Korešpondenčný seminár z programovania
XXXIV. ročník, 2016/17
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 1. kola zimnej časti

1. Zavesme tam niečo!

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

Ako funguje čítanie vstupu? Štandardne čítame vstup po slovách – funkcie ako `cin` a `read` nám do premennej uložia *ďalšie* slovo (alebo číslo) na vstupe.

Každý textový súbor (aj vstup pre naše programy) sa skladá z množstva znakov. Niektoré z nich sú takzvané *prázdne* znaky (whitespace characters) – napríklad medzera (`' '`), tabulátor (`'\t'`), znak nového riadku (`'\n'`) alebo aj znak konca súboru (EOF, End Of File). Program tieto prázdne znaky pri čítaní vstupu pomocou horeuvedených funkcií preskakuje. Na to, aby sme vyriešili túto úlohu, musíme postupovať inak...

Vstup musíme čítať buď po jednotlivých znakoch, kedy načítame naozaj každý znak na vstupe samostatne, alebo načítavať celé riadky naraz – teda načítať znaky, až kým neprídeme k znaku konca riadku.

Ako to ale naprogramovať? Ak ste nič takéto ešte nerobili, na internete rýchlo nájdete odpoveď.

Čítanie celých riadkov je pre nás o niečo pohodlnejšie. V `C++` na to môžeme použiť funkciu `getline(cin, s)`, kde `s` je názov premennej typu `string`. Táto funkcia vracia hodnotu typu `boolean`, ktorá je `true`, ak sa programu podarilo niečo načítať. Ak používame `Pascal`, môžeme použiť funkciu `readln(s)`. Ak by ste potrebovali zistiť, čo presne dané funkcie robia, ľahko nájdete ich presné popisy na internete v *dokumentácií* daných jazykov, napríklad pre C++ na stránke cplusplus.com.

Keď už vieme vstup načítať, musíme ešte vymyslieť, ako spočítať plochu obrazu. Jedna možnosť je pozerať sa na výskyt znakov `+`, ktoré tvoria rám obrazu. Musíme si však dať pozor, aby nás nepomýlil znak `+` vo vnútri obrazu. Najelegantnejšie to vyriešime tak, že si uložíme súradnice prvého a posledného `+` na vstupe a z nich následne vypočítame plochu obrazu.

Listing programu (C++)

```
#include <iostream>

using namespace std;

int main()
{
    string s;
    bool prvy = false;
    int sx, sy, fx, fy, riadok = 0;
    while(getline(cin, s))
    {
        for(int stlpec = 0; stlpec < s.size(); stlpec++)
        {
            if(s[stlpec] == '+')
            {
                if(!prvy)
                {
                    prvy = true;
                    sx = stlpec;
                    sy = riadok;
                }
                fx = stlpec;
                fy = riadok;
            }
        }
        riadok++;
    }
    cout << (fx-sx-1)*(fy-sy-1) << endl;
}
```

Druhou možnosťou je zistiť počet riadkov, ktoré nie sú prázdne, zistiť dĺžku jedného z nich po orezaní medzier na okrajoch a veľkosť dopočítať z týchto dvoch údajov.

Listing programu (C++)

```
#include <iostream>

using namespace std;

string trim(string str)
{
```

```
    int first = str.find_first_not_of(' ');
    int last = str.find_last_not_of(' ');
    if(first == -1) return "";
    return str.substr(first, (last-first+1));
}

int main(){
    string s;
    int sirka = 0;
    int vyska = 0;
    while(getline(cin, s)){
        s = trim(s);
        if(s.length() > 0){
            sirka = s.length();
            vyska++;
        }
    }
    cout << (sirka-2)*(vyska-2) << endl;
}
```

Listing programu (Python)

```
import sys

width, height = 0, 0

for line in sys.stdin:
    if len(line.strip()) > 0:
        height += 1
        width = len(line.strip())

print((width - 2) * (height - 2))
```

Zygro

2. Zrýchlenie parkoviska

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

Pre každý riadok vstupu (ŠPZ auta) riešime jednu z dvoch úloh: parkovanie (nájdi voľné miesto) alebo vyparkovanie (nájdi miesto, na ktorom auto bolo).

Obe sa dajú jednoducho vyriešiť napríklad takto: parkovisko si budeme pamätať ako pole, kde `parkovisko[i]` bude obsahovať buď ŠPZ auta, ktoré stojí na mieste i, alebo špeciálnu hodnotu, ak je miesto prázdne. Na zodpovedanie otázky prezeráme parkovisko miesto po mieste, kým nenájdeme dané auto a potom ho vyparkujeme. Ak auto nenájdeme, vyhľadáme mu voľné miesto. Toto riešenie je ale zjavne veľmi pomalé, pretože pre každé auto musíme prehľadať prinajhoršom celé parkovisko. Časová zložitosť takéhoto riešenia je potom $O(mn)$.

Malým, ale v niektorých prípadoch významným, zlepšením jednoduchého riešenia je nezačínať hľadanie prázdneho miesta vždy odznova. Budeme ho hľadať od miesta, kam zaparkovalo posledné auto. Týmto zrýchlime hľadanie miesta, samozrejme, len pokiaľ sa nezaplní posledné miesto parkoviska. Zvyšok výpočtu sa nespomalí, a preto je toto riešenie lepšie, hlavne ak prichádzajúcich áut je málo a parkovisko veľké.

