Korešpondenčný seminár z programovania
XXXV. ročník, 2017/18
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola zimnej časti

Matúš

1. Črieda pažravých vedúcich

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

Pre zadaný počet dielikov čokolády n chceme vedieť, koľkým najviac vedúcim ich vieme spravodlivo rozdeliť, ak je vedúcich nepárne veľa. V preklade to znamená, že hľadáme najväčšie nepárne číslo k, ktoré delí n bezo zvyšku.

Úplne najjednoduchšie riešenie môže fungovať tak, že postupne budeme cyklom prechádzať čísla $1 \dots n$ a pre každé otestujeme, či je nepárne a zároveň deliteľom n. Ak také nájdeme, uložíme si ho, a po skončení cyklu tak dostaneme správnu odpoveď. Takéto riešenie hrubou silou má pre jedno n časovú zložitosť $O(n)$, pre q požiadavok $O(n \cdot q)$ a mohli ste zaň dostať maximálne jeden bod.

Listing programu (C++)

```
#include <cstdio>
#include <cmath>

using namespace std;

int main(){
    long long n,q;
    scanf("%lld", &q);
    for (long long i = 0; i < q; i++){
        scanf("%lld", &n);
        long long m = 1;
        for (int k = 2; k <= n + 1; k++){
            if (n % k == 0 && k % 2 == 1){
                m = k;
            }
        }
        printf("%lld\n", m);
    }
}
```

Predošlé riešenie vieme vylepšiť tak, že budeme deliteľe skúšať nie po n ale iba $\sqrt{n}$. Ak by totiž n malo deliteľ d väčší ako $\sqrt{n}$ potom nutne $n \div d < \sqrt{n}$. Prejdeme teda $1 \dots \sqrt{n}$ a pre každé číslo k z tohto intervalu sa pozrieme, či delí n. Ak áno, pozrieme sa, či k alebo $n \div k$ je nepárne a ak je, aktualizujeme jeho hodnotou doterajšie nájdené maximum. Toto riešenie má celkovú časovú zložitosť $O(\sqrt{n} \cdot q)$ a na testovači získalo maximálne dva body (aj to len v rýchlych jazykoch).

Listing programu (C++)

```
#include <cstdio>
#include <cmath>
#include <algorithm>

using namespace std;

int main(){
    long long n,q;
    scanf("%lld", &q);
    for (long long i = 0; i < q; i++){
        scanf("%lld", &n);
        long long m = 1;
        for (long long j = 1; j < (long long)ceil(sqrt(n)); j++){
            if (n % j == 0){
                if (j % 2 == 1) m = max(m,j);
                if ((n / j) % 2 == 1) m = max(m,j)
            }
        }
        printf("%lld\n", m);
    }
}
```

Dostávame sa ku vzorovému riešeniu. Zamyslime sa, ako vieme efektívne nájsť *všetky* deliteľe čísla, alebo aspoň ich počet. Odpoveďou je rozklad na prvočísla, s ktorým ste sa už určite stretli. Každý deliteľ čísla n je súčinom niekoľkých (možno aj nula, možno aj všetkých) prvočísel z prvočíselného rozkladu n. Aby sme dostali

nepárny súčin, musíme násobiť iba nepárne čísla. Najväčší nepárny deliteľ čísla n teda bude súčin všetkých nepárnych prvočísel z prvočíselného rozkladu n[1].

To znamená, že z rozkladu stačí eliminovať všetky dvojky, keďže dvojka je jediné párne prvočíslo. V praxi nám teda stačí číslo n deliť dvojkou kým to ide a akonáhle to nejde, dostali sme náš hľadaný najväčší nepárny deliteľ.

Náš algoritmus urobí pre každé n iba toľko krokov, koľkokrát je n deliteľné dvojkou. V najhoršom prípade je n mocninou dvojky a algoritmus by sa delením dostal až ku jednotke. Urobil by teda logaritmicky[2] veľa krokov od veľkosti n a teda časová zložitosť tohto riešenia je $O(\log_2 n \cdot q)$. Toto riešenie už získa po otestovaní plný počet bodov.

Listing programu (C++)

```
#include <cstdio>
#include <cmath>
#include <algorithm>

using namespace std;

int main(){
    long long n,q;
    scanf("%lld", &q);
    for (long long i = 0; i < q; i++){
        scanf("%lld", &n);
        while (n % 2 == 0) n /= 2;
        printf("%lld\n", n);
    }
}
```

Roman B. a Žaba
(max. 12 b za popis, 8 b za program)

2. Oštara s huncútmi malými

Povieme si najprv niečo o riešení za polovicu bodov (kde $v = 1$) a od neho sa dostaneme k vzorovému riešeniu.

Odporúčame vám si dôsledne preštudovať uvedené zdrojové kódy (keď budú zverejnené). Najmä riešitelia, ktorí ešte nie sú úplne zbehlí v programovaní, sa tak dozvedia veľa užitočných trikov. Každé riešenie obsahuje komentáre o tom, čo sa presne deje a prečo, netreba sa teda kódov báť.

Riešenie za polovicu bodov

Zo zadania sme museli najprv pochopiť, akým spôsobom sa mravčia ríša pohybuje pozdĺž chodníka. Toto správanie potom vieme jednoducho nasimulovať pomocou cyklov.

Do pamäte si uložíme pole reprezentujúce chodník, ktoré si nazveme `chodnik[]`. Na x–tej pozícii poľa bude uložené číslo dňa, v ktorom na x-tý meter chodníka dopadla omrvinka. Okrem poľa `chodnik[]` si budeme v našej simulácii pamätať aj číslo aktuálneho dňa (premenná `den`) a pozíciu, na ktorej končí mravčia ríša (premenná `kde_som`). Tieto premenné budú mať na začiatku hodnotu 0.

Algoritmus bude prebiehať nasledovne:

1. Posúvame koniec mraveniska po chodníku ďalej pokiaľ platia obe nasledujúce podmienky:

   - ešte sme nedosiahli koniec chodníka: `kde_som` $<$ `n+1`
   - na ďalšom metri chodníka už je omrvinka: `chodnik[kde_som + 1]` $<=$ `den`

   Pre posun na ďalší meter chodníka zväčšíme `kde_som` o 1.

2. Keď sme sa dostali najďalej, ako to v daný deň išlo, inými slovami jedna z podmienok z kroku 1 prestala platiť, vypíšeme aktuálnu hodnotu `kde_som` a zvýšime číslo dňa o 1. Po zvýšení hodnoty `den` sa už možno budeme vedieť dostať po chodníku ďalej, keďže nám na chodníku možno pribudli ďalšie omrvinky. Ak sme sa ešte nedostali na koniec chodníka, znova pokračujeme krokom 1 a skúšame sa posunúť ďalej.

Listing programu (C++)

[1] Pre matematických detailistov: ak obsahuje prvočíselný rozklad čísla n nejaké prvočíslo vo vyššej mocnine, chápeme to tak, že obsahuje viac rovnakých prvočísel.

[2] Logaritmus čísla n so základom a nám hovorí, na koľkú treba umocniť a, aby sme dostali n. Napríklad $\log_2 16 = 4$ a $\log_3 27 = 3$

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

using namespace std;

int main()
{
    int n, v;
    cin >> n >> v;

    // pole obsahujuce casy kedy padli omrvinky na chodnik
    // chodnik[METER] = CISLO_DNA_KEDY_PADLA
    vector<int> chodnik(n+1);

    for(int i=0; i<n; i++)
    {
        int x, y;
        cin >> x >> y;
        chodnik[x]=y;
    }

    // premenne udrzujuce stav simulacie
    int kde_som=0;
    int den=0;

    while (kde_som != n+1)
    {
        // Krok 1: posuvaj sa kym mozes
        // teda, kym bud nedorazis na koniec chodnika
        // alebo na dalsom metri chodnika este nepadla omrvinka
        while (kde_som != n+1 && chodnik[kde_som+1]<=den)
            kde_som++;

        // Krok 2: kedze skoncil cyklus, uz sa dalej nevieme pohnut
        // vypiseme vysledok, zvysime cislo dna.
        cout << kde_som << endl;
        den++;

        // Nasleduje dalsia otacka cyklu (sme vo while cykle)
        // Vyhodnoti sa podmienka, ci este nie sme na konci chodnika
    }
}
```

$v > 1$ – jednoduché, no trochu pomalé riešenie

Ak sme chceli získať na testovači aspoň 6 bodov, bolo sa treba popasovať aj so situáciami, kde je expanzívnosť mravcov väčšia než 1.

Ako sa takéto riešenie líši od toho predošlého? V kroku 1 našej simulácie sme schopnosť posunúť sa ďalej overovali podľa toho, či na nasledujúce políčko už dopadla omrvinka. Avšak mravce dokážu od svojej hranice dosiahnuť na v ďalších políčok, musíme preto urobiť takýchto kontrol v – jednu pre každé políčko, na ktoré dočiahnu. Všimnite si, že aj predtým sme ich potrebovali v, akurát v bolo 1, a preto nám stačil jednoduchý `if`.

Na overenie druhej podmienky kroku 1 použijeme `for` cyklus, ktorý bude kontrolovať, či `chodnik[kde_som + i] <= den` pre všetky i od 1 po v. Samozrejme, treba si dať pozor, aby sme sa takýmto skúšaním nepozerali za koniec chodníka.

Listing programu (C++)

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

using namespace std;

int main()
{
    int n, v;
    cin >> n >> v;

    // pole obsahujuce casy kedy padli omrvinky na chodnik
    // chodnik[METER] = CISLO_DNA_KEDY_PADLA
    vector<int> chodnik(n+1);

    for(int i=0; i<n; i++)
    {
        int x, y;
        cin >> x >> y;
        chodnik[x]=y;
    }

    // premenne udrzujuce stav simulacie
    int kde_som=0;
    int den=0;

    // simulácia pre všetky dni
    while (kde_som < n+1)
    {
        // urcuje, ci sa dokazem pohnut dalej
        // a teda ma zmysel prehladavat nasledujucich V metrov chodnika
        bool mozem = true;
```

```
        // tento cyklus si nazvime Erik
        // v rámci tohto dňa sa dostane tak ďaleko ako sa len dá
        while (mozem)
        {
            mozem = false;

            // idem prehladavat nasledujucich V policok
            for (int i=1; i<=v; i++)
            {
                if ((kde_som+i) <= n+1 && chodnik[kde_som+i] <= den)
                {
                    // ak som nasiel padnutu omrvinku niekde v dosahu,
                    // zapisem si, ze som ju nasiel, a potom budem moct z toho
                    // noveho miesta opat hladat, ci sa v nasledujucich V
                    // polickach nachadza dalsia omrvinka
                    mozem = true;

                    kde_som = kde_som+i;

                    // ak som nasiel, nemusim teraz hladat vo vsetkych V polickach
                    // a radsej rovno skoncim,
                    // kedze cyklus Erik v dalsej otacke spusti nove hladanie
                    // z tohto noveho miesta, ktore sme si predchvilou
                    // ulozili do kde_som
                    break;
                }
            }
        }

        cout << kde_som << endl;
        den++;
    }

    return 0;
}
```

Prečo sme za toto riešenie nedostali všetky body?

Samozrejme, takéto riešenie je pomalé. Zaujímavá otázka však je, prečo? Uvedomme si, čo sa v algoritme deje. Každý deň sa pozeráme na v nasledujúcich políčok a zisťujeme, či na niektoré z nich nepadla omrvinka. Hodnota v však môže byť veľká a mohlo sa nám stať, že všetky omrvinky padli v posledný možný deň. Poprípade omrvinky, ktoré padli ako prvé, padli na koniec chodníka.

Z toho vyplýva, že v najhoršom možnom prípade budeme musieť pre každý deň (počet dní označme max_{den}) naozaj skontrolovať v políčok a pri tom sa ani vôbec neposunúť. To vedie k zložitosti $O(max_{den} \cdot v + N)$, čo je pre najväčšie možné hodnoty zo zadania priveľa.

Iný prístup

Namiesto toho, aby sme simulovali jednotlivé dni, poďme simulovať dopady jednotlivých omrviniek. Pomôže nám, že omrvikny sú na vstupe zoradené podľa času dopadu **vzostupne** – od prvej padnutej po poslednú.

Predstavme si, že stojíme na x-tom metri a na chodník dopadla ďalšia omrvinka. V závislosti od miesta jej dopadu vykonáme jednu z nasledovných akcií:

a) Ak dopadla do už obsadeného územia, čiže na meter menší alebo rovný x, nemá zmysel niečo robiť. Mravce majú záujem expandovať len vpred, nie vzad.

b) Ak dopadla tak ďaleko, že na ňu práve nedočiahneme, nemá zmysel sa momentálne pokúšať expandovať. Do budúcna by sa nám ale táto informácia mohla zísť, preto si ju musíme niekam zapísať. Vytvoríme si pole `uz_padla[]`, do ktorého si na i-tu pozíciu zaznačíme, či už na i-ty meter chodníka padla omrvinka (`uz_padla[i] = true`). Pole `uz_padla[]` bude obsahovať hodnoty typu `boolean`, na začiatku sú všetky `false`.

c) Ak omrvinka dopadla pred nás, ale ešte stále v našom dosahu, budeme sa posúvať dopredu. Na rozdiel od predošlých riešení by sme však chceli každý meter chodníka skontrolovať najviac raz.

Ako sa múdro posúvať ?

Keď omrvinka padla niekam v rámci nášho dosahu, chceme sa k nej hneď posunúť. Ako však potom rýchlo nájsť ďalšiu omrvinku v dosahu? A ako to spraviť bez toho, aby sme zakaždým pýtali na tie isté políčka?

Na úplnom začiatku stojíme na metri 0 a vieme, že na chodník ešte nepadla žiadna omrvinka. Začneme omrvinky spracovávať v poradí, v akom sa objavili na vstupe, teda podľa času ich dopadu. Kým padajú ďalej ako je vzdialenosť v, nevieme sa posunúť ďalej a iba si tieto omrvinky značíme do poľa `uz_padla[]`.

