeme cez jednotlivé stavy
        for(int j=i+1; j<n+2; j++){
            for(int k=j+1; k<n+3; k++){
                int current = data[i][j][k];
                data[j][k][k+1] = max(data[j][k][k+1], current + H[i]*B[k-2]);
                data[i][k][k+1] = max(data[i][k][k+1], current + H[j]*B[k-2]);
                data[i][j][k+1] = max(data[i][j][k+1], current + H[k]*B[k-2]);
            }
        }
    }

    cout<<data[n][n+1][n+2]<<endl;  //vypíšeme stav, kde nám ostávajú nami
                                    //doplnené bezcenné knihy
}
```

Vzorové riešenie

Pamäťová zložitosť $O(n^3)$ však stále nestačí na plný počet bodov, skúsme ju teda trochu orezať.

[2] Až na okrajový prípad, keď sú na kope menej ako štyri knihy.

Pri predchádzajúcej optimalizácií sme využili fakt, že tretia kniha na kope sa nám pri každom predaji zmení na tú pod ňou. Tento fakt nám ale dáva viac, než sme využili. Vďaka tomu totiž vieme, že zo stavov, kde tretia kniha je k-ta sa vieme dostať jedným predajom len do stavov, kde tretia kniha je $k+1$-vá. A naopak, do stavov kde tretia kniha je k-ta sa vieme dostať len zo stavov, kde tretia kniha je $k-1$-vá. Keďže potrebujeme len stavy $k-1$, nemusíme si pamätať všetky predchádzajúce. Stačí nám, keď si v jednom riadku poľa budeme pamätať už vypočítané stavy a do druhého budeme zapisovať novovypočítané stavy. No a keď ich dorátame, tak nám stačí len vymeniť riadky a počítať ďalej.

Tým, že si v jednom momente pamätáme stavy len pre dva prípady tretej knihy na kope, sme práve odobrali ďalšie n z pamäťovej zložitosti. Pamäťová zložitosť teda bude $O(n^2)$.

Listing programu (C++)

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


using namespace std;

#define MIN_INFTY -1023456789

int n, B[542], H[542];
int data[2][542][542];
//Prvá súradnica nám bude reprezantovať tretiu knihu v kope.


int main(){
    cin >> n;
    for(int i = 0; i<n; ++i) cin >> H[i];
    for(int i = 0; i<n; ++i) cin >> B[i];

    for(int i = 0; i<2; ++i) {
        for(int j = 0; j<n+2; ++j) {
            for(int k = 0; k<n+2; ++k) {
                data[i][j][k] = MIN_INFTY;
            }
        }
    }

    data[0][0][1] = 0; //Inicializujeme si počiatočný stav

    for(int i = 0; i<n; ++i) {
        for(int j = 0; j<n+2; ++j) {
            for(int k = 0; k<n+2; ++k) {
                int current = data[i%2][j][k];
                data[(i+1)%2][j][k]   = max(data[(i+1)%2][j][k], current+B[i]*H[i+2]);
                data[(i+1)%2][i+2][k] = max(data[(i+1)%2][i+2][k], current+B[i]*H[j]);
                data[(i+1)%2][j][i+2] = max(data[(i+1)%2][j][i+2], current+B[i]*H[k]);
                //simulujeme predaj postupne tretej, prvej a druhej knihy v kope
            }
        }
    }
    cout << max(data[n%2][n][n+1], data[n%2][n+1][n]) << endl;
}
```

7. Ochrana pred povodňami

Mário
(max. 10 b za popis, 10 b za program)

Najprv si popíšeme myšlienkovo jednoduchšie riešenie, kde po pridaní každej steny prehľadáme celú mapu odznova a ukážeme si zopár vylepšení tohto riešenia. Nakoniec si ukážeme vzorové riešenie – offline – načítame celý vstup, spočítame riešenie a vypíšeme celý výstup.

Viacnásobné prehľadávanie

S využitím prehľadávanie do šírky (BFS – breadth-first-search) alebo prehľadávania do hĺbky (DFS – depth-first-search) sa dá vymyslieť jednoduché, funkčné riešenie, ktoré mohlo získať na testovači aspoň 3 body.