Toto ale nestačí. Ak by po zaplnení parkoviska stále prichádzali autá na miesta s číslami m a $m/2$, pre každú otázku zo vstupu prehľadávame stále $O(m)$ miest. Chceme teda nájsť buď voľné miesto, alebo miesto, kde je auto zaparkované bez nutnosti prehľadať parkovisko. Na to si ukážeme dva zlepšováky:

Keď auto zaparkuje, jeho miesto sa už meniť nebude. Preto môžeme vyparkovanie zrýchliť na konštantný počet operácií tým, že pre každé zaparkované auto si budeme pamätať miesto, na ktorom stojí a pri vyparkovaní ho uvoľníme. Toto dosiahneme napríklad tým, že budeme mať pole a v ňom políčko pre každú možnú ŠPZ. Keď auto nie je zaparkované, pamätáme si v tomto políčku hodnotu -1 (alebo inú, ktorá nie je platné číslo parkovacieho miesta). Ak príde auto zaparkovať a nájdeme mu miesto, do tohto políčka uložíme číslo miesta, kam zaparkovalo. Keď bude odchádzať, stačí sa pozrieť na toto políčko a hneď vieme, ktoré miesto treba uvoľniť.

Druhým zlepšovákom chceme zrýchliť hľadanie voľného miesta. Uvedomte si, že je nám jedno, na ktoré voľné miesto prichádzajúce auto zaparkujeme. Dôležité je len, aby sme to robili rýchlo. Pamätajme si preto všetky voľné miesta v jednom poli. Na začiatku toto pole obsahuje všetky čísla od 1 po m.

Ak príde auto a chceme mu priradiť voľné miesto, pozrieme sa jednoducho na posledné miesto tohto poľa. Tam sa nachádza nejaké voľné miesto, ktoré mu priradíme. V tomto momente však toto miesto už nie je voľné – potrebovali by sme ho z tohto poľa vyškrtnúť a pole skrátiť. To vieme robiť rýchlo buď pomocou dynamických polí[1] alebo len použijeme jednu premennú `posledneVolne`, ktorá nám hovorí, koľko voľných parkovacích miest máme. Namiesto skracovania poľa nám preto bude stačiť zmenšiť túto premennú o 1.

Ak auto odchádza, vyššie uvedeným spôsobom zistíme, ktoré miesto uvoľnilo. Toto voľné miesto si zapíšeme do nášho poľa voľných miest, a to tak, že zväčšíme premennú `posledneVolne` a do poľa si na toto miesto zapíšeme uvoľnené parkovacie miesto.

[1] `vector::pop_back()` v C++, `list::pop()` v Pythone

Takto máme celý čas k dispozícii zoznam voľných miest a nemusíme sa preto pozerať na všetky ostatné (aj obsadené) parkovacie miesta. Dátová štruktúra, do ktorej vieme na koniec vkladať a z jej konca vyberať prvky sa často používa a tak dostala aj názov – zásobník, po anglicky `stack`. V predošlých odsekoch sme si vlastne popísali, ako taký zásobník naprogramovať.

Keď použijeme oba zlepšováky, na parkovanie alebo vyparkovanie jedného auta potrebujeme iba konštantný počet operácií. Obslúžiť n príchodov alebo odchodov bude teda trvať $O(n)$ operácií.

V pamäti si musíme držať voľné miesta, ktorých je nanajvýš m a parkovacie miesta pre všetky ŠPZky[2]. Napriek tomu, že v našej úlohe sú tieto hodnoty len do 1 000 000, nie je slušné povedať, že pamäťová zložitosť je konštantná. Omnoho viac prezradíme, ak si označíme maximálnu hodnotu ŠPZ ako napríklad S a potom môžeme povedať, že pamäťová zložitosť algoritmu je $O(m + S)$.

Listing programu (C++)

```cpp
#include <iostream>

#define MAXSPZ 1000042

using namespace std;

int parkovacie_miesto[MAXSPZ]; // -1 ak auto s danou SPZ nie je na parkovisku

int main() {
    int n, m;

    cin >> m >> n;
    int volne[m];

    //inicializujeme polia
    for (int i = 0; i < MAXSPZ; i++) {
        parkovacie_miesto[i] = -1;
    }
    int posledneVolne = m-1;
    for (int i = 0; i < m; i++) {
        volne[i] = i;
    }

    for (int i = 0; i < n; i++) {
        int spz;
        cin >> spz;
        if (parkovacie_miesto[spz] != -1) { // ak auto je uz zaparkovanie
            int praveUvolnene = parkovacie_miesto[spz];
            parkovacie_miesto[spz] = -1;
            posledneVolne++;
            volne[posledneVolne] = praveUvolnene;
            cout << praveUvolnene << endl;
        } else {
            if (posledneVolne == -1) {      // ak chcem parkovat ale nie je volne miesto
                cout << "plne" << endl;
            } else {                        // ak parkujem a mam kam
                int praveZaplnene = volne[posledneVolne];
                posledneVolne--;
                parkovacie_miesto[spz] = praveZaplnene;
                cout << praveZaplnene << endl;
            }
        }
    }
    return 0;
}
```