Zmena nastane pri prvej omrvinke, ktorá dopadne na miesto x, kde $x \leq v$. V tom momente sa môžeme posunúť na pozíciu x. Z novej pozície však možno dosiahneme na omrvinky, ktoré padli skôr. Mohli by sme sa preto začať pozerať na nasledujúcich v políčok a s pomocou poľa `uz_padla[]` zisťovať, či sa na nich nenachádza nejaká omrvinka. Pričom vždy keď nájdeme ďalšiu omrvinku v dosahu, tak sa posunieme na jej miesto

Uvedomme si však jednu dôležitú vec. **Až po meter v sa žiadna ďalšia omrvinka nenachádza**. Na začiatku bol totiž chodník prázdny a omrvinka, ktorá padla na pozíciu x bola prvá omrvinka, ktorá padla do nášho dosahu. Nemusíme teda kontrolovať políčka x až v, **stačí nám skontrolovať pozície od $v+1$ po $x+v$**.

Táto vlastnosť naviac platí aj vo všeobecnosti. Ak sme zastavili na metri y, pričom sme sa nevedeli posunúť ďalej, bolo to preto, lebo sme skontrolovali všetky políčka po $y+v$ a na žiadnom z nich sa omrvinka nenachádzala. Keď teda do nášho dosahu padne prvá omrvinka, stačí kontrolovať políčka začínajúc od pozície $y+v+1$.

Toto zlepšenie spôsobí, že každý meter chodníka skontrolujeme najviac raz. Zakaždým totiž kontrolovanie začneme od poslednej skontrolovanej pozície. A keďže chodník má dĺžku n, budeme potrebovať $O(n)$ času.

Samozrejme, zadanie po nás chcelo, aby sme vypísali pozície, na ktorých stoja mravce v každý možný deň. To však vieme do nášho programu ľahko pridať. Ak totiž stojíme na metri x, naposledy sme úspešne spracovali omrvinku, ktorá padla v deň d_1 a ďalšia omrvinka padne až v deň d_2 tak vieme, že odpoveď pre dni d_1 až d_2-1 bude x.

Je si však treba uvedomiť, že veľkosť výstupu nesúvisí s hodnotou n a jeho vypísanie nám tiež zaberie čas úmerný počtu vypísaných čísel. Ak si teda posledný deň, keď padne nejaká omrvinka označíme max_{den}, tak môžeme povedať, že časová zložitosť nášho algoritmu je $O(n+max_{den})$. Pamäťová zložitosť bude $O(n)$, keďže na naprogramovanie nám stačí jedno pole dĺžky n.

Listing programu (C++)

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


using namespace std;

int main()
{

    int n, v;
    cin >> n >> v;

    // pole obsahujuce casy kedy padli omrvinky na chodnik
    // chodnik[METER] = CISLO_DNA_KEDY_PADLA
    vector<pair<int, int>> zrnka;
    vector<bool> uz_padla(n+2, false);
    uz_padla[n+1] = true;

    // premenne ktore budeme potrebovat - kde som a v kolkom dni
    // spomenute v kroku #1
    int kde_som = 0;
    int kam_kontroloval = 0;
    int kolke_zrnko = 0;

    for(int i=0; i<n; i++)
    {
        int meter, den;
        cin >> meter >> den;
        zrnka.push_back(make_pair(den, meter));
    }

    // pamatame si aj cislo dna, pretoze vzdy ked spracujeme niekolko zrniek,
    // potrebujeme vypisat udaj, na akej pozicii sa prave nachadzame.
    // - takze spracuvavaj zrnka az pokial dalsie zrnko uz nepatri
    // do dalsieho dna. Cyklus JOZEF vtedy skonci, vypiseme vysledok pre dany den
    // a zase spustime JOZEFa nech sa cykly pre zrnka z nasledujuceho dna
    for (int den=0; kde_som < n+1; den++)
    {
        while (zrnka[kolke_zrnko].first == den) // cyklus JOZEF
        {
            int kam_padlo = zrnka[kolke_zrnko].second;

            // dva krat s tym istym zrnkom robit nechceme!
            kolke_zrnko++;

            if (kam_padlo <= kde_som)
                // pripad A - padlo niekam za nas - a teda nas nezaujima
                continue;

            else if (kam_padlo > kde_som + v)
            {
                // pripad B - padlo niekam za nas aktualny dosah
                // zapiseme si to, bude nam to treba neskor
                uz_padla[kam_padlo] = true;
                continue;
            }

            else
            {
                // padlo niekam v nasom dosahu
                // takze urcite sa chceme hned nanho presunut
                kde_som = kam_padlo;

                // vsimnite si tieto dve podmienky vo WHILE cykle
                // vyhladavam len od miesta poslednej kontroly po aktualny dosah A ZAROVEN
                // vyhladavam pokial som uz nekontroloval koniec chodnika -> to
                // by znamenalo, ze uz dokazeme koniec chodnika dosiahnut a tym padom
```

```
                // sme vyhrali a uz nic vyhladavat nemusime
                while (kam_kontroloval < kde_som+v && kam_kontroloval < n+1)
                {
                    // skontroloval som policko - takze hranicu kontroly mozem posunut
                    // o policko dalej
                    kam_kontroloval++;

                    // kontrolujem prave policko, na ktore sme si predtym zaznacili,
                    // ze padlo zrnko
                    // tak rovno na to policko skocime!
                    if (uz_padla[kam_kontroloval])
                        kde_som = kam_kontroloval;
                }
            }
        }

        cout << kde_som << endl;
    }

}
```

Erik a Mário

3. Korman je gurmán

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

Našou úlohou je usporiadať pole čísel od najmenšieho po najväčšie len pomocou toho, že budeme čísla prenášať na začiatok poľa. Presnejšie povedané, chceme zistiť, na koľko najmenej takýchto operácií sa dá naše pole usporiadať.

V tomto vzoráku si v prvej časti predstavíme základné pozorovania potrebné pre vyriešenie tejto úlohy – popisy vašich riešení by mohli obsahovať aspoň hrubo zvýraznené myšlienky. V druhej časti si ukážeme, ako sa dá jednoducho vypočítať riešenie v čase $O(n)$, ak vstup obsahuje len čísla 1 až n a ako sa dá toto riešenie zovšeobecniť pomocou triedenia na ľubovolné pole a dostať riešenie v čase $O(n \log n)$. Ak vás zaujíma len optimálne riešenie, môžete po prečítaní prvej časti preskočiť na poslednú časť, kde si ukážeme, ako získať časovú zložitosť $O(n)$ pomocou zásobníka.

Myšlienka riešenia

Prvé pozorovanie, ktoré nám pomôže pri riešení, je, že každé pole dĺžky n vieme usporiadať s použitím n operácií. Najprv presunieme na začiatok najväčšie číslo, potom druhé najväčšie, potom tretie, atď. Na konci postupu tak preložíme na začiatok najmenšie číslo a dostaneme usporiadané pole.

Pri takomto postupe ale robíme niektoré operácie zbytočne. Môžeme si všimnúť, že **najväčšie číslo nikdy prekladať nemusíme**. Ak totiž preložíme ostatných $n-1$ čísel, najväčšie číslo bude na konci poľa – teda tam, kde má byť.

Podobne si môžeme všimnúť, že ak je druhé najväčšie číslo naľavo od najväčšieho, ani toto číslo presúvať nemusíme, lebo keď vykonáme zvyšných $n-2$ operácií (presunieme na začiatok tretie najväčšie, štvrté najväčšie, ..., najmenšie) tak bude pole usporiadané. Túto úvahu môžeme zovšeobecniť.

Predpokladajme, že k najväčších čísel je na vstupe v správnom poradí (k-te najväčšie je naľavo od $(k-1)$-vého najväčšieho, to je naľavo od $(k-2)$-hého, ...). Potom nám stačí postupne presunúť zvyšných $n-k$ čísel, teda urobíme $n-k$ operácií.

Ak je naše k najväčšie možné (teda $(k+1)$-vé najväčšie číslo je už napravo od k-teho), tak sa pole nedá utriediť na menej ako $n-k$ operácií. Niekedy totiž musíme presunúť $(k+1)$-vé najväčšie číslo, aby sme ho dostali naľavo od k-teho. A potom **budeme už musieť preložiť aj všetky menšie čísla** aby sme ich dostali pred neho. To znamená, že každé z $n-k$ najmenších čísel musíme aspoň raz presunúť.

Zoberme si ako príklad postupnosť: $5, 6, 3, 7, 2, 1, 4$. Tri najväčšie čísla sú v správnom poradí (5 je naľavo od 6 a 6 od 7), teda najväčšie možné k je 3. Číslo 4 niekedy určite presunúť musíme, aby sme ho dostali naľavo od 5. Potom musíme na začiatok popresúvať aj všetky menšie čísla: $3, 2, 1$.

Na vyriešenie úlohy nám teda stačí nájsť najväčšie také k, že k najväčších čísel je na vstupe v správnom poradí, a potom vypísať hodnotu $n-k$. Otázkou stále zostáva, ako zistiť hodnotu k.

Riešenie pre čísla 1 až n

V polovici vstupných sád platilo, že na vstupe sa nachádza n-prvkové pole čísel 1 až n. Pri takýchto vstupoch vieme hneď povedať, že najväčšie číslo bude n, druhé najväčšie $n-1$, atď.

Mohli by sme teda pre každé číslo od n po 1 zisťovať, či sa naľavo od neho nenachádza väčšie číslo, čím by sme dostali algoritmus s časovou zložitosťou $O(n^2)$.

Efektívnejšie však bude priamo zostrojiť postupnosť k najväčších čísel. Pole raz prejdeme sprava doľava a postupne budeme hľadať čísla od najväčšieho po najmenšie. Aktuálne hľadané číslo si označíme ako `hladane_cislo`. Najprv hľadáme číslo n. Vždy keď nájdeme `hladane_cislo`, ďalej hľadáme najväčšie menšie číslo, teda `hladane_cislo`

zmenšíme o 1. Takto pokračujeme až kým neprídeme na začiatok poľa. Popritom si počítame, koľkokrát sme našli a zmenšovali `hladane_cislo` – tento počet je naše hľadané k.

Takéto riešenie má časovú aj pamäťovú zložitosť $O(n)$, lebo len raz prejdeme celé pole a v pamäti držíme len jedno pole dĺžky n a zopár jednoduchých premenných.

Listing programu (C++)

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

using namespace std;

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

    int k = 0, hladane_cislo = n;

    for(int i=n-1; i>=0; i--) {
        if(pole[i] == hladane_cislo) {
            hladane_cislo--;
            k++;
        }
    }

    cout << n - k << endl;
    return 0;
}
```

Využitie tej istej myšlienky pre pole ľubovoľných čísel

Predošle riešenie fungovalo len ak pole malo čísla od 1 po n, lebo sme predpokladali, že maximum je n, druhé najväčšie číslo je $n-1$ atď. Ak máme v poli ľubovoľné čísla, nevieme, ktoré je najväčšie, druhé najväčšie atď.

Riešenie pre čísla od 1 po n však vieme jednoducho upraviť. Trik spočíva v tom, že si vytvoríme ešte jedno pole, nakopírujeme doň čísla zo vstupu a usporiadame ho vzostupne. Ak veľmi nepoznáte triediace algoritmy, odporúčame prečítať si o nich v kuchárke a v odkazoch z nej[3]. Pri riešení tejto úlohy nám bude stačiť vedieť, že pole n prvkov vieme efektívne usporiadať v čase $O(n \log n)$.

Pomocou usporiadaného poľa už vieme rýchlo zisťovať, aké je i-te najväčie číslo – maximum je na indexe $n-1$ (ak pole indexujeme od 0), druhé najväčšie na indexe $n-2$, atď. Vieme teda použiť rovnaký algoritmus ako v predošlom riešení s jednou drobnou zmenou: prezeraný prvok poľa neporovnávame s hodnotou v premennej `hladane_cislo` ale s prvkom utriedeného poľa s indexom `index_hladaneho_cisla`. V riešení teda znova prechádzame vstupné pole od konca a ak je prezerané číslo práve to hľadané, zmenšíme ukazovateľ `index_hladaneho_cisla` o 1.

Optimálne triedenie má časovú zložitosť $O(n \log n)$, a po utriedení vykonávame len algoritmus z predošlej časti, ktorý beží v lineárnom čase $O(n)$. Celkovo je teda časová zložitosť $O(n \log n + n) = O(n \log n)$. Pamäťová zložitosť je $O(n)$, lebo potrebujeme 2 polia dĺžky n.

Listing programu (Python)

```python
n = int(input())
A = [int(x) for x in input().split()]

B = sorted(A)

index_hladaneho_cisla = n-1
k = 0

# do 'a' si postupne priradzujme prvky pola a prechadzame pole od konca
for a in A[::-1]:
    if a == B[index_hladaneho_cisla]:
        index_hladaneho_cisla -= 1
        k += 1

print(n - k)
```

Optimálne riešenie

Keďže pri riešení úlohy musíme minimálne prečítať celý vstup, žiadne riešenie nemôže byť rýchlejšie ako $O(n)$. Ak teda vymyslíme algoritmus s lineárnou zložitosťou, budeme si istí, že je najlepší možný (ak porovnávame

[3] https://www.ksp.sk/kucharka/triedenie/

algoritmy asymptotickou časovou zložitosťou[4]).

Predošlé riešenie nám spomaľovalo triedenie, ktoré sme potrebovali na to, aby sme vedeli určiť hľadané i-te najväčšie číslo. Ak mierime na zložitosť $O(n)$, musíme sa triedeniu vyhnúť a ideálne vyriešiť úlohu len jedným prechodom cez pole.

Vráťme sa v úvahách späť k myšlienke "niekde v poli bude usporiadaná podpostupnosť niekoľkých najväčších čísel a práve tieto čísla nemusíme presúvať". V predošlých riešeniach sme najprv zistili, ktoré sú to tie najväčšie čísla a potom sme konkrétnu postupnosť odzadu hľadali vo vstupnom poli. Pokúsme sa teraz začať prechádzať pole od konca, bez toho, aby sme vedeli čo hľadáme. Priebežne si budeme udržovať kandidátsku podpostupnosť – postupnosť najväčších doteraz videných čísel, ktoré boli vzájomne dobre usporiadané. Na uchovávanie kandidátskej podpostupnosti použijeme zásobník[5].

Vstupné pole budeme prechádzať sprava doľava, a postupne si budeme čísla vkladať do zásobníka. Predtým, než vložíme nové číslo x do zásobníka, z jeho vrchu postupne odstránime všetky čísla, ktoré sú menšie ako x. To nám zabezpečí, že v zásobníku budú čísla usporiadané – na spodku bude najväčšie doteraz videné číslo. Keď prejdeme celé pole, tak na spodku bude najväčšie číslo. Druhý prvok odspodu bude najväčšie číslo, ktoré bolo naľavo od maxima. Tretí prvok odspodu bude najväčšie číslo, ktoré bolo ešte viac naľavo, atď.