V takomto riešení si udržujeme mapku celého ostrova ako dvojrozmerné pole. V nej máme zaznačené, ktoré políčka sú voľné a na ktorých je múr. Postupne načítavame vstup a po pridaní každého kúska múru spustíme prehľadávanie od okrajov mapy. Označíme/navštívime tak všetky políčka zaliate vodou. Spočítame si počet zaliatych políčok z a výstup programu, počet chránených políčok, bude: $w \cdot h - z$.

Pri každom prehľadávaní navštívime každé políčko mapy najviac raz, teda časová zložitosť jedného prehľadávania bude $O(w \cdot h)$. Prehľadávanie púšťame po načítaní každého z n múrov, teda celková časová zložitosť bude $O(n \cdot w \cdot h)$.

Mierne zlepšenia[3] môžu byť napríklad:

[3] Ktoré nezmenia asymptotickú časovú zložitosť pre všeobecný prípad.

- spúšťať prehľadávanie len ak sú okolo múru iné múry – len vtedy vieme uzavrieť nejakú oblasť ochranným múrom
- spúšťať prehľadávanie len ak uzavrieme súvislý komponent múrov
- spúšťať prehľadávanie od miesta, kde sme postavili múr – ak nájdeme okraj, táto časť prehľadávala územie nechránené pred vodou. Ak okraj nenájdeme, prehľadali sme územie ochránené pred vodou.
- ...

Takýmto spôsobom sa na testovači dalo získať až 5 bodov.

Vzorové riešenie

Na predošlom riešení si môžeme všimnúť, že robíme tú istú prácu stále odznova. Políčka zaliate vodou prehľadávame v každom kroku. Aby sme dosiahli lepšiu časovú zložitosť, nemôžeme si dovoliť prehľadávať celú mapu veľakrát. Chceli by sme teda vedieť použiť údaje z minulosti.

Ale ako? Keď uzavrieme múr – ochránime územie vnútri. Chceli by sme vedieť, aké je toto územie veľké. Po predošlom prehľadávaní je ale vonkajšok aj vnútro múru označené ako voda, čo nám vôbec nepomôže.

Čo ak by sme to robili odzadu? Čo ak by sme múry namiesto pridávania odoberali? Územie vonku múru môže byť označené ako VODA, územie vnútri ako CHRANENE. Ak odstránime stenu, vieme spustiť prehľadávanie na políčkach označených ako CHRANENE a spočítať, koľko políčok bolo vnútri – koľko ich bude zaliatych vodou. Nevieme teda múr stavať, ale vieme ho rýchlo búrať.

Presne toto aj urobíme. Vyriešime úlohu odzadu. Presnejšie, načítame celý vstup, vypočítame všetky riadky riešenia a vypíšeme ich v opačnom poradí, ako sme ich počítali.

Opäť budeme mať dvojrozmernú mapu a na nej bude každé políčko buď STENA, VODA alebo CHRANENE.

Všetky políčka mapy najprv nastavíme ako CHRANENE. Načítame si celý vstup a do mapy zaznačíme všetky STENY.

Pomocou jedného prehľadávania od okrajov mapy potom zaplavíme všetko územie, ktoré sa dá, VODOU. Prehľadávanie navštívi všetky zaplavené políčka, a tak prvá odpoveď (posledná vypísaná.) bude $w \cdot h - zaplavene$.

Následne spracúvame vstup odzadu. Odstránime stenu[4] a pokiaľ stena susedila s vodou, spustíme prehľadávanie z tohto miesta po políčkach CHRANENE. Zaplavíme tak všetko územie, ktoré bolo doteraz vnútri múru.

Odpoveď v každom kroku je $w \cdot h - zaplavene$. Ak stena nesusedila s vodou, zostane odpoveď rovnaká ako v predošlom kroku, lebo veľkosť zaplaveného územia sa vtedy nezmení. Výstupy vypíšeme v opačnom poradí ako sme ich vypočítali.