Listing programu (Python)

```python
m, n = [int(x) for x in input().split(' ')]

parkovacie_miesto = [-1] * 1000047     # -1 ak auto s danou SPZ nie je na parkovisku
volne = [x for x in range(0, m)]

for i in range(n):
    spz = int(input())
    if (parkovacie_miesto[spz] != -1): # ak auto je uz zaparkovane
        miesto = parkovacie_miesto[spz]
        volne.append(miesto)
        parkovacie_miesto[spz] = -1
        print(miesto)
    else:
        if (len(volne) == 0):          # ak chcem parkovat ale nie je volne
            print('plne')
        else:                          # ak chcem parkovat a mam kam
            miesto = volne.pop()
            parkovacie_miesto[spz] = miesto
            print(miesto)
```

[2] Ak by sme použili hashovaciu tabuľku, stačilo by nám pamätať si v nej pre každé auto jeho miesto a zmestili by sme sa do pamäte $O(m + n)$. V tejto úlohe to však nebolo potrebné.

Mário

3. Znovu začíname

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

V tejto úlohe sme na vstupe dostali popis *terajšej výsledkovky* a bolo našou úlohou zistiť, či mohla táto výsledkovka vzniknúť z nejakých výsledkov predošlého kola a ak áno, mali sme jednu takúto *predošlú výsledkovku* (riešenie úlohy) vypísať.

Kedy to nejde?

Uvažujme najprv taký vstup, v ktorom nie sú **nuloví riešitelia**. Poďme skúsiť nájsť nejaké kritériá, ktoré musí spĺňať terajšia výsledkovka, ak mala vzniknúť z nejakej predošlej:

- **Prvý riešiteľ musí byť plusový**. Ak by bol mínusový, znamenalo by to, že ho od minulého kola niekto predbehol ... tento riešiteľ by teda musel byť v terajšej výsledkovke vyššie ako prvý, čo sa ale nemôže nikdy stať.
- **Posledný riešiteľ musí byť mínusový**. Podobne, ak by bol posledný riešiteľ plusový, musel sa od predošlého kola zlepšiť a niekoho predbehnúť. Ten niekto by mal byť v terajšej výsledkovke pod ním. Pod posledným riešiteľom ale už nie je nikto, a tak posledný nemôže byť plusový.

Našli sme dve podmienky, ktoré musíme overiť. Ak ľubovoľná z nich neplatí, žiadna predošlá výsledkovka neexistuje. Nevieme však, či stačí overiť **len** tieto podmienky. Možno aj po ich splnení riešenie stále niekedy existovať nebude...

Ako by to mohlo ísť?

Napriek tejto neistote sa pokúsime nájsť riešenie len za predpokladu, že sú splnené tieto dve podmienky.

Keďže mínusoví riešitelia od minulého kola klesli, je pomerne dobrý nápad dať ich čo najviac navrch predošlej výsledkovky (aby mali odkiaľ klesnúť) a podobne môžeme skúsiť dať plusových riešiteľov čo najviac na spodok. Spravíme to ale tak, aby sme zachovali relatívne poradie plusových riešiteľov navzájom a poradie mínusových navzájom:

1. zoberieme všetkých mínusových a v poradí, v akom sú v terajšej výsledkovke ich zapíšeme na vrch predošlej výsledkovky
2. zoberieme všetkých plusových riešiteľov a v poradí, v akom sú v terajšej výsledkovke ich postupne zapíšeme pod mínusových

Z terajšej výsledkovky teda vytvoríme predošlú takto:

```
 terajsia       predosla
 --------       --------
+ peter      (-) zygro
+ petka      (-) mario
- zygro      (+) peter
+ janko      (+) petka
- mario      (+) janko
```

Bude to fungovať?

To, že tento postup bude vždy(ak sú splnené 2 predošlé podmienky) fungovať, si musíme **dokázať**. To ale bude jednoduché.

Pozrime sa na umiestnenie ľubovoľného plusového riešiteľa (napr. Peťky) v predošlej (4. miesto) a v terajšej výsledkovke (2. miesto). Miesto, na ktorom je daný riešiteľ závisí len od toho, koľkí riešitelia sú nad ním vo výsledkovke. V predošlej výsledkovke sú teda pred týmto plusovým riešiteľom všetci plusoví, čo sú pred ním aj v terajšej (Peter) a zároveň aj všetci mínusoví (Zygro a Mário). Keďže ale v terajšej výsledkovke nie sú pred plusovým všetci mínusoví (lebo jeden mínusový je posledný) tak je v terajšej výsledkovke aj náš plusový riešiteľ určite vyššie, lebo je nad ním menej riešiteľov ako v predošlej.

Pre mínusových riešiteľov vieme podobne dokázať, že v predošlej výsledkovke boli vyššie, pretože sme pod nich umiestnili všetkých plusových riešiteľov (aj toho plusového, ktorý je v terajšej výsledkovke prvý).

Čo s nulovými riešiteľmi?

Je jasné, že nuloví riešitelia museli byť v predošlom kole na rovnakom mieste ako sú teraz (lebo sa ani nezhoršili, ani nezlepšili). Môžeme si teda úlohu zjednodušiť a riešiť ju akoby bez nich. Jednoducho ich pomyselne **vyškrtneme z terajšej výsledkovy** a úlohu vyriešime vyššie uvedeným postupom len pre plusových a mínusových riešiteľov – nazvime toto riešenie *nenulové*.