Zásobník teda bude obsahovať niekoľko najväčších, dobre usporiadaných čísel, no môže obsahovať aj čísla, ktoré musíme presúvať. Pozrime sa na príklad $1, 5, 3, 7, 2, 6, 8, 4$. Po prejdení celého poľa ostane v zásobníku podpostupnosť $1, 5, 7, 8$. Číslo 6 ale musíme pri usporiadavaní preložiť na začiatok a tak budeme musieť preložiť aj 5 a 1, ktoré zostali v zásobníku. Na to, aby sme dostali našu žiadanú postupnosť nehybných čísel, musíme teda zo zásobníka odstrániť všetky čísla, ktoré sú menšie ako najväčšie odstránené číslo.

Môžeme si všimnúť, že čísla odstránené zo zásobníka počas prechodu poľa sú vlastne tie čísla, ktoré bolo nutné presunúť, lebo naľavo od nich sa nachádzalo nejaké väčšie číslo. Čísla odstránené zo zásobníka na záver sú tie čísla, ktoré bolo nutné presunúť, lebo sa pred nich dostalo nejaké väčšie číslo.

Celkový počet potrebných presunov je teda počet čísel odstránených zo zásobníka, alebo jednoducho n mínus počet čísel v zásobníku.

Toto riešenie bude mať časovú zložitosť $O(n)$, lebo prechádzame celé pole len raz a každé číslo do zásobníka vložíme práve raz a odstránime ho najviac raz. Pamäťová zložitosť zostáva $O(n)$, keďže potrebujeme len pole dĺžky n a zásobník dĺžky najviac n. Za toto riešenie ste mohli dostať plný počet bodov.

Listing programu (C++)

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

using namespace std;

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

    int vyhodene_maximum = -1;
    stack<int> zasobnik;

    for(int i=n-1; i>=0; i--) {
        while ((!zasobnik.empty()) && (zasobnik.top() < pole[i])){
            vyhodene_maximum = max(vyhodene_maximum, zasobnik.top());
            zasobnik.pop();
        }
        zasobnik.push(pole[i]);
    }

    while ((!zasobnik.empty()) && (zasobnik.top() < vyhodene_maximum)) {
        zasobnik.pop();
    }
    cout << n - zasobnik.size() <<endl;
    return 0;
}
```

Mišo

4. Ondrej odpúšťa

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

Úlohou bolo čo najrýchlejšie zistiť k-tu cifru v Zajovom zošite. Tam sa jedno za druhým nachádzajú všetky prirodzené čísla, v ktorých sa nenachádza cifra 5, počínajúc jednotkou.

[4] https://www.ksp.sk/kucharka/zlozitost1/
[5] https://www.ksp.sk/kucharka/stack_queue_deque/

Keby sme nemuseli vynechávať tú päťku, tak by to bolo ľahšie, nemyslíte?

Ako sa vysporiadať s päťkou

Vynechávanie päťky nám nerobí až taký veľký problém. V prvom rade si uvedomme, že **vynechávame všetky celé čísla, ktoré obsahujú cifru päť**. Nevynechávame len jednotlivé cifry.

Tým sa dostávame dočinenia s prirodzenými číslami, ktoré sa skladajú z cifier $0, 1, 2, 3, 4, 6, 7, 8, 9$.

Vidíte to? Deväť rôznych cifier. Idú jedna po druhej. Po 2 ide 3, po 4 ide 6.

V skutočnosti jednáme s číslami zapísanými v deviatkovej sústave!

Aby to bolo zrejmejšie, stačí cifry $6, 7, 8, 9$ prepísať na cifry $5, 6, 7, 8$. Tým sa reálne nič nezmení, len sme si premenovali cifry. Teraz už pracujeme s ciframi $0, 1, 2, 3, 4, 5, 6, 7, 8$.

Trošku praktickej ukážky. Poďme zistiť, ktoré číslo (celé číslo, nie len cifra) je 31728-té v poradí v Zajovom zošite. Najprv musíme zistiť, ktoré prirodzené číslo je 31728-té v deviatkovej sústave. To zistíme jednoducho tak, že prepíšeme desiatkové číslo 31728_{10} do deviatkovej sústavy. Dostaneme 47463_9. Teraz sme dostali hľadané číslo, len v skutočnej deviatkovej sústave. Skutočná deviatková sústava v podstate vynecháva cifru 9, my vynechávame cifru 5. Preto musíme cifry väčšie ako 4 zväčšiť o jedna. Hľadaným číslom je teda číslo 48473.

Hrubá sila

Najjednoduchšie riešenie je priamočiare generovanie jednotlivých cifier, až kým nevygenerujeme hľadanú cifru, ktorú následne vypíšeme ako výstup. Postupne ideme cez prirodzené čísla od 1. Každé skonvertujeme do deviatkovej sústavy a dané cifry pripíšeme na koniec poľa, kde si pamätáme všetky cifry v Zajovom zošite v poradí. Cifry väčšie ako 4 zväčšíme o 1. Keď sa už v poli nachádza dostatok cifier, aby sme vedeli zodpovedať otázku na vstupe, prestaneme generovať a vypíšeme odpoveď. Keďže cifry v Zajovom zošite sú vždy také isté, dané pole cifier si môžeme pamätať medzi otázkami, čím si ušetríme kopu času, ktorý by sme zabili generovaním tých istých cifier viac krát.

Aká je časová zložitosť tohto riešenia? Každú cifru musíme osobitne vygenerovať a umiestniť do poľa. Ak chceme povedať ako vyzerá k-ta cifra, musíme vygenerovať všetkých prvých k cifier. Keďže si však pole pamätáme medzi otázkami, každú cifru vygenerujeme len raz. Časová zložitosť je preto $O(n+k)$, kde n je počet otázok a k je maximálna cifra v otázke.

Takáto časová zložitosť je dostatočná pre prvú sadu vstupov. Je však celkom zrejmé, že pri $k = 10^{18}$ nám to už stačiť nebude.

Chytré triky na druhú a tretiu sadu

V zadaní bolo napísané, že druhú a tretiu sadu vstupov sa dalo riešiť nejakými trikmi. A je to pravda!

V druhej sade si musíme všimnúť dôležitú vec: existuje len 257 možných otázok. Môžeme si teda všetky tieto vstupy dopredu predrátať na vlastnom počítači s neobmedzeným časovým limitom. Potom ich napíšeme priamo do zdrojového kódu programu a okamžite ich vypíšeme na požiadanie.

V tretej sade tiež nie je veľa možných vstupov: 10^5. To už je však priveľa na to, aby sme ich písali na tvrdo do zdrojového kódu. Všimnime si však inú vec. Máme dočinenia s ciframi s indexami od 10^9 do $10^9 + 10^5$. *Všetky cifry sa nachádzajú vnútri 9-ciferných čísel v Zajovom zošite.* (Ak neveríte, overte si). Spočítaním počtov cifier v jednotlivých rádoch (jednociferné, dvojciferné, ...) vypočítame, že prvá cifra nachádzajúca sa v prvom 9-cifernom čísle je cifra číslo 338992928.

A teraz prichádza veľká myšlienka. Jednáme iba s ciframi 9-ciferných čísel a vieme, ktorá cifra je prvá cifra prvého 9-ciferného čísla. Teda, keď dostaneme číslo cifry, ktorú máme vypísať, vieme ľahko vypočítať, v ktorom 9-cifernom čísle sa táto cifra nachádza.

Napríklad, povedzme, že máme vypísať cifru číslo 1000008371. Odčítaním $1000008371 - 338992928 = 661015443$ zistíme, koľkatá cifra v 9-ciferných číslach to je. Teraz, keďže všetky 9-ciferné čísla sú rovnako dlhé, celočíselným delením $661015443/9 = 73446160$ vypočítame, v koľkatom 9-cifernom čísle sa táto cifra nachádza. Je to číslo 263173164 (v deviatkovej sústave). Ako sme už raz spravili, keďže my vynechávame cifru 5, popo-súvame všetky cifry väčšie ako 4 o jedna a jedná sa teda o číslo 273183174 v Zajovom zošite. Ďalej, pomocou zvyšku po delení $661015443 \mod 9 = 3$ ľahko zistíme, že hľadaná cifra je štvrtá v danom čísle. Správna odpoveď je teda 1.

Vzorové riešenie

Od triku, ktorý sme použili v tretej sade nám ostáva už len krôčik k vzorovému riešeniu. V tretej sade nám pomohlo, že sme vedeli koľko ciferné je číslo, v ktorom sa nachádza hľadaná cifra. **Nevedeli by sme túto informáciu vypočítať pre ľubovoľnú cifru?** Samozrejme, vedeli.

Stačí, keď si predpočítame indexy prvých cifier v každom ráde: v 1-ciferných, 2-ciferných, 3-ciferných, a tak ďalej, až po 18-ciferné (s väčšími totiž nebudeme mať do činenia). Potom, keď dostaneme index cifry, ktorej hodnotu máme vypísať, stačí zistiť, medzi ktorými dvoma začiatkami sa jej index nachádza a podľa toho zistíme, v koľkocifernom čísle ju hľadať. Napríklad, cifra s indexom 4321 sa nachádza v nejakom 4-cifernom čísle, pretože prvá cifra zo štvorciferného čísla má index 2096 a prvá cifra 5-ciferného čísla má index 25424. Index 4321 sa nachádza medzi týmito dvoma začiatkami, preto vieme, že sa jedná o cifru 4-ciferného čísla.

Keď zistíme, v koľkocifernom čísle sa hľadaná cifra nachádza, použijeme rovnaký postup, ako pri tretej sade, len ho zovšeobecníme na ľubovoľný počet cifier.

Časová zložitosť

Aká je časová zložitosť tohoto riešenia? Pre každú otázku potrebuje spraviť tri veci:

1. Nájsť rád čísla, v ktorom sa nachádza daná cifra.
2. Vypočítať, v ktorom čísle sa daná cifra nachádza.
3. Vypočítať, koľkatá cifra to je, čím už vieme vypočítať konkrétnu cifru.

Časová zložitosť krokov 2. a 3. je jasná: $O(1)$. Koľko nám trvá nájsť rád cifry? Vzhľadom na limity na veľkosť vstupu, počet možných rádov je najviac 18. Pre zjednodušenie môžeme teda povedať, že nájsť rád trvá $O(1)$.

Pokiaľ by však veľkosť čísel na vstupe bola neobmedzená, museli by sme započítať aj nejaké logaritmy navyše. Takéto logaritmy by sme však museli potom započítať aj do krokov 2. a 3.

Celková časová zložitosť je teda $O(n)$.

Listing programu (C++)

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

long long devat_na[18];
long long zaciatok_i_cifernych[19];

void predrataj()
{
    devat_na[0] = 1;
    for(int i=1; i<18; i++)
    {
        devat_na[i] = devat_na[i-1] * 9;
    }

    long long cifier_zatial = 0;
    for(int i=1; i<=18; i++)
    {
        zaciatok_i_cifernych[i] = cifier_zatial;
        long long pocet_i_cifernych = 8 * devat_na[i-1];
        cifier_zatial += pocet_i_cifernych * i;
    }
}

vector<int> do_deviatkovej_sustavy(long long x)
{
    vector<int> vysledok;
    while(x > 0)
    {
        vysledok.push_back(x % 9);
        x /= 9;
    }
    reverse(vysledok.begin(), vysledok.end());
    return vysledok;
}

int main()
{
    predrataj();

    int n;
    cin >> n;
    for(int test = 0; test < n; test++)
    {
        long long k;
        cin >> k;
        int kolko_ciferne;
        for(int i=18; i>=1; i--)
        {
            if(zaciatok_i_cifernych[i] <= k)
            {
                kolko_ciferne = i;
                break;
            }
        }
        long long kolke_i_ciferne = (k - zaciatok_i_cifernych[kolko_ciferne]) / kolko_ciferne;
        long long ake_cislo = devat_na[kolko_ciferne-1] + kolke_i_ciferne;

        vector<int> v_deviatkovej = do_deviatkovej_sustavy(ake_cislo);
        int kolka_cifra = (k - zaciatok_i_cifernych[kolko_ciferne]) % kolko_ciferne;
```

```
        if(v_deviatkovej[kolka_cifra] >= 5)
        {
            printf("%d\n", v_deviatkovej[kolka_cifra]+1);
        }
        else
        {
            printf("%d\n", v_deviatkovej[kolka_cifra]);
        }
    }
    return 0;
}
```

Buj

5. Lenivé prasa

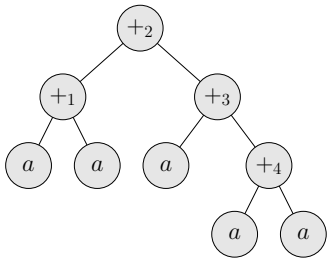
(max. 12 b za popis, 8 b za program)

Reprezentácia výrazu

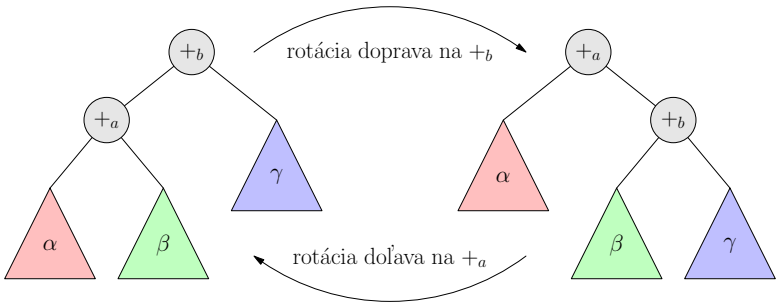
S výrazom chceme vedieť robiť rotácie. Ak by sme ale pracovali s jeho textovou reprezentáciou, tak by sme zistili, že je to dosť nešikovné.

Zamyslime sa nad tým, čo to je kompletne uzátvorkovaný výraz. Je to buď a, alebo $(A + B)$ pre nejaké kompletne uzátvorkované podvýrazy A, B. Teda ak sme kompletne uzátvorkovaný výraz, musíme si pamätať, či sme prvého alebo druhého typu. A ak sme druhého typu, tak si musíme tiež pamätať ľavý podvýraz a pravý podvýraz. Z tejto úvahy vidno, že sa na výrazy dá pozerať ako na binárne stromy: koreň reprezentuje celý výraz, ľavý syn reprezentuje ľavý podvýraz a pravý syn reprezentuje pravý podvýraz.