V tomto algoritme každé políčko mapy navštívime prehľadávaním práve raz[5] a tak je jeho časová zložitosť len $O(w \cdot h)$.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
#include <queue>
#define For(i,N) for(int i=0; i<N; i++)
#define pb push_back
#define mp make_pair
#define ff first
#define ss second

#define VODA 0
#define CHRANENE 1
#define STENA 2

using namespace std;

typedef pair<int,int> pii;

int dx[] = {-1,0,1,-1,1,-1,0,1};
int dy[] = {-1,-1,-1,0,0,1,1,1};

int w, h, n, zaplavene=0;
vector<vector<int> > mapa;
vector<pii> steny;      // vstup
vector<int> odpovede;   // vystup

// z policka [x,y] rozlievame vodu do vsetkych CHRANENYCH
void bfs(int x, int y){
    if(mapa[y][x] != CHRANENE)
        return;
```

[4] Najskôr ale skúste povedať váš tip: kto je za stenou? Priateľ z detstva? Bývalá láska? Niekto komu ste pomohli? Odstránime stenu?

[5] Na konci algoritmu je mapa prázdna – každé políčko je VODA.

```
    queue<int> q;
    q.push(x); q.push(y);
    mapa[y][x] = VODA;
    zaplavene++;

    while (!q.empty()){
        int x = q.front(); q.pop();
        int y = q.front(); q.pop();

        For(i,8){
            int nx = x+dx[i];
            int ny = y+dy[i];
            if(mapa[ny][nx] == CHRANENE){
                q.push(nx); q.push(ny);
                mapa[ny][nx] = VODA;
                zaplavene++;
            }
        }
    }
}


int main(){
    scanf("␣%d␣%d␣%d", &w, &h, &n);
    mapa.resize(h+2, vector<int> (w+2, CHRANENE));

    // po okrajoch mapy dame vodu
    For(i, w+2) mapa[0][i] = mapa[h+1][i] = VODA;
    For(i, h+2) mapa[i][0] = mapa[i][w+1] = VODA;

    // nacitame zoznam stien a zaznacime vsetky do mapy
    For(i,n){
        int x,y;
        scanf("␣%d␣%d", &x, &y);
        steny.pb(mp(x,y));
        mapa[y][x] = STENA;
    }

    // rozlievame vodu postupne od okrajov
    For(i,w){ bfs(i+1,1); bfs(i+1,h); }
    For(i,h){ bfs(1,i+1); bfs(w,i+1); }

    for(int i=n-1; i>=0; i--){
        odpovede.pb(w*h - zaplavene);

        int x = steny[i].ff;
        int y = steny[i].ss;

        // odstranime stenu
        mapa[y][x] = CHRANENE;

        // ak je aspon 1 sused voda, zalejeme vnutro komponentu
        For(j,8)
            if(mapa[y+dy[j]][x+dx[j]] == VODA){
                bfs(x,y);
                break;
            }
    }

    for(int i=n-1; i>=0; i--)
        printf("%d\n", odpovede[i]);

    return 0;
}
```

Úloha sa dá naprogramovať aj trocha jednoduchšie, tu môžete vidieť stručnejšiu implementáciu algoritmu.

Listing programu (C++)

```
#include<cstdio>
#include<vector>
#define For(i, n) for(int i = 0; i<(n); ++i)
#define CHRANENE 0
#define VODA 1
#define STENA 2
typedef std::vector<int> vi;

int w,h,n, mokre = 0, M[2047][2047];
int dx[] = {1,1,1,0,-1,-1,-1,0};
int dy[] = {1,0,-1,-1,-1,0,1,1};

void zaplav(int x, int y) {
    if (x<0 || y <0 || x>=w || y>=h) return;
    if (M[x][y] != CHRANENE) return;
    M[x][y] = VODA;
    mokre++;
    For(d, 8) zaplav(x+dx[d], y+dy[d]);
}

int main() {
    scanf("%d%d%d", &w, &h, &n); w+=2; h+=2;
    vi X = vi(n), Y = vi(n), A = vi(n);
    For(i,n) {
        scanf("%d%d", &X[i], &Y[i]);
        M[X[i]][Y[i]] = STENA;
```

```
    }
    zaplav(0,0);