Riešenie úlohy aj s nulovými riešiteľmi získame tak, že ich najprv umiestnime na ich pôvodné miesta a potom postupne zaplníme zvyšné miesta nenulovými riešiteľmi v takom poradí, v akom boli v nenulovom riešení.

Ak nenulové riešenie neexistuje, nebude existovať ani žiadne riešenie úlohy (predošlá výsledkovka aj s nulovými riešiteľmi). Nutné podmienky vieme totiž prepísať tak, aby platili aj pre riešenie s nulovými riešiteľmi:

- **Prvý nenulový riešiteľ musí byť plusový**. Ak by bol mínusový, musel ho niekto predbehnúť, no pred ním sú len nuloví riešitelia, ktorí tu boli aj predtým.
- **Posledný nenulový riešiteľ musí byť mínusový**. Ak by bol plusový, musel niekoho predbehnúť, no pod ním sú len nuloví riešitelia, ktorí tam boli aj v predošlom kole.

Ako to naprogramovať?

Hneď na začiatku si vytvoríme tri polia reťazcov. Pole mínusových riešiteľov, plusových riešiteľov a pole, kam budeme zapisovať riešenie – predošlú výsledkovku.

Ak pri čítaní vstupu narazíme na nulového riešiteľa, zapíšeme si ho do poľa s riešením na jeho pôvodné miesto. Meno plusového alebo mínusového riešiteľa si zapíšeme do plusového alebo mínusového poľa.

Ďalej potrebujeme overiť dve podmienky. Na to si stačí počas čítania vstupu do premenných zapamätať, aké znamienko bolo pri prvom a poslednom nenulovom riešiteľovi. Po dočítaní vstupu podmienky overíme a program buď vypíše `FAIL` a skončí, alebo bude pokračovať.

Na záver bude stačiť vyplniť zvyšné miesta v riešení. Stačí nám teda raz prejsť pole s riešením a ak je nejaké miesto voľné, priradíme sem (alebo rovno vypíšeme) ďalšieho mínusového alebo (keď sa minú mínusoví) plusového riešiteľa.

Listing programu (Python)

```
n = int(input())
plusovi, minusovi = list(), list()
riesenie = ["" for x in range(n)]

prvy_nenulovy = "0"
posledny_nenulovy = "0"

for i in range(n):
    zmena, meno = input().split()

    if zmena != "0":
        if prvy_nenulovy == "0":
            prvy_nenulovy = zmena
        posledny_nenulovy = zmena

    if zmena == "+":
        plusovi.append(meno)
    elif zmena == "-":
        minusovi.append(meno)
    else:
        riesenie[i] = meno  # nulovych hned pridame na ich povodne miesto

if posledny_nenulovy == "+" or prvy_nenulovy == "-":
    print("FAIL")
    exit(0)

print("OK")
nenulovi = minusovi + plusovi
dalsi_nenulovy = 0

for i in range(n):          # na zvysne miesta postupne pridame minusovych a plusovych
    if riesenie[i] == "":
        riesenie[i] = nenulovi[dalsi_nenulovy]
        dalsi_nenulovy += 1

print("\n".join(riesenie))
```

Zložitosť

Časová zložitosť algoritmu je $O(n)$ – lineárna od počtu ľudí vo výsledkovke, keďže stačí len raz prejsť vstup, a prípadne raz prejsť cez všetky pozície výsledkovky.

Pamäťová zložitosť tohto algoritmu je tiež $O(n)$ – každé z našich troch polí bude obsahovať najviac n reťazcov.

4. Zápolenie o cukríky

Kubo a Mário
(max. 9 b za popis, 6 b za program)

Riešenie hrubou silou

Ako pri väčšine úloh, aj pri tejto sa dá vcelku ľahko vymyslieť riešenie, ktoré vyskúša všetky možnosti zjedenia cukríkov, vypočíta chutnosť každej podpostupnosti a vypíše najväčšiu z nich.

Vezmime si prvý cukrík. Ten v našej výslednej postupnosti buď bude, alebo nebude. Zrátame teda akú najväčšiu chutnosť môže dosiahnuť zvyšok postupnosti ak tento prvý cukrík vezmeme a ak ho nevezmeme, a vyberieme z nich tú lepšiu možnosť. Spravíme si teda rekurzívnu funkciu `rek`, ktorá dostane ako parametre pozíciu momentálne spracovávaného cukríka v postupnosti a typ posledného zjedeného cukríka. Funkcia pomocou samej seba (rekurzívne) spočíta dve hodnoty – chutnosť zvyšku postupnosti podľa toho či momentálne spracovávaný cukrík v postupnosti bude alebo nie – a vyberie možnosť s najväčšou chutnosťou. Maximálnu chutnosť vráti ako výstup. Na výpočet celej úlohy potom stačí zavolať `rek` na prvý cukrík s tým, že predtým nebol zjedený žiaden iný a vypísať hodnotu, ktorú vráti.

Už len odhadneme časovú zložitosť takéhoto riešenia. Pre každý cukrík máme dve možnosti, čo s ním spraviť, zjeme ho alebo ho nezjeme. Všetkých možností, ako vybrať niekoľko cukríkov je preto 2^n, čo bude aj časová zložitosť nášho riešenia – $O(2^n)$.