Napríklad výrazu $(a + a) + (a + (a + a))$ prislúcha nasledujúci strom:



Na strome vieme rotáciu implementovať jednoducho: vrcholy pospájame hranami trochu iným spôsobom. Na obrázku nižšie ilustrujeme rotácie v oboch smeroch.



Listing programu (C++)

```
struct Vrchol {
    int id = -1;
    Vrchol* synovia[2] = {NULL, NULL}; // 0 lavy, 1 pravy
    Vrchol* otec = NULL;

    int smerKu(Vrchol* syn) {
        // vrati smer k zadanemu synovi
        if (synovia[0] == syn) return 0;
        if (synovia[1] == syn) return 1;
    }
    bool jeList() {
        return (synovia[0] == NULL) && (synovia[1] == NULL);
    }
    void nastavSyna(Vrchol* novy, bool smer) {
        // uprav sebe linku na syna, a novemu synovi linku na otca
        novy->otec = this;
        synovia[smer] = novy;
    }
    void zrotuj(bool smer) {
        // zrotuje okolo tohto vrcholu, 0 dolava, 1 doprava
        Vrchol* povodnyOtec = otec;
        Vrchol* novyOtec = synovia[!smer];
```

```
        Vrchol* novySyn = novyOtec->synovia[smer];
        nastavSyna(novySyn, !smer);
        novyOtec->nastavSyna(this, smer);
        if (povodnyOtec != NULL) {
            povodnyOtec->nastavSyna(novyOtec, povodnyOtec->smerKu(this));
        }
        else {
            novyOtec->otec = NULL;
        }
    }
};
```

Ako ale z textovej reprezentácie dostaneme reprezentáciu stromovú? Tejto otázke je venovaná nasledujúca časť.

Parsovanie

Kedže ťažiskom úlohy nie je parsovanie, uvedieme tu iba rýchly algoritmus, ktorý nám z textovej reprezentácie vyrobí strom v čase $O(n)$.

Myšlienka je taká, že budeme strom vyrábať od koreňa k listom, rekurzívne. Predstavme si, že nám zo vstupu postupne prichádzajú znaky výrazu, a my si chceme postupne vybudovať jeho strom. Máme niekoľko možností:

- Buď je prvý znak výrazu `a`, v takom prípade je náš výraz určite a. Jemu prislúchajúci strom pozostáva iba z jedného vrcholu.
- Alebo je prvý znak `(`. Potom je výraz je zložený, teda má tvar $(A + B)$ pre nejaké podvýrazy A a B. Uvedomme si, že ďalej na vstupe dostaneme podvýraz A, potom `+`, potom podvýraz B a nakoniec `)`. Stačí nám preto rekurzívne vybudovať strom pre A (pričom ho celý prečítame), potom prečítame `+`, rekurzívne zostrojíme strom pre B, a nakoniec prečítame `)`. Stromy zavesíme pod spoločný koreň reprezentujúci prečítané `+`.

Listing programu (C++)

```
using namespace std;

Vrchol* parsuj(string& vyraz, int& pozicia, int& volneId) {
    if (vyraz[pozicia] == 'a') { // podvyraz 'a'
        pozicia++;
        return new Vrchol;
    }
    // zostrojime stromy pre lavy a pravy operand
    pozicia++; // precitame '('
    Vrchol* lavy = parsuj(vyraz, pozicia, volneId);
    int id = volneId;
    volneId++;
    pozicia++; // precitame '+'
    Vrchol* pravy = parsuj(vyraz, pozicia, volneId);
    pozicia++; // precitame ')'

    // vyrob vrchol, ktoreho podstrom reprezentuje tento podvyraz
    Vrchol* ja = new Vrchol;
    ja->id = id;
    ja->nastavSyna(lavy, 0);
    ja->nastavSyna(pravy, 1);
    return ja;
}

Vrchol* parsuj(string& vyraz) {
    int pozicia = 0, volneId = 0;
    return parsuj(vyraz, pozicia, volneId);
}
```

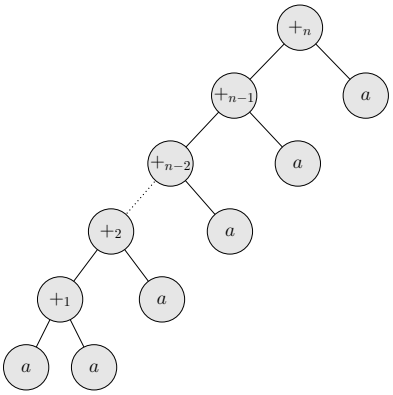
Každý znak výrazu prečítame iba raz, preto je časová zložitosť algoritmu lineárna od dĺžky výrazu, teda $O(n)$. Pamäťová zložitosť je $O(n)$.

Rotovanie

Všetky naše algoritmy, pomalé či rýchle, budú založené na nasledovnom pozorovaní: keď na strom použijeme rotáciu, vhodnou rotáciou sa vieme vrátiť k pôvodnému stromu (viď obrázok vyššie).

Nemusíme teda priamo upravovať výraz A na B, stačí nám oba výrazy upraviť na ten istý tvar C (tvar C budeme volať *normálny tvar*). Na úpravu B na C sa potom pozrieme odzadu, čím dostaneme úpravu C na B. Spolu s úpravou A na C nám to dokopy dá hľadanú úpravu A na B.

Tvar C chceme z výrazov určit jednoznačne. Je prirodzené za tento *normálny tvar* zvoliť *najľavejšie uzátvorkovanie* výrazu, teda $((((\ldots + a) + a) + a) + a)$. Tomu zodpovedá nasledovný strom:



(Rovnako dobre by sme ale mohli použiť *najpravejšie uzátvorkovanie*, teda $(a + (a + (a + (a + \ldots))))$.)

Pomalé rotovanie

Chceme upraviť výraz $(A + B)$ na normálny tvar. Ťažko sa ale rozmýšľa nad prípadom, keď A a B môžu byť ľubovoľné. Ak by A a B mali nejaký konkrétny tvar, tak by sa nám s nimi možno ľahšie pracovalo. Už sa tu spomínali normálne tvary, ak by sme uvažovali tie, tak by sme mali navyše tú výhodu, že A a B by sme do nich vedeli dostať rekurzívne.

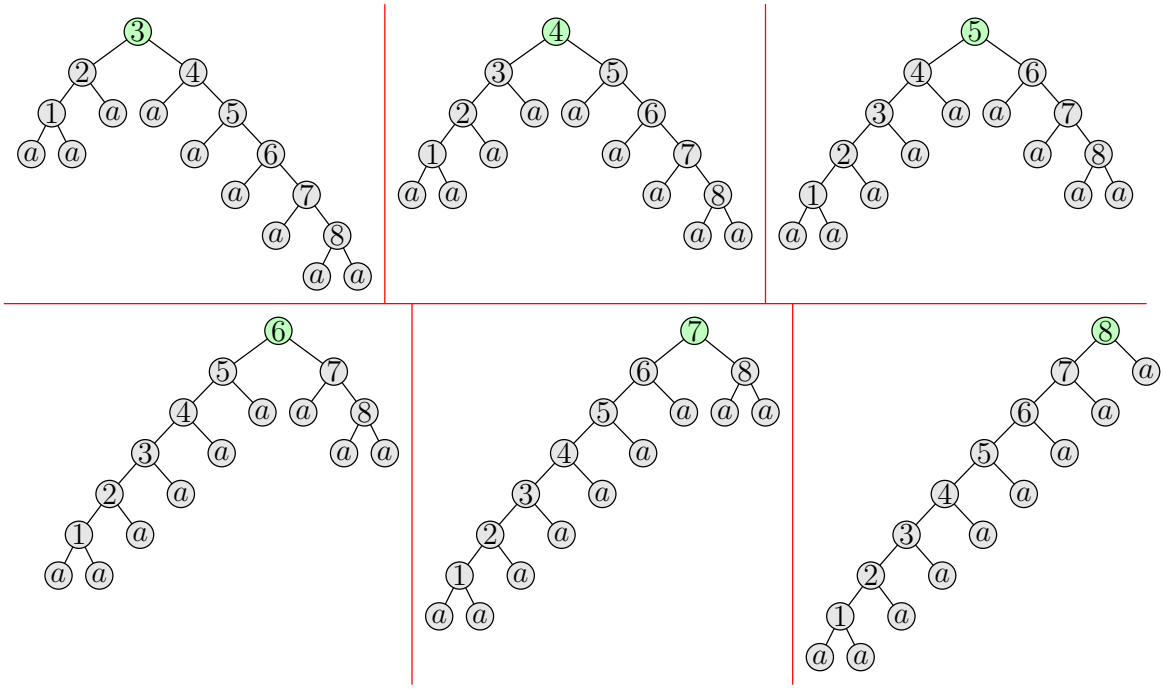
Predstavme si preto, že by A bol v najľavejšom uzátvorkovaní a B v najpravejšom. (Ako uvidíme, takáto voľba bude viesť k tomu, že veci budú vychádzať pekne.) Teda celý výraz vyzerá nasledovne:

$$(\overbrace{(((((\ldots + a) + a) + a) + a)}^{l}) +_l (\overbrace{a + (a + (a + (a + \ldots)))}^{r})$$

kde l označuje počet a-čok v A a r označuje počet a-čok v B. Čo sa s výrazom stane, ak by sme spravili rotáciu doľava na $+$ spájajúcom A a B? Dostali by sme

$$(\overbrace{(((((\ldots + a) + a) + a) + a)}^{l+1}) + (\overbrace{a + (a + (a + (a + \ldots)))}^{r-1})$$

Teda vhodným počtom rotácii doľava vieme náš výraz dostať do najľavejšieho uzátvorkovania. Podobne vhodným počtom rotácii doprava vieme výraz dostať do najpravejšieho uzátvorkovania. Čo je presne to, čo chceme docieliť.



Predpokladali sme ale, že podvýraz A je v najľavejšom uzátvorkovaní a B v najpravejšom. To vo väčšine prípadov neplatí, čo s tým? Stačí ich na začiatku rekurzívne upraviť na požadovaný tvar.

Listing programu (C++)

```
#include <iostream>
#include <queue>          // deque
#include "tree.cpp"       // reprezentacia vyrazu
#include "parse.cpp"      // parsovanie
using namespace std;

void doNormalnehoTvaru(Vrchol* koren, deque<pair<int, char> >& rotacie, bool inverz, bool smer) {
    // Da strom do najlavejsieho/najpravejsieho uzatvorkovania (podla
    // <smer>), a postupnost rotacii prida do <rotacie>. Ak
    // <inverz> == true, tak tam prida namiesto toho inverznu postupnost.
    if (koren->jeList()) {
        return;
    }
    doNormalnehoTvaru(koren->synovia[0], rotacie, inverz, 0);
    doNormalnehoTvaru(koren->synovia[1], rotacie, inverz, 1);
    while (!koren->synovia[!smer]->jeList()) {
        koren->zrotuj(smer);
        if (inverz) {
            rotacie.push_front({koren->otec->id, (smer ? 'L' : 'R')});
        }
        else {
            rotacie.push_back({koren->id, (smer ? 'R' : 'L')});
        }
        koren = koren->otec;
    }
}

void vyries(Vrchol* strom1, Vrchol* strom2, deque<pair<int, char> >& rotacie) {
    // Do <rotacie> ulozi postupnost rotacii, ktora upravi <strom1>
    // na <strom2>.
    doNormalnehoTvaru(strom1, rotacie, false, 0);
    deque<pair<int, char> > rotacie2;
    doNormalnehoTvaru(strom2, rotacie2, true, 0);
    for (int i = 0; i < (int)rotacie2.size(); i++) {
        rotacie.push_back(rotacie2[i]);
    }
}

void pridajVonkajsieZatvorky(string& vyraz) {
    if ((int)vyraz.size() > 1) {
        vyraz = '(' + vyraz + ')';
    }
}

int main() {
    int t;
    cin >> t;
    for (int test = 0; test < t; test++) {
        // nacitame vyrazy a vyrobime stromove reprezentacie
        string vyraz1, vyraz2;
        cin >> vyraz1 >> vyraz2;
        pridajVonkajsieZatvorky(vyraz1);
        pridajVonkajsieZatvorky(vyraz2);
        Vrchol* strom1 = parsuj(vyraz1);
        Vrchol* strom2 = parsuj(vyraz2);

        // vyriesime a vypiseme
        deque<pair<int, char> > rotacie;
        vyries(strom1, strom2, rotacie);
        cout << rotacie.size() << "\n";
        for (int i = 0; i < (int)rotacie.size(); i++) {
            cout << rotacie[i].first + 1 << " " << rotacie[i].second << "\n";
        }
    }
    return 0;
}
```

Naša rekurzívna funkcia sa zavolá do každého vrcholu stromu práve raz, pričom v každom volaní urobíme nanajvýš $O(n)$ roboty. To znamená, že určite nerobíme dokopy viac ako $O(n^2)$ roboty. Ukazuje sa, že na najhorších vstupoch[6] naozaj urobíme až kvadraticky veľa práce, teda časová zložitosť je $\Theta(n^2)$. Pamäťová zložitosť je $O(n)$, nakoľko pracujeme iba so stromom výrazu.