    for(int i = n-1;i>=0;--i) {
        A[i] = w*h-mokre;
        M[X[i]][Y[i]] = CHRANENE;
        For(d, 8) if (M[X[i]+dx[d]][Y[i]+dy[d]] == VODA) zaplav(X[i], Y[i]);
    }
    For(i, n) printf("%d\n", A[i]);
}
```

Žaba

8. O()kúzlení

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

Ako získať ľahké body

V piatich z ôsmich testovacích sád platilo, že $n \cdot q \leq 1\,000\,000$. To znamená, že pri každej otázke môžeme prejsť celým poľom, v ktorom si pamätáme, pod ktorým pohárikom sa nachádza guličkа a niečo s ním spraviť. Toto nám dáva takú voľnosť, že sa úloha blíži obtiažnosťou k úlohe číslo 4. Preto, ak ste nezískali za túto úlohu žiadne body, zoberte si to ako ponaučenie a nabudúce sa pozerajte, čo vám dovoľujú limity v zadaní.

Ako teda vymyslieť prvé riešenie, ktoré spracováva každú otázku v čase $O(n)$? Potrebujeme spracovávať dve operácie – invertovanie intervalu a zistenie najdlhšej podpostupnosti núl a následne jednotiek. Prvá operácia je ľahká. Naozaj prejdeme celý zadaný interval a zmením každé číslo na to opačné.

Čo sa týka druhej, použijeme veľmi jednoduché dynamické programovanie. Budeme poľom prechádzať zľava doprava a v každom momente si budeme pamätať dve čísla. Aká je najdlhšia podpostupnosť z doposiaľ videných prvkov, ktorá končí nulou (p_0) a aká je najdlhšia postupnosť z doposiaľ videných prvkov končiaca jednotkou (p_1). Nie je problém vymyslieť, ako sa tieto čísla budú meniť.

Ak budeme mať na aktuálnom políčku nulu, tak $p_0 = p_0 + 1$ a p_1 zostane nezmenené. V opačnom prípade, keď sa na políčku nachádza jednotka, tak p_0 zostane nezmenené a $p_1 = \max(p_0 + 1, p_1 + 1)$. Po prejdení celého poľa si jednoducho vyberieme väčšiu z hodnôt p_0 a p_1 ako dĺžku najdlhšej hľadanej podpostupnosti.

Pamäťová zložitosť takéhoto riešenia je $O(n)$ a časová $O(nq)$. Aj samotný program, ktorý implementuje toto riešenie je pomerne jednoduchý:

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
#include <cmath>
using namespace std;

#define For(i,n) for(int i=0; i<(n); i++)

int main() {
    int n;
    scanf("%d",&n);
    vector<int> A;
    For(i,n) {
        char c;
        scanf(" %c",&c);
        A.push_back(c-'0');
    }
    int q;
    scanf("%d",&q);
    For(i,q) {
        int t;
        scanf(" %d",&t);
        if(t==1) {
            int p0=0,p1=0;
            For(j,n) {
                if(A[j]==0) p0=p0+1;
                else p1=max(p0+1,p1+1);
            }
            printf("%d\n",max(p0,p1));
        }
        else {
            int z,k;
            scanf("%d %d",&z,&k);
            for(int j=z-1; j<k; j++) A[j]=(A[j]+1)%2;
        }
    }
}
```

Intervalový strom

Keď sa pozeráme na úlohu, mali by sme sa snažiť nájsť podobnosti s inými úlohami, ktoré sme riešili. Z veľkostí vstupu nám môže vyplynúť, že chceme každú operáciu vykonať v čase $O(\log n)$. Naviac naše operácie

sa zameriavajú na intervaly našeho poľa. Toto všetko by malo ukazovať na to, že chceme použiť intervalový strom.

Samozrejme, toto nemusí byť zakaždým pravda. Môže sa stať, že riešenie použije vyhľadávacie stromy, alebo otázky spracováva offline, alebo kopec iných možností. Vždy sa však oplatí si prejsť zoznam používaných konceptov a skúsiť ich aplikovať.

Na prvý pohľad to však pôsobí zvláštne. Klasický sčítací, či maximový/minimový strom nám je v tomto prípade úplne nanič. To však nie je jediné využitie intervalových stromov. Tie nám totiž ponúkajú oveľa všeobecnejšiu štruktúru, ktorá si pamätá v každom vrchole **nejaké informácie** o prislúchajúcom intervale a vie jednoduchým spôsobom spájať informácie pre synov do informácií pre otca. V našej úlohe teda musíme vymyslieť, čo si musí pamätať každý z vrcholov.

Vieme, že jedna z otázok, ktorú sa pýtame, je: "Aká je dĺžka najdlhšej podpostupnosti, ktorá začína nulami a končí jednotkami, na celom poli?" Čo v podstate znamená, že túto hodnotu si musí pamätať koreň našeho intervalového stromu. A ak si to pamätá on, musí si to pamätať každý vrchol. Pre každý vrchol si teda zapamätáme hodnotu p_{01} – dĺžku najdlhšej podpostupnosti, ktorá sa nachádza na danom intervale, začína nulami a končí jednotkami.