Za takéto riešenie bolo možné získať 1 bod.

Listing programu (C++)

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

using namespace std;

vector <int> cukriky;
vector <int> chutnost;
int cena_nova;

int rek (int pozicia_aktualneho, int typ_predosleho) {
    // už sme pojedli všetky cukríky
    if (pozicia_aktualneho == cukriky.size()) return 0;

    int typ_aktualneho = cukriky[pozicia_aktualneho];
    int chutnost_aktualneho = chutnost[typ_aktualneho];

    int zobrali_sme_aktualny = rek(pozicia_aktualneho+1, typ_aktualneho);
    int nezobrali_sme_aktualny = rek(pozicia_aktualneho+1, typ_predosleho);

    if (typ_predosleho == typ_aktualneho){
        return max(nezobrali_sme_aktualny, zobrali_sme_aktualny + chutnost_aktualneho);
    } else {
        return max(nezobrali_sme_aktualny, zobrali_sme_aktualny + cena_nova);
    }
}

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

    cukriky.resize(n);
    for (int i=0; i<n; i++) {
        cin >> cukriky[i];
        cukriky[i]--;
    }

    chutnost.resize(m);
    for (int i=0; i<m; i++) {
        cin >> chutnost[i];
    }

    cin >> cena_nova;

    // zavoláme sa na prvý cukrík, predchádzajúci zjedený je žiadny
    cout << rek(0, -1) << endl;
    return 0;
}
```

Dynamické programovanie

Poďme teda vymyslieť niečo rýchlejšie. Takéto rekurzívne riešenie totiž zbytočne veľakrát počíta stále tie isté hodnoty: Vezmime si postupnosť cukríkov `1 1 1 1`. Už pri takto malom vstupe trikrát počítame hodnotu funkcie `rek` pre tretí cukrík, ak predchádzajúci bol typu 1 – raz pre možnosť, kedy zoberieme iba prvý cukrík, raz keď zoberieme iba druhý a raz keď zoberieme oba.

Hodnota, ktorú vráti funkcia `rek(2, 1)` (2 prislúcha tretiemu cukríku, lebo indexujeme od 0) sa však nezmení bez ohľadu na to, ktorú z týchto možností sme zvolili. Ak by sme sa dokázali vyhnúť tomu, aby sme počítali tú istú vec duplicitne, naše riešenie by sa určite zrýchlilo.

Pri rekurzívnych funkciách existuje spôsob ako niečo takéto docieliť – volá sa memoizácia. V podstate spočíva v tom, že po vypočítaní nejakej hodnoty si zapamätáme tento výsledok a nabudúce, keď náš program bude potrebovať spočítať túto hodnotu, iba si ju nájde v pamäti. Viac sa o memoizácii môžete dozvedieť napríklad v tomto vzorovom riešení: https://www.ksp.sk/ulohy/riesenia/1098/

Použitím memoizácie na vyššie ukázanú rekurziu by sme dostali riešenie s časovou zložitosťou $O(nm)$, za ktoré by sme dostali 4 body. Toto riešenie však následne už nevieme zlepšiť, preto sa na náš problém musíme pozrieť z inej strany.

Pokúsme sa vyriešiť takúto úlohu: Akú najväčšiu chutnosť vie mať postupnosť končiaca cukríkom na i-tej pozícii? Túto najväčšiu možnú chutnosť si označme $D[i]$.

Keďže vieme, ktorý cukrík zjeme ako posledný (i-ty), možno má zmysel pozrieť sa na to, ktorý sme zjedli ako predposledný. Ten totiž ovplyvní, ako nám bude chutiť ten na pozícii i. Nech je predposledný zjedený cukrík na pozícii j. Potom na základe toho, či sú i-ty a j-ty cukrík rovnakého typu vieme povedať, ako veľmi nám chutil ten na pozícii i.

Všimnime si však, že i-ty cukrík, keďže je posledný, nám neovplyvní ako nám chutili predošlé cukríky v postupnosti. Preto ak chceme mať čo najväčšiu chutnosť a vieme, že na predposlednej pozícii je j-ty cukrík, najväčšia chutnosť postupnosti končiacej j-tym cukríkom je $D[j]$.

Preto platí, že:

- Ak je i-ty cukrík rovnakého typu ako j-ty, najväčšia možná chutnosť postupnosti, ktorá končí týmito dvoma cukríkmi je $D[i] = D[j] + chutnost[typ[i]]$
- Ak sú dva posledné cukríky rôznych typov, chutnosť postupnosti je $D[i] = D[j] + c$ (kde c je chutnosť cukríka ak nasleduje po cukríku iného typu)

My síce nevieme, ktorý cukrík bol predposledný, teda čo je tá pozícia j, nič nám však nebráni, aby sme vyskúšali všetky možnosti medzi 0 až $i-1$ a z nich si vybrali tú najlepšiu. Zapísané ako vzorec:

$$rovnakypredosly = chutnost[typ[i]] + \max\{D[j] \mid j \leftarrow 0, 2, \ldots, i-1; typ[i] = typ[j]\}$$
$$inypredosly = c + \max\{D[j] \mid j \leftarrow 0, 2, \ldots, i-1; typ[i] \neq typ[j]\}$$
$$D[i] = \max\{rovnakypredosly, inypredosly\}$$

Vyrobíme si teda pole `D[]`, v ktorom budeme mať pre každý cukrík uloženú **najväčšiu chutnosť postupnosti, akú vieme dosiahnuť, ak ho vezmeme ako posledný**. Keďže pri výpočte hodnoty $D[i]$ potrebujeme poznať všetky hodnoty $D[j]$ pre $j < i$, budeme hodnoty tohto poľa vypĺňať zaradom od prvej pozície. Takýto prístup, kde z menších hodnôt postupne počítame čím ďalej väčšie hodnoty, sa nazýva *dynamické programovanie*.

Keď vyplníme celé pole, tak výsledok bude maximálna hodnota v ňom, a tú vypíšeme.

Pre spočítanie jednej hodnoty $D[i]$ sa potrebujeme pozrieť na predošlých $i-1$ políčok. Celkovo sa preto pozrieme na $0+1+2+\ldots+(n-1) = \frac{(n-1)n}{2}$ políčok, a časová zložitosť tohto algoritmu bude $O(n^2)$. Za takéto riešenie ste na testovacích vstupoch mohli získať 3 body.

Listing programu (C++)