Rýchle rotovanie

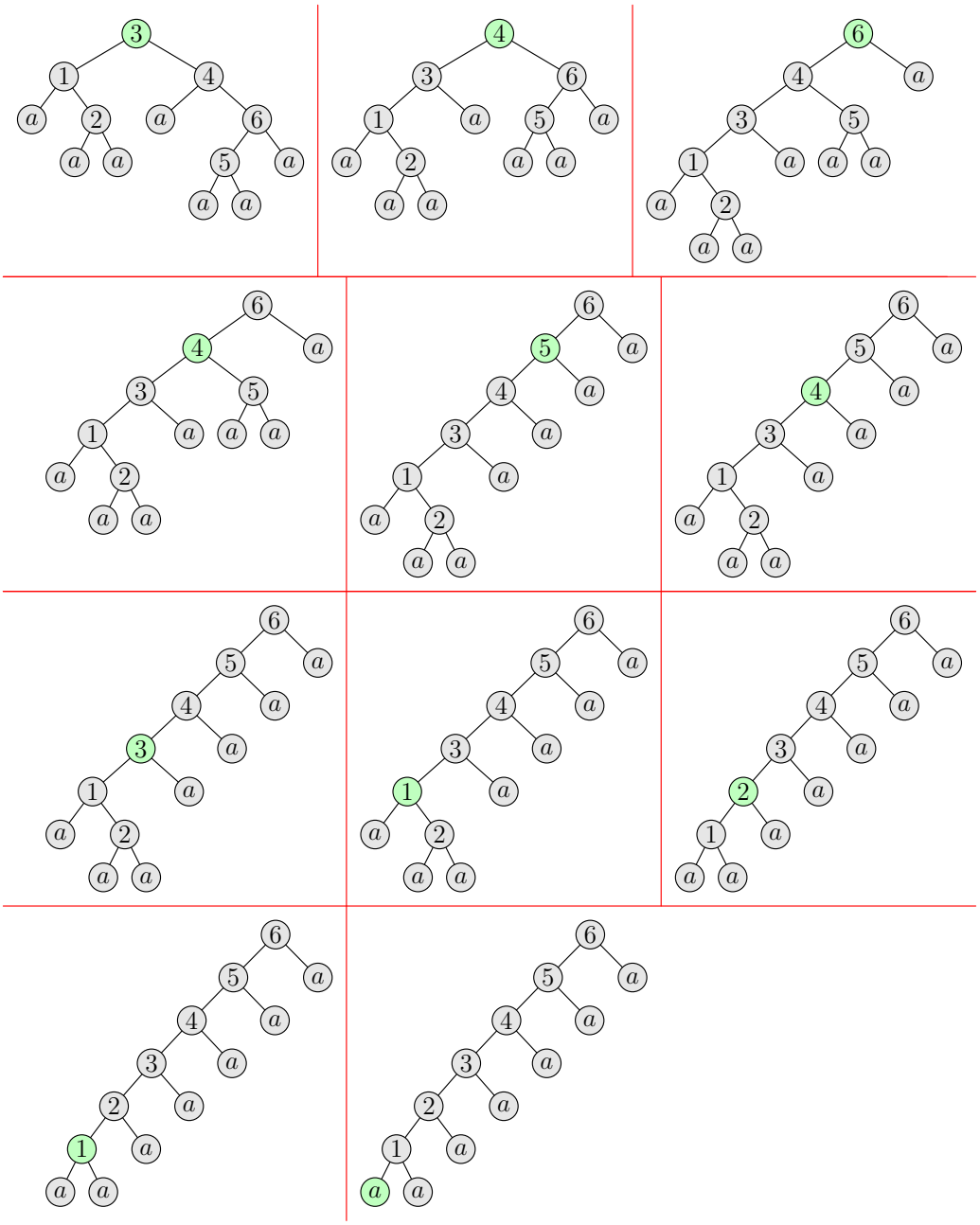
Najprv algoritmus popíšeme a potom ukážeme, prečo je rýchly.

Ak má strom iba jeden vrchol, tak už je v najľavejšom uzátvorkovaní (reprezentuje výraz a). V opačnom prípade je koreň +, má ľavého aj pravého syna. Vtedy urobíme nasledovné:

1. Kým platí, že pravý syn koreňa nie je list (pravý podstrom koreňa obsahuje aspoň dva vrcholy), robíme rotácie doľava okolo aktuálneho koreňa. Pri každej takejto rotácii sa nám zmení koreň, pričom nový koreň bude mať pravý podstrom o niečo menší, než mal ten starý. Po konečnom počte krokov teda prídeme k stromu, kde je pravý syn koreňa list (zodpovedajúci výraz má tvar $(A + a)$).

2. Ľavý podstrom koreňa rekurzívne upravíme na najľavejšie uzátvorkovanie.

[6] Príklad zlého vstupu je "cik-cakový" strom, ktorý dostaneme z výrazu $(a + \ldots((a + ((a + (a + a)) + a)) + a)) + a)) \cdots + a))$.



Listing programu (C++)

```
#include <iostream>
#include <queue>          // deque
#include "tree.cpp"       // reprezentacia vyrazu
#include "parse.cpp"      // parsovanie
using namespace std;

void doLavehoTvaru(Vrchol* koren, deque<pair<int, char> >& rotacie, bool inverz) {
    // Da strom do najlavejsieho uzatvorkovania, a postupnost rotacii
    // prida do <rotacie>. Ak <inverz> == true, tak tam prida
    // namiesto toho inverznu postupnost.
    Vrchol* pivot = koren;
    while (!pivot->jeList()) {
        if (!pivot->synovia[1]->jeList()) {
            pivot->zrotuj(0);
            if (inverz) {
                rotacie.push_front({pivot->otec->id, 'R'});
            }
            else {
                rotacie.push_back({pivot->id, 'L'});
            }
            pivot = pivot->otec;
        }
        else {
            pivot = pivot->synovia[0];
        }
```

```
    }
}

void vyries(Vrchol* strom1, Vrchol* strom2, deque<pair<int, char> >& rotacie) {
    // Do <rotacie> ulozi postupnost rotacii, ktora upravi <strom1>
    // na <strom2>.
    doLavehoTvaru(strom1, rotacie, false);
    deque<pair<int, char> > rotacie2;
    doLavehoTvaru(strom2, rotacie2, true);
    for (int i = 0; i < (int)rotacie2.size(); i++) {
        rotacie.push_back(rotacie2[i]);
    }
}

void pridajVonkajsieZatvorky(string& vyraz) {
    if ((int)vyraz.size() > 1) {
        vyraz = '(' + vyraz + ')';
    }
}

int main() {
    int t;
    cin >> t;
    for (int test = 0; test < t; test++) {
        // nacitame vyrazy a vyrobime stromove reprezentacie
        string vyraz1, vyraz2;
        cin >> vyraz1 >> vyraz2;
        pridajVonkajsieZatvorky(vyraz1);
        pridajVonkajsieZatvorky(vyraz2);
        Vrchol* strom1 = parsuj(vyraz1);
        Vrchol* strom2 = parsuj(vyraz2);

        // vyriesime a vypiseme
        deque<pair<int, char> > rotacie;
        vyries(strom1, strom2, rotacie);
        cout << rotacie.size() << "\n";
        for (int i = 0; i < (int)rotacie.size(); i++) {
            cout << rotacie[i].first + 1 << " " << rotacie[i].second << "\n";
        }
    }
    return 0;
}
```

Prečo je tento algoritmus rýchly? Pozerajme sa v priebehu algoritmu na *ľavú diagonálu* – tak budeme volať množinu vrcholov, ktoré vieme dosiahnuť z koreňa tak, že sa niekoľkokrát presunieme po hrane do ľavého syna.

Vrchol, okolo ktorého rotujeme (teda koreň podstromu spracovávaného v aktuálnom volaní našej rekurzívnej funkcie), leží vždy na tejto diagonále. Pri každej rotácii sa počet vrcholov na tejto diagonále zväčší o 1 (rozmyslite si). Počet vrcholov na diagonále nemôže byť väčší, ako počet vrcholov stromu n, teda spravíme nanajvýš n rotácii. Časová zložitosť je preto $O(n)$. Pamäťová zložitosť je $O(n)$.

Žaba

6. Ázijec a továreň na čokoládu

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

Prvý krok k riešeniu je vždy poriadne si prečítať zadanie a zistiť, čo od nás vlastne chce. V tomto prípade je tento krok obzvlášť dôležitý, lebo zadanie je pomerne komplikované. Skúsme si teda spoločne zopakovať, čo je našou úlohou.

Na vstupe máme orientovaný graf, ktorého každá hrana má kladnú váhu w_i. Postupne dostaneme q otázok, každá má tvar: Aká je najkratšia cesta medzi vrcholmi a_i a b_i, ktorá ide cez najviac c_i hrán a váhy týchto hran navyše celú dobu stúpajú?

Hľadanie cesty s obmedzeným počtom hrán

Keďže úloha obsahuje pomerne veľké množstvo parametrov, na ktoré si musíme dávať pozor, skúsme si ju zjednodušiť. Napríklad, pre začiatok nemusíme uvažovať o tom, že hrany na výslednej ceste musia rásť. A takisto sa nezaoberajme tým, že máme viacero otázok. Ak dokážeme úlohu vyriešiť pre jednu otázku, prinajhoršom tento postup q krát zopakujeme.

Nová úloha, ktorú riešime preto znie: **Nájdite dĺžku najkratšej cesty medzi vrcholmi x a y, ktorá neobsahuje viac ako c hrán.**

Takéto zadanie vyzerá oveľa jednoduchšie. Máme predsa presne daný začiatok aj koniec a poznáme aj počet hrán na ceste, ktorú hľadáme. Mohli by sme sa teda pokúsiť siahnuť po niektorom z klasických algoritmov na hľadanie najkratších ciest – Dijkstrov algoritmus, poprípade prehľadávanie do šírky. Obom však niečo chýba, Dijkstra sa nepozerá na počet hrán v ceste a prehľadávanie do šírky zase nepracuje s váhami hrán. Navyše, keď sa pozrieme do zadania, vidíme, že hodnoty n a m sú pomerne malé. Možno teda vôbec nepotrebujeme tak rýchle riešenie.

Pozrime sa na to inak. Čo by sa stalo, keby sa $c = 0$? Úloha by bola zrazu veľmi jednoduchá. Ak môžeme prejsť po 0 hranách, z vrchola x sa dostaneme akurát do vrchola x tým, že sa nepohneme. A keby bolo $c = 1$?

V tom prípade by sme z x mohli prejsť po jednej hrane. Vedeli by sme sa teda pozrieť na všetky hrany, ktoré z x vedú a to by nám ukázalo vrcholy, do ktorých sa vieme dostať. Naviac, ak by do jedného vrchola viedlo z x viacero hrán, vybrali by sme tú kratšiu.

A čo s $c = 2$? V tom prípade by sme sa museli pozrieť na vrcholy, do ktorých sme sa vedeli dostať z x na jednu hranu a pridať hranu ďalšiu. Opäť by sme teda získali všetky vrcholy, do ktorých sa vieme dostať z x s pomocou 2 hrán. Naviac vieme zistiť aj najkratšiu dĺžku ciest do týchto vrcholov.

V tomto momente by malo byť jasné, že tento postup vieme zovšeobecniť. Ak poznáme najkratšie cesty s k hranami, ktoré vedú z vrchola x, tak ich vieme použiť na vypočítanie najkratších ciest s $k + 1$ hranami.

Pozrime sa na to ešte trochu bližšie. Majme dvojrozmerné pole `dlzka[][]`, pričom na indexe `dlzka[p][r]` si pamätáme **dĺžku najkratšej cesty vedúcej z vrchola x do vrchola p, ktorá prejde cez práve r hrán**. Ostáva už len vypočítať túto hodnotu. Zoberme si nejakú hranu, ktorá vedie z vrchola p' do vrchola p a má dĺžku w. Potom môžeme povedať, že `dlzka[p][r] = w + dlzka[p'][r-1]`. Teda ak k dĺžke najkratšej cesty z vrchola x do vrchola p', ktorá vedie cez $r - 1$ hrán pripočítame ďalšiu hranu, ktorá z p' vedie do vrchola p a má dĺžku w, dostaneme cestu z x do p.

Táto cesta síce nemusí byť najkratšia, ak ale zoberieme všetky hrany, ktoré vedú do vrchola p a zoberieme minimum z týchto hodnôt, určite nájdeme najkratšiu cestu z x do p, ktorá ide cez r hrán.

Takéto riešenie vieme dokonca veľmi jednoducho naprogramovať. Stačia nám na to dva `for`-cykly. Vonkajší bude určovať, koľko hrán chceme použiť – teda najskôr 1, potom 2, 3... a vnútorný cyklus bude prechádzať cez všetky hrany a postupne ich skúsi pridať k cestám o jedno kratším.

Listing programu (C++)

```
struct hrana {
    int x, y;  // hrana vedie z vrcholu x do vrcholu y
    int w;     // hrana ma vahu w
};

vector<hrana> hrany;
int x;             // zaciatocny vrchol
int dlzka[n][m];   // najkratsia cesta z vrchola x do vrchola n s m hranami
                   // na zaciatku vyplnena dostatocne velkymi hodnotami

for(int r = 1; r <= m; r++) {
    for(int j = 0; j < hrany.size(); j++) {
        int p1 = hrany[j].x, p = hrany[j].y;
        int w = hrany[j].w;
        dlzka[p][r] = min(dlzka[p][r], w + dlzka[p1][r-1])
    }
}
```

V tejto časti si môžeme položiť ešte jednu otázku. Vieme, že hodnota $c \leq m$. Môžeme mať však najkratšiu cestu, ktorá prechádza cez m hrán? Odpoveď je, že nie. Dokonca nemôžeme mať najkratšiu cestu, ktorá prechádza cez viac ako $n - 1$ hrán. Uvedomme si totiž, že v najkratšej ceste sa nám nikdy neoplatí vrátiť do toho istého vrchola. Ak by sme to spravili, tak by bolo predsa lepšie vynechať tú časť cesty, v ktorej sme sa iba vracali do už predtým navštíveného vrchola a radšej pokračovať ďalej. Každá hrana najkratšej cesty teda musí ísť do nového, ešte nenavštíveného vrchola, a teda môže mať dĺžku najviac $n - 1$. Časová zložitosť nášho programu je teda zatiaľ $O(nm)$.

Riešenie viacerých otázok

Vyriešili sme prvú časť úlohy, je na čase pridať si späť podmienky, ktoré sme odstránili. Začnime tým, že musíme vyriešiť q otázok. Ako sme si už spomínali vyššie, najľahší spôsob je proste zopakovať vyššie uvedený proces q krát. Otázok však môže byť až 1000, čo je pomerne dosť. Naviac si uvedomme, že vyššie uvedený spôsob počítal viac ako len najkratšiu cestu z x do y.

Vyššie uvedenný program spočítal pre začiatočný vrchol x najkratšie cesty všetkých dĺžok do všetkých zvyšných vrcholov a uložil ich do poľa `dlzka[][]`. Ak teda dostaneme dve otázky, ktoré majú rovnaký začiatočný vrchol, neoplatí sa nám toto pole počítať od začiatku, lebo výsledok bude ten istý. Iba sa budeme musieť pozrieť do iných políčok tohto poľa.