Otázkou ale je, či vieme z hodnôt p_{01} pre synov určiť túto hodnotu aj pre otca. Odpoveďou je, že nie. Ak totiž spojím dve takéto postupnosti, dostanem postupnosť, ktorá začína nulami, končí jednotkami, ale niekde v strede sa to ešte zmení z jednotiek na nuly a opačne. A to nie je dobre – v celej podpostupnosti máme mať len jeden zlom medzi nulami a jednotkami.

Pozrime sa, kde sa tento zlom nachádza. Ak sa nachádza v interale prislúchajúcom ľavému synovi, znamená to, že z pravého syna môžeme zobrať už iba jednotky. A kľudne môžeme zobrať všetky jedotky, ktoré sa v jeho intervale nachádzajú. Naopak, ak sa zlom nachádza v pravom synovi, z ľavého môžeme zobrať všetky nuly. To znamená, že každý vrchol si musí pamätať ďalšie dve hodnoty – p_0 je počet núl v jeho intervale a p_1 je počet jednotiek.

V tomto okamihu vieme všetky tri hodnoty vyrátať veľmi jednoduchým spôsobom. Hodnoty l_0, l_1 a l_{01} nech patria ľavému synovi, hodnoty r_0, r_1 a r_{01} pravému a tie s p patria otcovi. Potom platí:

$$p_0 = l_0 + r_0$$

$$p_1 = l_1 + r_1$$

$$p_{01} = \max(l_{01} + r_1, l_0 + r_{01}, l_0 + r_1)$$

Všetky vyššie uvedené vzorce by mali byť intuitívne, ak vám niektorý nie je jasný, nakreslite si to. Posledná možnosť $l_0 + r_1$ nastane vtedy, ak je zlom presne medzi ľavým a pravým synom.

Tieto tri hodnoty nám stačia, keď sa nám invertuje iba interval o dĺžke jeden. Keďže však budeme meniť aj väčšie intervaly, pridáme si do každého vrcholu ešte hodnotu p_{10} – najdlhšia podpostupnosť v tomto intervale, ktorá začína jednotkami a končí nulami. Jej rátanie je veľmi podobné tomu pri p_{01}:

$$p_{10} = \max(l_{10} + r_0, l_1 + r_{10}, l_1 + r_0)$$

Spracovávanie dlších intervalov

V našom riešení už vieme odpovedať na jeden typ operácie (otázku), tým že jednoducho z koreňa vypíšeme hodnotu p_{01}. Treba však vyriešiť, ako meniť náš strom na základe druhého typu operácií (inverz intervalu).

V prípade, že menený interval má dlžku 1, je všetko pomerne jednoduché. Proste zmeníme hodnotu tohto konkrétneho prvka v poli a následne postupujeme hore stromom až do koreňa, pričom zakaždým prepočítame hodnoty vo vrchole. Zložitosť takejto operácie je $O(\log n)$. To nám však nepomôže, keď máme zmeniť dlhší interval.

V takomto prípade použijeme metódu, ktorá sa volá lazy loading. Namiesto toho, aby sme začali zdola, začneme od koreňa a postupne budeme zisťovať, ktoré vrcholy majú celý svoj interval vnútri invertovaného intervalu. Keď nájdeme takýto vrchol, spravíme nasledovné: vieme, že všetky čísla, ktoré ležia pod týmto vrcholom sa zmenili na opačné. Tým pádom sa nám vymenia hodnoty p_0 a p_1 a tiež p_{01} a p_{10} (preto sme si ju pridali). Tento vrchol x teda vieme opraviť na správne požadované hodnoty bez toho, aby sme mali správne hodnoty aj v jeho synoch.

Čo však s vrcholmi, ktoré ležia pod ním? Tým zatiaľ zatajíme, že sa zmenili, akurát si do tohto vrchola x v premennej *lazy* zaznačíme, že ešte nepovedal vrcholom pod sebou, že sa majú invertnúť. Následne prepočítame hodnoty všetkých vrcholov na ceste z koreňa do tohto vrchola a máme vybavené.