```
#include <cstdio>
#include <algorithm>
#include <vector>

using namespace std;

int n, m, c;
vector<int> cukriky;
vector<int> chutnosti;

vector<long long> D; // D[pozicia posledneho] -> cena

int main(){
    scanf("%d%d", &n, &m);
    cukriky.resize(n);
    chutnosti.resize(m);

    for(int i=0; i<n; i++){
        scanf("%d", &cukriky[i]);
        cukriky[i]--;
    }

    for(int i=0; i<m; i++)
        scanf("%d", &chutnosti[i]);

    scanf("%d", &c);

    D.resize(n, 0LL);
```

```
    for(int i=0; i<n; i++){
        long long najlepsie = c;
        for(int j=0; j<i; j++)
            if (cukriky[i] == cukriky[j])
                najlepsie = max(najlepsie, D[j] + (long long)(chutnosti[cukriky[i]]));
            else
                najlepsie = max(najlepsie, D[j] + (long long)(c));
        D[i] = najlepsie;
    }

    long long maxi = 0;
    for(int i=0; i<n; i++)
        maxi = max(maxi, D[i]);
    printf("%Ld\n", maxi);
}
```

Jemné vylepšenie

V predchádzajúcom riešení si môžeme všimnúť, že pri počítaní $D[i]$ prechádzame všetkých $i-1$ políčok poľa D len preto, aby sme spomedzi nich vybrali buď najväčšie $D[j]$, kde cukrík j má rovnaký typ ako cukrík i alebo najväčšie $D[j]$ zo všetkých ostatných.

Predstavme si, že dva cukríky na pozícii a a b sú rovnakého typu a máme pre ne vypočítané hodnoty $D[a]$ a $D[b]$, pričom platí $D[a] < D[b]$. Uvedomme si, že pri počítaní ďalších hodnôt $D[i]$ sa budeme vždy pozerať na obe tieto pozície, budú patriť do tej istej skupiny (rovnaký alebo rôzny ako i-ty cukrík) a vždy z nich budeme vyberať maximum. Hodnotu $D[a]$ teda už **nikdy** viac nebudeme potrebovať, lebo hodnota $D[b]$ je proste lepšia.

Bude nám preto stačiť, ak si pre každý typ cukríka zapamätáme len tú najväčšiu hodnotu $D[i]$, ktorú sme dovtedy videli ak i-ty cukrík bol príslušného typu.

Budeme mať teda pole `T[]` veľkosti m, kde na k-tej pozícii bude uložená **najväčšia doteraz videná chutnosť postupnosti, ktorá končí cukríkom typu** k. Hodnotu $D[i]$ potom budeme počítať tak, že buď zoberiem hodnotu postupnosti končiacej rovnakým typom cukríka, teda hodnotu $T[typ[i]]$ alebo maximum zo všetkých ostatných.

V našom programe dokonca vynechávame samotné pole `D[]`, namiesto hodnoty `D[i]` používame iba lokálnu premennú `najlepsie`. Všimnite si tiež, že na záver nezabudneme hodnotu `najlepsie` vložiť do poľa `T[typ[i]]` – samozrejme, ak je väčšia ako doterajšia hodnota.

Takéto riešenie má zložitosť $O(nm)$ a na testovacích vstupoch dostalo 4 body.

Listing programu (C++)

```
...

vector<long long> T; // T[typ] -> najlepsia cena

...

T.resize(m, -1LL);
for(int i=0; i<n; i++){
    long long najlepsie = c;
    for(int j=0; j<m; j++)
        if (T[j] == -1LL)
            najlepsie = max(najlepsie, (long long)(c));
        else if (cukriky[i] == j)
            najlepsie = max(najlepsie, T[j] + (long long)(chutnosti[j]));
        else
            najlepsie = max(najlepsie, T[j] + (long long)(c));
    T[cukriky[i]] = max(T[cukriky[i]], najlepsie);
}