Naviac, vrcholov je iba 150, takže túto tabuľku budeme musieť prepočítavať najviac 150 krát. Aj to je však mierne ošemetné. Predsa len si musíme otázky rozdeliť podľa počiatočného vrchola a potom to vo vhodných momentoch prepočítať. Na to sme príliš leniví. Pridajme nášmu poľu teda ešte jeden rozmer. Hodnota `dlzka[x][y][r]` bude teda dĺžka najkratšej cesty z vrchola x do vrchola y, ktorá prejde cez práve r hrán.

Takúto tabuľku si vieme spočítať dopredu v čase $O(m \cdot n^2)$. Ako z nej však zistíme odpoveď na zadanú otázku? Otázky, ktoré dostávame sa pýtajú na cesty **s najviac c_i hranami**. Pre zadané a_i a b_i sa teda musíme pozrieť na všetky hodnoty $dlzka[a_i][b_i][j]$ pre $0 \leq j \leq c_i$ a vybrať z nich tú najmenšiu. Odpovedanie na jednu otázku by nám teda trvalo čas $O(n)$.

Vieme to však spraviť aj lepšie. Uvedomme si, že na začiatku si vypočítame celé pole `dlzka[][][]`. No a správna odpoveď pre a_i, b_i a c_i je vždy minimom z hodnôt $dlzka[a_i][b_i][j]$ $(0 \leq j \leq c_i)$. Môžeme si tieto minimá preto pre každú dvojicu vrcholov (a, b) vypočítať dopredu. Stojí nás to síce čas $O(n^3)$, ten sa však ľahko schová do časovej zložitosti $O(m \cdot n^2)$[7], ktorou počítame pole `dlzka[][][]`.

Odpoveď na každú otázku vieme teda získať v čase $O(1)$ jedným pozretím do tabuľky.

Rast váhy hrany na ceste

Ostáva nám vyriešiť poslednú podmienku zadania. Hrany v našich cestách musia postupne rásť.

Pozrime sa najskôr na to, ako funguje naše doterajšie riešenie. Pre zvolený začiatok cesty sa postupne pokúša pridať ďalšie a ďalšie hrany, pričom skúša pridať všetky možné hrany. Vo vzorovom riešení však nemôžeme skúšať všetky hrany. V okamihu ako na nejakej ceste použijeme hranu s váhou w, nemôžeme k tejto ceste pripájať ľahšie hrany. Opačne, ak chceme použiť nejakú hranu s váhou w, môžeme túto hranu pripojiť iba na cestu, ktorá sa skladá z iba ľahších hrán.

Náš algoritmus by sme teda chceli upraviť tak, aby sme v momente, keď pridávame hranu s váhou w, už mali vypočítané všetky cesty z kratších hrán. Naviac aj tieto cesty musia spĺňať podmienku o raste váh hrán. To ale znamená, že hrany musíme spracovávať v poradí od najľahšej po najťažšiu.

Celé riešenie teda vyzerá nasledovne. Usporiadame si hrany podľa váhy. Následne v tomto poradí hrany spracovávame. V každom kroku vyskúšame aktuálnu hranu pripojiť na každú možnú cestu, ktorú zatiaľ máme. Tieto cesty sa pritom dajú reprezentovať pomocou začiatočného vrchola x, koncového vrchola y a počtu hrán, ktoré obsahujú. V okamihu, keď sme túto hranu vyskúšali pridať na každé miesto, pokračujeme s ďalšou hranou a k tejto sa už nevraciame.

Uvedomme si, že v okamihu keď začneme spracovávať nejakú hranu, máme už vypočítané všetky cesty skladajúce sa z ľahších hrán. Neskôr túto hranu už použiť nemusíme (a nemôžeme), každá nová cesta totiž musela vzniknúť pridaním ťažšej hrany a k tej ju nemôžeme pridať.

Najkrajšie je, že naše riešenie sa ani veľmi nezmení. Jediné čo musíme spraviť je, že presunieme cyklus, ktorý prechádza všetkými hranami, na najvyššiu úroveň. To bude totiž predstavovať to, že hrany spracovávame jednu po druhej.

Časová zložitosť nášho riešenia je $O(m \log m + m \cdot n^2 + q)$. Pamäťová zložitosť je $O(n^3)$, keďže si musíme pamätať pole `dlzka[][][]`.

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++)
#define mp(a,b) make_pair((a),(b))
typedef long long ll;
typedef pair<int,int> pii;

struct hrana {
    int x,y;
    int w;
};

bool cond(hrana a, hrana b) {
    if(a.w < b.w) return true;
    return false;
}

int n,m,q;
int dlzka[160][160][160];
const int INF = 1000000000;

int main() {
    scanf("%d %d %d",&n,&m,&q);
    For(i,160) For(j,160) For(k,160) dlzka[i][j][k]=INF;
    For(i,n) dlzka[i][i][0]=0; // na 0 krokov sa vieme dostat do toho isteho vrchola
    vector<hrana> hrany(m);
    For(i,m) {
        scanf("%d %d %d",&hrany[i].x, &hrany[i].y, &hrany[i].w);
        hrany[i].x--; hrany[i].y--;
    }
    sort(hrany.begin(), hrany.end(), cond);
    For(i, hrany.size()) {
        int x = hrany[i].x, y = hrany[i].y, w = hrany[i].w;
        For(j,n) for(int k = 1; k<n; k++) {
            dlzka[j][y][k] = min(dlzka[j][y][k], w + dlzka[j][x][k-1]);
        }
    }
```

[7] Samozrejme, musíme predpokladať, že $m \geq n$. Takýto predpoklad je však rozumný, pretože inak by sme nemali súvislý graf a náš algoritmus by sme vedeli zopakovať na každom komponente zvlášť.

```
    // upravim pole dlzka tak, aby obsahovalo vysledky
    For(i,n) For(j,n) for(int k = 1; k < n; k++) {
        dlzka[i][j][k] = min(dlzka[i][j][k], dlzka[i][j][k-1]);
    }
    // odpoviem na otazky
    For(i,q) {
        int x,y,c;
        scanf("%d %d %d",&x,&y,&c);
        x--; y--;
        c = min(c, n-1);
        if(dlzka[x][y][c] == INF) printf("-1\n");
        else printf("%d\n",dlzka[x][y][c]);
    }
}
```

Hodobox a Jano
(max. 12 b za popis, 8 b za program)

7. Dobrý recept

Stará dobrá hrubá sila

Ako inak – úloha sa dá riešiť aj hrubou silou. Chceme vedieť koľko receptov čokolád má rovnaký rozdiel chutí od tých troch na vstupe. Keďže každý recept je binárny reťazec dĺžky n, možných receptov je 2^n. Pre daný recept vieme v lineárnom čase spočítať chuťový rozdiel od každej čokolády na vstupe a keď nám tento rozdiel vyjde rovnaký pre všetky tri čokolády, pripočítame k odpovedi 1.

Ostáva nám už len vygenerovať všetky binárne reťazce dĺžky n čo najpohodlnejšie – napríklad použiť bitovú reprezentáciu čísel; prejdeme všetky čísla od 0 po $2^n - 1$, pričom recept bude posledných n bitov tohto čísla.

Stačí si nám pamätať reťazce zo vstupu a pár konštantných premenných, pamäťová zložitosť teda bude $O(n)$. Ako sme už spomínali, skúšame 2^n receptov a pre každý spočítame jeho rozdiel chutí od troch čokolád v lineárnom čase, teda časová zložitosť je $O(n2^n)$.

Listing programu (C++)

```
#include <iostream>
using namespace std;

int main() {
    int n, ans = 0;
    cin >> n;
    string cokolady[3];
    for (int i=0; i<3; ++i) cin >> cokolady[i];
    for (int mask = 0; mask < (1<<n); ++mask) {
        int rozdiel[3] = {0,0,0};
        for (int i=0;i<3;++i)
            for (int k=0;k<n;++k)
                if (cokolady[i][k]-'0' != ((mask>>k)&1))
                    ++rozdiel[i];
        if (rozdiel[0]==rozdiel[1] && rozdiel[1]==rozdiel[2])
            ++ans;
    }
    cout << ans % 1000000007 << "\n";
}
```

Dynamické programovanie

Úlohu si môžeme zovšeobecniť. Namiesto toho, aby sme hľadali počet receptov, ktoré sa od každej čokolády líšia rovnako, pre každú trojicu a, b, c spočítame koľko receptov sa líši od prvej čokolády na a pozíciách, od druhej na b pozíciách a od tretej na c pozíciách. Navyše tieto hodnoty postupne spočítame pre n prípadov: čo keby sme uvažovali len recepty dĺžky i a porovnávali ich len s prvými i ingredienciami? Počet reťazcov dĺžky i, ktoré sa od reťazcov na vstupe líšia postupne na a,b,c z prvých i pozícií, označíme $P[i, a, b, c]$.

Vyzerá to, že sme si veľmi nepomohli, lebo teraz musíme spočítať $O(n^4)$ hodnôt namiesto jednej. No opak je pravdou, pretože tieto hodnoty ľahko spočítame pomocou techniky *dynamického programovania*.

Pre $i = 0$ máme len jednu možnosť, ako môže reťazec vyzerať, a tento reťazec sa na žiadnej z 0 pozícií od žiadnej z troch čokolád chuťovo nelíši. Takže $P[0, 0, 0, 0] = 1$ a $P[0, a, b, c] = 0$ pre ostatné hodnoty a, b, c.

Teraz si ukážeme, ako spočítať $P[i, a, b, c]$ pre $i > 0$, ak už poznáme hodnoty $P[\,]$ pre $i - 1$. Pozrime sa na i tu ingredicenciu. Nech a_i, b_i, c_i sú postupne i-te znaky (0 alebo 1) reťazcov na vstupe. Keď zoberieme ľubovoľný reťazec dĺžky $i - 1$, ktorý sa od vstupných troch reťazcov odlišuje postupne na a, b, c pozíciách, a pridáme na koniec nulu, bude sa od reťazcov líšiť na $a + a_i$, $b + b_i$ a $c + c_i$ pozíciách. Keby sme na koniec pridali jednotku, bude sa odlišovať na $a + (1 - a_i)$, $b + (1 - b_i)$ a $c + (1 - c_i)$ pozíciách. Preto

$$P[i, a, b, c] = P[i - 1, a - a_i, b - b_i, c - c_i] + P[i - 1, a - (1 - a_i), b - (1 - b_i), c - (1 - c_i)]$$

Všimnite si, že súčet hodnôt $P[i, a, b, c]$ pre všetky možné a, b, c je 2^i, čo je práve počet binárnych reťazcov dĺžky i.

Týmto výrazom dokážeme spočítať všetky hodnoty P a na konci už len sčítame počet spôsobov, ktorými vieme získať recepty dĺžky n s rozdielmi chutí k od všetkých troch čokolád pre $0 \leq k \leq n$. Čiže postupne sčítame $P[n, k, k, k]$ pre všetky k.

Výpočet každej z $O(n^4)$ hodnôt nám trvá konštantne dlho, takže časová zložitosť je $O(n^4)$.

Pamätáme si všetkých $O(n^4)$ hodnôt, taká je teda pamäťová zložitosť. Ak si všimneme že pre odpovede na dĺžku receptov $i + 1$ nás už zaujímajú len odpovede pre dĺžku i, môžeme si vždy pamätať len odpovede pre poslednú vypočítanú dĺžku a tú, ktorú práve počítame. Tým by sme sa zlepšili na $O(n^3)$.

Listing programu (C++)

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

int main() {
  int n, mod = 1000000007;
  cin >> n;
  string s[3];
  for (int i = 0; i < 3; ++i) cin >> s[i];

  int dp[n + 1][n + 1][n + 1][n + 1];
  memset(dp, 0, sizeof(dp)); //vynulujeme
  dp[0][0][0][0] = 1;

  for (int i = 0; i < n; ++i)
    for (int a = 0; a <= i; ++a)
      for (int b = 0; b <= i; ++b)
        for (int c = 0; c <= i; ++c)
          for (int znak = 0; znak <= 1; ++znak) {
            // nove rozdiely chuti troch cokolad ak doteraz ich rozdiely boli a/b/c
            // a dalsia ingrediencia v cokolade je (znak = 1) alebo nie je (znak = 0)
            int A = a + (s[0][i] - '0' != znak);
            int B = b + (s[1][i] - '0' != znak);
            int C = c + (s[2][i] - '0' != znak);
            dp[i + 1][A][B][C] += dp[i][a][b][c];
            if (dp[i + 1][A][B][C] >= mod) dp[i + 1][A][B][C] -= mod;
          }

  // scitame pocet receptov pre ktore je rozdiel rovnaky pre vsetky cokolady
  int sum = 0;
  for (int k = 0; k <= n; ++k) sum += dp[n][k][k][k];
  cout << sum % mod << "\n";
}
```

Myšlienka vzorového riešenia

Dôležité je všimnúť si, že na poradí ingrediencií vôbec nezáleží a jediné dôležité sú tieto štyri počty:

- počet ingrediencií, v ktorých sú druhá a tretia čokoláda rovnaké, ale odlišné od prvej
- počet ingrediencií, v ktorých sú prvá a tretia čokoláda rovnaké, ale odlišné od druhej
- počet ingrediencií, v ktorých sú prvá a druhá čokoláda rovnaké, ale odlišné od tretej
- počet ingrediencií, v ktorých sú všetky tri čokolády rovnaké

Tieto typy pozícií si postupne označíme pozície typu A, typu B, typu C a typu D. (Žiadne iné typy ingrediencií nie sú.) Počty týchto pozícií si označíme postupne A, B, C, D. Riešenie úlohy závisí len od týchto štyroch čísel, akonáhle ich poznáme, môžeme samotné recepty čokolád zabudnúť. Taktiež nie je dôležité, v akom poradí boli čokolády na vstupe uvedené, takže si čísla A, B, C môžeme poprehadzovať, tak aby platilo $A \leq B \leq C$. Toto preusporiadanie nám zjednoduší riešenie, lebo nebudeme musieť rozoberať veľa prípadov.

Každý možný výsledný recept R je charakterizovaný štyrmi číslami

- a je počet pozícií typu A, na ktorých sa R líši od prvého reťazca
- b je počet pozícií typu B, na ktorých sa R líši od druhého reťazca
- c je počet pozícií typu C, na ktorých sa R líši od tretieho reťazca
- d je počet pozícií typu D, na ktorých sa R líši od všetkých reťazcov

Pozor, a, b, c v tejto časti znamenajú niečo úplne iné ako a, b, c v časti o dynamickom programovaní.