Čo však keď budeme potrebovať hodnotu z nejakého vrchola, ktorý má byť invertovaný, ale ešte o tom nevie? Ak sa stane niečo takéto, určite budeme musieť prejsť cez vrchol x. Ten však v momente, keď cez neho prechádzame, zbadá, že ešte nepovedal vrcholom pod sebou, že sa majú invertovať, tak im to rýchlo oznámi a

zmaže si hodnotu *lazy*. Vrcholy pod ním si teda povymieňajú svoje hodnoty a zaznačia si hodnotu *lazy*, lebo to musia povedať tým pod sebou. Tým pádom vždy, keď potrebujeme použiť hodnotu nejakého vrchola, tento vrchol už pozná svoje skutočné hodnoty a vie nám ich povedať.

Dôvod, prečo je niečo takéto rýchlejšie je ten, že robíme iba robotu, ktorá je absolútne nevyhnutná. Ak sa totiž nikdy nespýtame na žiaden vrchol pod x, načo by sme im oznamovali, že sú zmenené? A vďaka premenným *lazy* si vieme pamätať, kto ešte musí niečo povedať. Tento princíp vôbec nie je taký zložitý, ako by sa mohlo na prvý pohľad zdať. Odporúčame si nakresliť nejaký strom na papier a popozerať sa, ako by sa tie hodnoty mali meniť a takisto si poriadne preštudovať nižšie uvedený program, hlavne funkcie *update*() a *process_lazy*().

Zložitosť takejto operácie nie je o nič väčšia ako klasické intervalové vyhľadávanie – $O(\log n)$. Dostávame algoritmus s časovou zložitosťou $O(q \log n)$ a pamäťovou zložitosťou $O(n)$.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>
#include <cmath>
using namespace std;

#define For(i,n) for(int i=0; i<(n); i++)

struct vertex {
    int p0,p1,p01,p10;
    int lazy;
};

vertex T[3000047];
int moc2;

vertex create(int a, int b, int c, int d) {
    vertex v;
    v.p0=a; v.p1=b; v.p01=c; v.p10=d;
    v.lazy=0;
    return v;
}

void reverse_vertex(int v) {
    swap(T[v].p0,T[v].p1);
    swap(T[v].p01,T[v].p10);
}

void process_lazy(int v) {
    if(T[v].lazy%2==0) return;
    reverse_vertex(v);
    T[v].lazy=0;
    if(v>=moc2) return;
    T[2*v].lazy++; T[2*v+1].lazy++;
}

int answer() {
    process_lazy(1);
    return T[1].p01;
}

void repair(int v) {
    T[v].p0=T[2*v].p0+T[2*v+1].p0;
    T[v].p1=T[2*v].p1+T[2*v+1].p1;
    T[v].p01=max(T[2*v].p01+T[2*v+1].p1,max(T[2*v].p0+T[2*v+1].p01,T[2*v].p0+T[2*v+1].p1));
    T[v].p10=max(T[2*v].p10+T[2*v+1].p0,max(T[2*v].p1+T[2*v+1].p10,T[2*v].p1+T[2*v+1].p0));
}

//<from,to) je interval, ktory potrebujeme zmenit, <zac,kon) je interval, ktory patri vrcholu v
void update(int v, int from, int to, int zac, int kon) {
    process_lazy(v);
    if(to<=zac || kon<=from) return;
    if(from<=zac && to>=kon) {
        T[v].lazy++;
        process_lazy(v);
        return;
    }
    int stred=(zac+kon)/2;
    update(2*v,from,to,zac,stred);
    update(2*v+1,from,to,stred,kon);
    repair(v);
}

int main() {
    int n,m;
    scanf("%d",&n);
    moc2=1;
    while(n>moc2) moc2*=2;
    For(i,n) {
        char c;
        scanf("_%c",&c);
        if(c=='0') T[moc2+i]=create(1,0,1,1);
        else T[moc2+i]=create(0,1,1,1);
    }
    for(int i=moc2-1; i>0; i--) repair(i);
    scanf("%d",&m);
    int xx;
    For(i,m) {
```

```
        scanf("␣%d",&xx);
        if(xx==1) printf("%d\n",answer());
        else {
            int l,r;
            scanf("␣%d␣%d",&l,&r);
            l--;
            update(1,l,r,0,moc2);
        }
    }
return 0;
}
```