...
```

Ešte o rád rýchlejšie

Označme si najväčšiu chutnosť spomedzi cukríkov iných typov než typu $k = typ[i]$ ako:

$$M_k = \max\{T[1], T[2], \ldots T[k-1], T[k+1], \ldots T[m]\}$$

Hodnotu $D[i]$ sme teda v predošlom riešení počítali ako $\max\{T[k] + chutnost[k], M_k + c\}$.

Strácali sme ale veľa času tým, že sme M_k vždy hľadali v čase $O(m)$ – prezeraním všetkých ostatných typov. Takmer vždy nám ale stačí pamätať si len najväčšiu chutnosť zo všetkých, $M = \max\{T[1], T[2], \ldots T[m]\}$. Chutnosť postupnosti, ktorá končí i-tym cukríkom, ktorý má typ k vieme teda spočítať ako $T[k] = \max\{T[k] + chutnost[k], M + c\}$.

Hodnota M však patrí niektorému typu cukríkov. A ak je i-ty cukrík rovnakého typu, tak neplatí, že $M_{typ[i]} = M$ ($M_{typ[i]}$ je maximum z typov *iných* ako $typ[i]$). V takomto prípade by sme museli spočítať hodnotu $M_{typ[i]}$ a použiť tú.

V skutočnosti ju ale nemusíme počítať, stačí si zapamätať **typ cukríka, ktorým končí druhá doteraz najchutnejšia postupnosť** a použiť ju namiesto $M_{typ[i]}$.

V našom riešení si teda budeme pamätať pole pre chutnosti jednotlivých typov, `T[]`, a dva typy s najväčšími chutnosťami a a b, pričom $T[a] \geq T[b]$.

- Pri výpočte chutnosti postupnosti, ktorá končí i-tym cukríkom sa pozrieme na postupnosť, ktorá končila *rovnakým typom*, a jedna možnosť výsledku bude $T[typ[i]] + chutnost[typ[i]]$.
- Druhá možnosť výsledku je taká, že zoberieme najchutnejšiu postupnosť, ktorá končí *iným typom* cukríka ako $typ[i]$ a potom je možnosť výsledku $M + c$, pričom M bude $T[a]$ ak $typ[i] \neq a$ a $T[b]$ v opačnom prípade.

Z týchto dvoch možností vyberieme ako výsledok tú chutnejšiu.

Časová zložitosť riešenia je teda $O(n)$, lebo pre každý cukrík overíme len pár podmienok a upravíme dva najvýhodnejšie typy.

Listing programu (Python)

```
n, m = [int(x) for x in input().split()]
cukriky = [int(x)-1 for x in input().split()]
chutnosti = [int(x) for x in input().split()]
c = int(input())

T = [-1] * m + [0]

najlepsi_typ = m
druhy_najlepsi_typ = m

for typ_i in cukriky:
    if typ_i != najlepsi_typ:
        M = T[najlepsi_typ]
    else:
        M = T[druhy_najlepsi_typ]

    if T[typ_i] == -1:
        T[typ_i] = M + c
    else:
        T[typ_i] = max(T[typ_i] + chutnosti[typ_i], M + c)

    if typ_i != najlepsi_typ:
        if T[typ_i] > T[najlepsi_typ]:
            druhy_najlepsi_typ = najlepsi_typ
            najlepsi_typ = typ_i
        elif T[typ_i] > T[druhy_najlepsi_typ]:
            druhy_najlepsi_typ = typ_i