Pomocou čísel a, b, c, d vieme presne vyjadriť, na koľkých miestach sa líši R od jednotlivých reťazcov. Napríklad od prvého reťazca sa líši na $a + (B - b) + (C - c) + d$ pozíciách. Aby sa náš recept líšil na rovnako veľa pozíciách aj od druhého reťazca, musí platiť, že toto číslo je rovné $(A-a)+b+(C-c)+d$. Čiže $b-a = (B-A)/2$. Podobne zistíme, že $c - a = (C - A)/2$.

To okrem iného znamená, že ak čísla A, B, C nemajú rovnakú paritu, tak odpoveď na riešenie úlohy je 0. V opačnom prípade vieme, že rozdiel $b - a$ a $c - a$ je konštantný, takže hodnoty b a c sú určené hodnotou a. Na hodnote d nezáleží, takže na pozíciách typu D môže mať náš reťazec hocičo.

Pre spočítanie odpovede nám teda stačí pre každé a, $0 \leq a \leq A$ zistiť, koľko existuje binárnych reťazcov dĺžky n, ktorých počty jednotiek na jednotlivých typoch pozícií sú a pre typ A, $a + (B - A)/2$ pre typ B, $a + (C - A)/2$ pre typ C a hocikoľko pre typ D.

Pre konkrétne a je tento počet $2^D \binom{A}{a} \binom{B}{a+(B-A)/2} \binom{C}{a+(C-A)/2}$. Pričom $\binom{n}{m}$ označuje "koľkými možnosťami vieme vybrať m prvkov z n prvkovej množiny", čoho hodnota je $\frac{n!}{m!(n-m)!}$.

Celé riešenie úlohy je

$$2^D \sum_{a=0}^{A} \binom{A}{a} \binom{B}{a + (B - A)/2} \binom{C}{a + (C - A)/2}.$$

A zostáva nám už len zistiť, ako čo najrýchlejšie spočítať túto sumu. Opäť si všimnite, že odpoveď závisí len od čísel A, B, C, D, ako sme spomínali na začiatku.

Odporúčame dať si na tomto mieste krátku pauzu a napiť sa vody.

Ako krátke intermezzo spomenieme, že spôsobov ako riešiť túto úlohu vzorovo je viacero a väčšina z nich eventuálne skončí tak, že chceme spočítať nejakú sumu, v ktorej nejako vystupujú kombinačné čísla (to sú tie zvieratká typu $\binom{n}{m}$). Podľa toho, ako rýchlo dokážeme spočítať túto sumu rozlišujeme tri hlavné riešenia.

Pomalé kombinačné čísla

Najjednoduchší spôsob ako spočítať sumu je, skonštruovať si prvých C riadkov Pascalovho trojuholníka. V ňom sa nachádzajú všetky kombinačné čísla, ktoré v sume potrebujeme, takže samotnú sumu spočítame v čase $O(n)$. Spočítanie Pascalovho trojuholníka nám však zaberie čas $O(n^2)$.

Modulárne umocňovanie

Už sme spomínali, že kombinačné číslo $\binom{n}{m}$ sa rovná $\frac{n!}{(n-m)!m!}$. Faktoriály vyriešime ľahko, tie si vieme na začiatku predpočítať v čase $O(n)$. Problém však robí delenie modulo p. Pre ilustráciu aké ťažké je deliť, môžete skúsiť zhlavy spočítať, koľko je 7 deleno 13 modulo $10^9 + 7$? (Je to 76923078.) Namiesto delenia preto budeme násobiť inverzným prvkom. Náš prípad, pre $p = 10^9 + 7$, by vyzeral takto: $7/13 \equiv 7 \cdot 13^{-1} \equiv 7 \cdot 13^{p-2} \equiv 7 \cdot 153846155 \equiv 76923078 \pmod{p}$. Pre overenie spočítame, že $13 \cdot 76923078 \equiv 1000000014 \equiv 7 \pmod{p}$. Viac sa môžete dočítať v článku o modulárnom umocňovaní a inverzných prvkoch[8].

Pre nás je dôležité, že inverzný prvok vieme spočítať v čase $O(\log p)$. Inverzný prvok potrebujeme poznať pre každý faktoriál, takže dokopy je to $O(n \log p)$ času, čo podľa podmienky v zadaní je $O(n \log n)$.

Vzorové riešenie v $O(n)$

Po tom, ako si rozpíšeme kombinačné čísla na faktoriály, dokážeme celú sumu napísať v tvare $\sum_{a=0}^{A} \left(\frac{x_a}{y_a}\right)$. Ak si dopredu predpočítame hodnotu faktoriálov od 1! po $n!$ modulo p, dokážeme vypočítať každú z hodnôt x_a, y_a v konštantnom čase.

Teraz si ukážeme všeobecný algoritmus, ktorý dokáže takúto sumu v tvare $\sum_{i=0}^{n} \frac{x_i}{y_i}$ spočítať modulo ľubovoľné prvočíslo v čase $O(n)$. Tým vlastne dostaneme vzorové riešenie celej úlohy v čase $O(n)$. Kľúčom k riešeniu je, že chceme počítať inverzný prvok len raz (pretože ak by sme ho počítali n-krát, trvalo by nám to $O(n \log n)$ času). Preto si sumu prevedieme na spoločný menovateľ čím dostaneme

$$\frac{x_1 y_2 \cdots y_n + x_2 y_1 y_3 \cdots y_n + \cdots + x_n y_1 \cdots y_{n-1}}{y_1 \cdots y_n}$$

Už máme síce len jedno delenie, ale potrebujeme spočítať všetky súčiny $y_1 \cdots y_n$ bez jedného prvku. Postavme si teda binárny strom, ktorý bude mať v listoch hodnoty $y_1 \cdots y_n$, pričom postupnosť y doplníme na konci jednotkami, aby jej dĺžka bola mocnina dvoch. Nájdeme teda také k, že $n \leq 2^k$. Vrcholy stromu si zhora očíslujeme 1 až $2^{k+1} - 1$ (tak, že vrchol v má synov $2v$ a $2v + 1$). Tento strom prejdeme odspodu a pre každý vrchol v si v konštantnom čase spočítame hodnotu Y_v, súčin y-ov v listoch jeho podstromu. Na to nám stačí vynásobiť hodnoty Y v jeho synoch: $Y_v = Y_{2v} \cdot Y_{2v+1} \mod p$.

Prejdeme binárny strom znova, tentokrát odvrchu a spočítame si hodnoty Y'_v definované nasledovne:

- $Y'_1 = 1$,
- $Y'_{2v} = Y'_v \cdot Y_{2v+1} \mod p$ pre $1 \leq v \leq 2^k - 1$
- $Y'_{2v+1} = Y'_v \cdot Y_{2v} \mod p$, pre $1 \leq v \leq 2^k - 1$

[8] https://www.ksp.sk/kucharka/modularna_aritmetika

Rozmyslite si, že hodnoty Y'_v predstavujú súčin všetkých y **okrem** tých v listoch podsromu v, takže hodnoty Y' v listoch stromu sú hľadané súčiny.

Celé to vieme spočítať dvoma prechodmi binárneho stromu v čase $O(n)$. Pamäťová zložitosť riešenia je tiež $O(n)$. Lepšiu časovú zložitosť dosiahnuť nevieme, pretože musíme načítať celý vstup.

Listing programu (C++)

```
#include<cstdio>
#include<algorithm>
using namespace std;
#define For(i, n) for(int i = 0; i<int(n); ++i)
typedef long long ll;

inline ll mod(const ll& x) { return x % 1000000007; }
inline ll inv(ll x) {  // Inverzny prvok ku x
    int e = 1000000005;
    ll r = 1;
    while (e > 0) {
        if (e%2) r = mod(r*x);
        x = mod(x*x);
        e/=2;
    }
    return r;
}
char cok[3][300047];  // cokolady
int n;
ll A, B, C, D, pocet = 1;
ll F[300047], Y[400047], YY[400047]; // A<=100000

int main() {
    // spracujeme vstup
    scanf("%d␣%s␣%s␣%s", &n, cok[0], cok[1], cok[2]);
    For(i, n) {
        A += cok[0][i] != cok[1][i] && cok[1][i] == cok[2][i];
        B += cok[0][i] != cok[1][i] && cok[0][i] == cok[2][i];
        C += cok[0][i] == cok[1][i] && cok[0][i] != cok[2][i];
        D += cok[0][i] == cok[1][i] && cok[0][i] == cok[2][i];
    }
    // ked nesedi parita, odpoved je 0
    if ((A+B)%2 || (A+C)%2) {
        printf("0\n");
        return 0;
    }
    // utriedime A,B,C
    if (A > B) swap(A,B);
    if (B > C) swap(B,C);
    if (A > B) swap(A,B);
    ll BA = (B-A)/2, CA = (C-A)/2;
    // spocitame mocninu dvojky, faktorialy, citatele sumy
    For(i, D) pocet = mod(pocet * 2);
    F[0] = 1;
    For(i, C) F[i+1] = mod((i+1)*F[i]);
    pocet = mod(mod(pocet*F[A])*mod(F[B]*F[C]));
    // mocnina dvojky vacsia ako A
    int npow = 8;
    while(npow <= A) npow *= 2;
    // najdeme menovatele sumy
    For(i, npow*2) Y[i] = 1;
    For(i, A+1) {
        Y[npow+i] = mod(
            mod(F[i]*F[A-i]) *
            mod(
                mod(F[BA+i]*F[B-BA-i]) *
                mod(F[CA+i]*F[C-CA-i])
            )
        );
    }
    // spocitame sumu(i=0..A) 1/Y[npow+i] v O(n)
    for(int i = npow-1; i>0; --i) {
        Y[i] = mod(Y[2*i]*Y[2*i+1]);
    }
    YY[1] = 1;
    for(int i = 1; i<npow; ++i) {
        YY[2*i] = mod(YY[i] * Y[2*i+1]);
        YY[2*i+1] = mod(YY[i] * Y[2*i]);
    }
    ll suma = 0, menovatel = 1;
    For(i, A+1) {
        suma += YY[npow+i];
        menovatel = mod(menovatel*Y[npow+i]);
    }
    pocet = mod(mod(pocet*inv(menovatel)) * mod(suma));
    printf("%lld\n", pocet);
}
```

Baklažán

8. Automat

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

Spôsobov, ako môže Erik postupne nakupovať tyčinky, je veľa. Na začiatok je preto dobré uvedomiť si, že je dôležité, koľko akých tyčiniek Erik kúpi, ale nie je dôležité, v akom poradí:

- Ak nám Erik povie, že kúpil (t. j. vybral a **zaplatil**) $t_1, t_2, \ldots, t_n$ tyčiniek druhov $1, 2, \ldots, n$, vieme už jednoznačne určiť, koľko akých tyčiniek mu automat vysypal:
  - Tyčinky druhu n dával automat Erikovi iba vtedy, keď si ich naozaj kúpil, teda Erik ich dostal dokopy t_n.
  - Tyčinky druhu $n-1$ sa automat snažil Erikovi dať vždy, keď si kúpil tyčinku druhu $n-1$ alebo n. To znamená, že ak má automat na začiatku $t_n + t_{n-1}$ alebo viac tyčiniek druhu $n-1$, Erik bude mať na konci $t_n + t_{n-1}$ tyčiniek tohto druhu. V opačnom prípade dostane Erik všetkých p_{n-1} tyčiniek druhu $n-1$.
  - Túto úvahu môžeme zovšeobecniť: tyčiniek druhu i Erik dostane presne

$$\min(p_i,\ t_i + t_{i+1} + \cdots + t_n).$$

Samozrejme, ak by Erik nakupoval tyčinky v zlom poradí, mohol by sa nejaký druh minúť skôr, než z neho Erik kúpi všetky tyčinky ktoré chcel. Tomuto sa však vždy dá ľahko vyhnúť: jednoducho najprv nakúpime všetky tyčinky druhu 1 ktoré chceme, potom tyčinky druhu 2, potom druhu 3 atď.. Ako sme si ukázali, toto poradie je rovnako dobré ako každé iné.

Pri hľadaní optimálneho riešenia teda môžeme spokojne zabudnúť na to, že tyčinky sa kupujú v nejakom poradí a stačí nám nájsť optimálnu sadu tyčiniek, ktoré má Erik kúpiť.

Dynamické programovanie

Naše riešenie bude založené na technike zvanej *dynamické programovanie*. Keď riešime úlohu pomocou dynamického programovania, musíme si položiť dobrú otázku. V našom prípade to bude otázka:

Pre dané X, Y, Z: Akú najväčšiu hodnotu vieme získať v tyčinkách prvých X druhov, ak na tyčinky prvých X druhov minieme nanajvýš Y peňazí a z posledných $n-X$ druhov kúpime dokopy presne Z tyčiniek?

Táto otázka znie dosť krkolomne,[9] má však niekoľko kľúčových dobrých vlastností. Označme si odpoveď na našu otázku pre nejaké X, Y, Z ako $o(X, Y, Z)$:

- Je len ohraničený počet možných trojíc X, Y, Z, pre ktoré má zmysel sa našu otázku pýtať.
- Hodnota $o(n, k, 0)$ je riešením našej úlohy.
- Ak $X = 0$, odpoveď vieme ľahko zistiť: je rovná 0.
- Odpoveď pre nejaké $X > 0, Y, Z$ vieme ľahko zistiť, ak poznáme odpovede pre menšie X. Stačí nám rozobrať všetky možnosti, koľko tyčiniek druhu X kúpime:
  - Ak sa rozhodneme nekúpiť žiadnu tyčinku druhu X, získame od automatu $\min(Z, p_X)$ tyčiniek tohto druhu (keďže kupujeme Z tyčiniek druhov s vyšším číslom). Tie budú mať hodnotu $c_X \cdot \min(Z, p_X)$. V tyčinkách druhov 1 až $X-1$ potom získame nanajvýš $o(X-1, Y, Z)$ peňazí, dokopy teda získame $c_X \cdot \min(Z, p_X) + o(X-1, Y, Z)$.
  - Ak kúpime jednu tyčinku druhu X, získame $\min(Z+1, p_X)$ tyčiniek druhu X, v hodnote $c_X \cdot \min(Z+1, p_X)$. Na tyčinky druhov 1 až $X-1$ nám pritom ostane $Y - c_X$ peňazí, pričom tyčiniek druhov X až n kupujeme dokopy $Z+1$. To znamená, že v tyčinkách druhov $1, \ldots, X-1$ získame nanajvýš $o(X-1, Y-c_X, Z+1)$ peňazí, dokopy $o(X-1, Y-c_X, Z+1) + c_X \cdot \min(Z+1, p_X)$.
  - Všeobecne, ak kúpime k tyčiniek druhu X, získame $c_X \cdot \min(Z+k, p_X)$ v tyčinkách druhu X a $o(X-1, Y-kc_X, Z+k)$ v tyčinkách nižších druhov. Toto má, samozrejme, zmysel iba pre také k, že $kc_X \leq Y$ a $k \leq p_X$.

  Hodnota $o(X, Y, Z)$ je teda maximum z hodnôt

$$c_X \min(Z+k, p_X) + o(X-1, Y-kc_X, Z+k)$$

  pre všetky zmysluplné k.

Odpoveď na našu otázku by sme teda vedeli počítať jednoduchou rekurzívnou funkciou:

[9] Ak viete používať dynamické programovanie, vymyslieť túto otázku bola pravdepodobne najťažšia časť tejto úlohy.

```
function o(X, Y, Z):
    if X = 0:
        return 0
    best := 0
    max_k := min(Y / c[X], p[X])
    for k := 0, 1, ... , max_k:
        best := max(best, c[X] * min(Z + k, p[X]) + o(X-1, Y - k * c[X], Z + k))
    return best
```

Túto funkciu nám stačí zavolať pre $X = n, Y = k, Z = 0$ a ona nám vypočíta výsledok. Keďže v rekurzívnych volaniach sa táto funkcia volá so stále menším X (a pre $X = 0$ sa už rekurzívne nevolá), skončí to v konečnom čase. Časová zložitosť však bude nepekná.

Memoizácia

Časovú zložitosť nášho algoritmu môžeme podstatne zlepšiť tým, že prestaneme počítať rovnaké veci veľakrát. Konkrétne, vždy keď zavoláme našu funkciu pre nejakú kombináciu X, Y, Z prvý raz, spočítame výsledok a zapamätáme si ho. Keď niekedy neskôr zavoláme funkciu s tými istými parametrami, nebudeme ju znovu počítať, ale vrátime už zapamätaný výsledok. Tomuto triku sa hovorí *memoizácia*. Uvedomte si, že tým preskočíme aj rekurzívne volania, ktoré by vyprodukovali kopec ďalších rekurzívnych volaní, takže nám to často ušetrí veľmi veľa času.

Ako si však pamätať vypočítané hodnoty našej funkcie? Jednoducho: použijeme obyčajné trojrozmerné pole, kde na indexe `[X][Y][Z]` bude buď hodnota $o(X, Y, Z)$ ak sme ju už niekedy spočítali, alebo špeciálna hodnota (napr. `-1`, `NULL` alebo `None`) signalizujúca, že sme $o(X, Y, Z)$ ešte nikdy nepočítali.

Aké veľké má byť toto pole? Parameter X bude v každom volaní funkcie `o(X, Y, Z)` niekde medzi 0 a n. Parameter Y bude v prvom volaní k a v každom ďalšom už len menší alebo rovnaký, teda stále bude medzi 0 a k. Parameter Z bude v prvom volaní funkcie 0 a v ďalších volaniach bude rásť. Nikdy však nebude väčší, než počet všetkých tyčiniek v automate. Označme si počet tyčiniek najpočetnejšieho druhu (najväčšie z čísel $p_1, \ldots, p_n$) ako P. Potom môžeme parameter Z zhora odhadnúť číslom nP. Naše pole s výsledkami funkcie `o(X, Y, Z)` teda bude mať veľkosť $(n + 1) \times (k + 1) \times (nP + 1)$.

Zložitosť

Akú zložitosť má tento algoritmus? Pamäťová zložitosť je jednoduchá: najviac pamäte zaberie naša tabuľka s výsledkami, ktorá má veľkosť $O(n^2kP)$. Pri časovej zložitosti potrebujeme sčítať časy všetkých volaní funkie `o(X, Y, Z)`. Tie si môžeme rozdeliť na dve skupiny:

- Volania, pri ktorých sme funkciu naozaj počítali (dané X, Y, Z sme videli prvýkrát). Takýchto volaní je nanajvýš $O(n^2kP)$ (lebo toľko je zmysluplných kombinácií X, Y, Z) a každé z nich trvá čas $O(P)$ (musíme vyskúšať všetky možné počty tyčiniek X-tého druhu). Tieto volania majú teda zložitosť $O(n^2kP^2)$.
- Volanie, pri ktorých sme iba vytiahli výsledok z tabuľky. Každé z týchto volaní má zložitosť $O(1)$. Navyše vieme, že funkciu `o(X, Y, Z)` sme rekurzívne volali iba vo volaniach prvého druhu, v každom z nich nanajvýš P-krát. To znamená, že všetkých volaní funkcie $o(X, Y, Z)$ bolo dokopy nanajvýš $O(n^2kP^2)$ a teda volaní druhého druhu mohlo byť tiež nanajvýš $O(n^2kP^2)$. To znamená, že ich časová zložitosť je dokopy $O(n^2kP^2)$.

Celý náš algoritmus má teda časovú zložitosť $O(n^2kP^2)$.

Jednoduché zlepšenia

Časovú zložitosť nášho algoritmu môžeme zlepšiť, ak si všimneme zopár vecí. Cenu najdrahšieho druhu tyčiniek označme C. Ak má Erik PC alebo viac peňazí, dokáže kúpiť všetky tyčinky v automate: stačí mu vždy kupovať tyčinku s najvyšším číslom, ktorú automat ešte má. Takýmto spôsobom dostane pri každom nákupe jednu tyčinku z každého druhu (okrem tých, čo sa už minuli), a teda po P nákupoch bude mať všetky tyčinky.

To znamená, že ak $k \geq PC$, nemusíme volať našu rekurzívnu funkciu, ale stačí nám iba sčítať ceny všetkých tyčiniek. Vďaka tomu nebudeme nikdy volať funkciu `o(X, Y, Z)` s parametrom Y väčším ako PC, teda nám aj stačí tabuľka s výsledkami veľkosti $(n + 1) \times (PC + 1) \times (nP + 1)$ (pri obmedzeniach $k \leq 200\,000$ a $P, C \leq 50$, ktoré boli v praktickom testovaní, je to zlepšenie).

Ďalej si môžeme všimnúť, že ak kúpime P alebo viac tyčiniek druhov $X + 1, \ldots, n$, potom už automaticky dostaneme všetky tyčinky druhov $1, \ldots, X$. To znamená, že našu funkciu nemá zmysel volať s paramtrom Z väčším ako P. Našu tabuľku teda môžeme zmenšiť na $(n + 1) \times (PC + 1) \times (P + 1)$.

Týmito dvoma zlepšeniami stlačíme pamäťovú zložitosť na $O(nP^2C)$ a časovú na $O(nP^3C)$. Toto stačilo na plný počet bodov v praktickom testovaní.

Zložitejšie zlepšenie

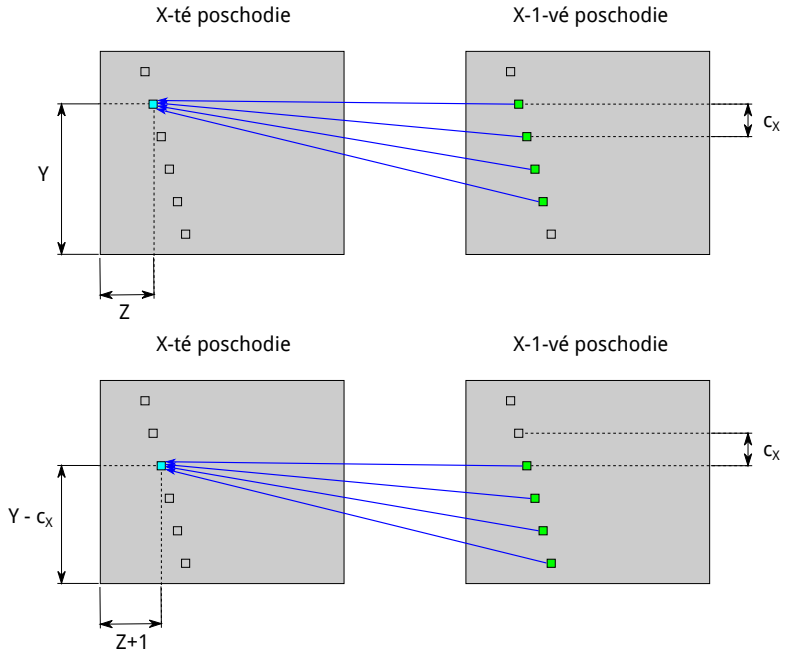
Zoberme si nejaké fixné čísla X, Y, Z. V našom výpočte $o(X, Y, Z)$ počítame v cykle maximum z čísel

$$c_X \min(Z + k, p_X) + o(X - 1, Y - kc_X, Z + k)$$

pre k od 0 po p_X, resp. pokým $Y - kc_X \geq 0$ a $Z + k \leq P$. Pri výpočte $o(X, Y - c_X, Z + 1)$ počítame maximum z čísel

$$c_X \min(Z + 1 + k, p_X) + o(X - 1, Y - c_X - kc_X, Z + 1 + k)$$

pre k od 0 po p_X (resp. pokým dáva daný výraz zmysel), čo je vlastne presne tá istá postupnosť čísel, akurát posunutá o jeden člen. Pri výpočte $o(X, Y - 2c_X, Z + 2)$ budeme opäť rátať maximum z tej istej postupnosti, posunutej o ďalší člen.



Ak si zoberieme čísla

$$c_X \min(Z + k, p_X) + o(X - 1, Y - kc_X, Z + k)$$

pre **všetky** (aj záporné) k, pre ktoré dáva výraz $o(X-1, Y-kc_X, Z+k)$ zmysel, dostaneme nejakú postupnosť čísel, označme ju T. Pre každé zmysluplné (aj záporné) j potom platí, že hodnota $o(X, Y - jc_X, Z + j)$ je maximum z nejakých $p_X + 1$ po sebe idúcich členov postupnosti T (resp., ak sme bližšie než p_X od konca postupnosti, tak maximum všetkých členov od nejakého bodu až do konca). V úlohe 6[10] predchádzajúceho kola sme si ukázali, ako sa dajú v lineárnom čase spočítať maximá všetkých l-tíc po sebe idúcich členov nejakej postupnosti. To môžeme urobiť pre $l = p_X + 1$. Ak teda poznáme hodnoty prvkov postupnosti T, všetky hodnoty $o(X, Y - jc_X, Z + j)$ (pre zmysluplné j) dokážeme spočítať v čase lineárnom od ich počtu. To je rovnako dobré, ako keby sme vedeli funkciu o v jednom z týchto bodov spočítať za konštantný čas.

Vylepšený algoritmus teda bude vyzerať nasledovne:

- Tabuľku s hodnotami `o(X, Y, Z)` budeme vypĺňať postupne po "poschodiach": najprv vyplníme všetky hodnoty, kde $X = 0$, potom hodnoty kde $X = 1$ atď.
- Pri vypĺnaní jedného poschodia:
  - Najprv si jednotlivé políčka tabuľky rozdelíme do skupín, pričom v jednej skupine budú všetky políčka s indexami tvaru `[X][Y - j * c[X]][Z + j]` pre nejaké Y, Z.
  - Pre každú skupinu si zostrojíme príslušnú postupnosť T (na to využijeme predošlé poschodie tabuľky).
  - Následne v lineárnom čase vypočítame hodnotu funkcie `o` na všetkých indexoch v našej skupine.

[10] https://www.ksp.sk/ulohy/zadania/1425/

Keďže vypočítanie jednej hodnoty $o(X, Y, Z)$ bude v priemere trvať konštantný čas, celková zložitosť nášho algoritmu je $O(nP^2C)$, rovnako ako pamäťová. Pamäťová zložitosť sa dá ešte vylepšiť na $O(P^2C)$, ak si nebudeme pamätať celú našu tabuľku, ale iba aktuálne vypĺňané poschodie a poschodie tesne pod ním.

Listing programu (C++)

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

vector<int> maxima_intervalov(vector<int> T, int l)
{
    vector<int> vysledok;
    deque<pair<int, int> > fronta; // dvojice (hodnota, index)
    for(int zaciatok=-(l-1); zaciatok < (int)T.size(); zaciatok++)
    {
        int koniec = zaciatok + l - 1;
        if(koniec < T.size())
        {
            while(!fronta.empty() && fronta.back().first <= T[koniec])
            {
                fronta.pop_back();
            }
            fronta.push_back(pair<int, int>(T[koniec], koniec));
        }
        if(zaciatok >= 0)
        {
            while(fronta.front().second < zaciatok)
            {
                fronta.pop_front();
            }
            vysledok.push_back(fronta.front().first);
        }
    }
    return vysledok;
}

int main()
{
    int n, k;
    scanf("%d %d", &n, &k);
    vector<int> c(n), p(n);
    int P = 0, C = 0; //maximalny pocet a maximalna cena
    for(int i=0; i<n; i++)
    {
        scanf("%d", &c[i]);
        C = max(C, c[i]);
    }
    for(int i=0; i<n; i++)
    {
        scanf("%d", &p[i]);
        P = max(P, p[i]);
    }

    int max_penazi = min(k, P*C);
    vector<vector<int> > predosle_poschodie(max_penazi+1, vector<int> (P+1, 0));
    vector<vector<int> > aktualne_poschodie(max_penazi+1, vector<int> (P+1, -1));

    for(int X = 0; X < n; X++)
    {
        for(int Y = 0; Y <= max_penazi; Y++)
        {
            for(int Z = 0; Z <= P; Z++)
            {
                if(Z == 0 || Y + c[X] > max_penazi) // prvy prvok nejakej postupnosti
                {
                    vector<int> T;
                    for(int k = 0; Y - k*c[X] >= 0 && Z+k <= P; k++)
                    {
                        T.push_back(c[X] * min(Z+k, p[X]) + predosle_poschodie[Y - k*c[X]][Z+k]);
                    }
                    vector<int> maxima = maxima_intervalov(T, p[X] + 1);
                    int i = 0;
                    for(int k = 0; Y - k*c[X] >= 0 && Z+k <= P; k++)
                    {
                        aktualne_poschodie[Y - k*c[X]][Z+k] = maxima[i];
                        i++;
                    }
                }
            }
        }
        predosle_poschodie = aktualne_poschodie;
    }
    printf("%d\n", aktualne_poschodie[max_penazi][0]);
    return 0;
}
```