print(max(*T))
```

Mišo

5. Oblepené mesto

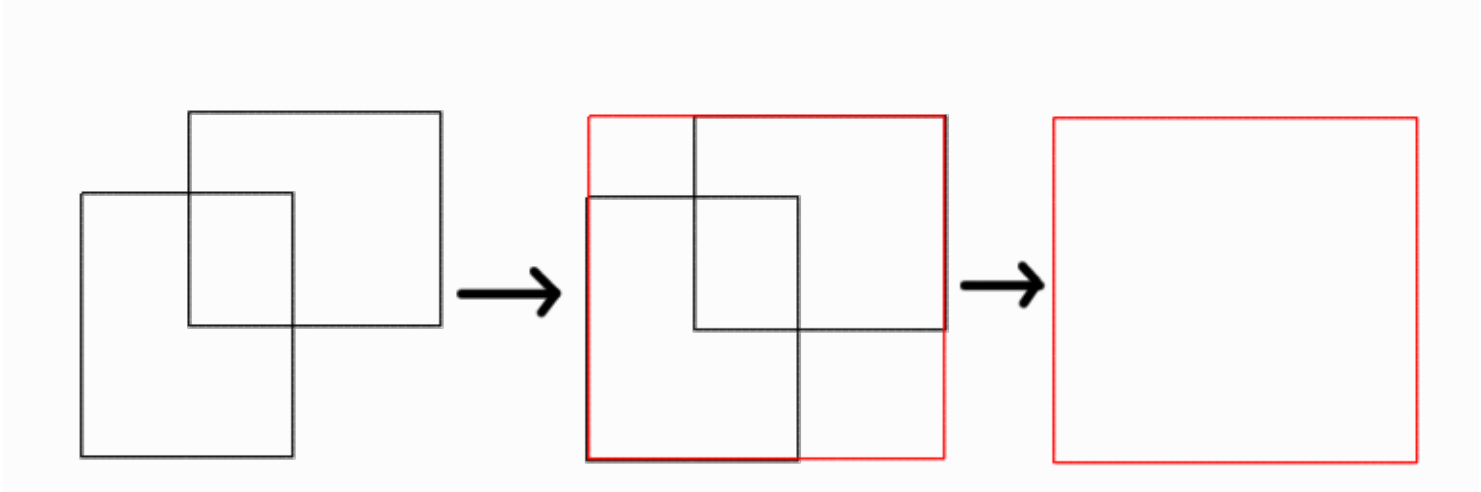
(max. 9 b za popis, 6 b za program)

V tomto popise vzorového riešenia budeme používať nasledovné názvoslovie:

- *Nutný obdĺžnik* = obdĺžnik, ktorý musíme celý prekryť – určený číslami x, y, w, h zo vstupu.
- *Postačujúci obdĺžnik* = najmenší obdĺžnik, ktorý pokrýva nutný obdĺžnik a zároveň spĺňa podmienky zadania – správna odpoveď a teda určený číslami x_p, y_p, w_p, h_p.
- *Pumpovací obdĺžnik* = riešenie bude vždy spočívať v tom, že začneme s nutným obdĺžnikom a budeme ho rozťahovať až kým nedostaneme postačujúci obdĺžnik. Obdĺžnik v procese rozťahovania budeme volať pumpovací obdĺžnik.

Pozorovanie

Najdôležitejšie pozorovanie v celej tejto úlohe je toto: k správnemu riešeniu sa vždy dostaneme nasledovným spôsobom. Za pumpovací obdĺžnik si najprv zoberieme nutný obdĺžnik. Kým sa pumpovací obdĺžnik čiastočne prekrýva s nejakým plagátom, tak pumpovací obdĺžnik roztiahneme tak, aby ten plagát akurát celý prekrýval. Lepšie je to objasnené na nasledujúcom obrázku:



Po roztiahnutí sa môže stať, že sa pumpovací obdĺžnik začne prekrývať s nejakým ďalším plagátom. Tento postup teda opakujeme až kým sa pumpovací obdĺžnik neprekrýva čiastočne so žiadnym plagátom. Vtedy sa pumpovací obdĺžnik rovná postačujúcemu obdĺžniku.

Pomalé riešenie

Toto riešenie bude spočívať v naivnej implementácii spomenutého pozorovania. Najprv si pumpovací obdĺžnik nastavíme na nutný. Potom prejdeme cez všetky plagáty a zistíme, či sa nejaký z nich čiastočne neprekrýva s pumpovacím obdĺžnikom. Ak áno, tak pumpovací obdĺžnik roztiahneme tak, aby akurát pokrýval daný plagát. Tento postup opakujeme, kým dochádza k nejakým zmenám. Keď prestane dochádzať k zmenám, našli sme správne riešenie.

Objasnime si, čo znamená "roztiahnuť obdĺžnik tak, aby akurát pokrýval daný plagát". Znamená to vytvoriť taký obdĺžnik, ktorý súčasne pokrýva pôvodný obdĺžnik aj plagát a je zároveň čo najmenší. Vieme ho vypočítať takouto funkciou:

Listing programu (Python)

```
def roztiahni(x1, y1, w1, h1, x2, y2, w2, h2):
    left1 = x1
    right1 = x1 + w1
    top1 = y1
    bottom1 = y1 + h1

    left2 = x2
    right2 = x2 + w2
    top2 = y2
    bottom2 = y2 + h2

    left = min(left1, left2)
    right = max(right1, right2)
    top = min(top1, top2)
    bottom = max(bottom1, bottom2)

    x = left
    y = top
    w = right - left
    h = bottom - top

    return x, y, w, h
```

Funkcia berie ako argumenty dva obdĺžniky. Prvý je zadaný číslami x_1, y_1, w_1, h_1 a druhý číslami x_2, y_2, w_2, h_2. Výsledkom je obdĺžnik, ktorého ľavý okraj je tam, kde je ľavý okraj "ľavejšieho" z obdĺžnikov, pravý tam kde je okraj "pravejšieho" a podobne pre zvyšné prípady.

Pozrime sa na časovú a pamäťovú zložitosť tohto riešenia. Existujú vstupy, v ktorých sa na začiatku nutný obdĺžnik čiastočne prekrýva len s jedným plagátom. Hneď ako ho pokryje, prekryje sa s jedným ďalším a tak ďalej, až kým postupne pokryje všetky plagáty. V tomto riešení potrebujeme s každým novým pokrytím preiterovať cez všetky plagáty, aby sme zistili, s ktorým sa pumpovací obdĺžnik prekrýva. Keďže postupne pokryjeme všetkých n plagátov a pri každom potrebujeme preiterovať cez všetkých n plagátov, časová zložitosť je $O(n^2)$. Pamäťová zložitosť je $O(n)$, keďže si potrebujeme pamätať len pozície a rozmery všetkých obdĺžnikov.

Rýchle riešenie

Počet plagátov v tejto úlohe môže byť až 100 000 a tak je predchádzajúce riešenie príliš pomalé. Pokúsime sa ho teda zrýchliť.

Najprv si uvedomme, že na každý plagát narazíme buď sprava, zľava, zvrchu, alebo zospodu (teda: najprv sa pumpovací obdĺžnik s daným plagátom neprekrýva, potom sa pumpovací obdĺžnik roztiahne doľava, čo spôsobí, že sa prekryje s nejakým iným plagátom; v tomto prípade naň narazil sprava).

Ďalej je dôležité si uvedomiť, v akom poradí budeme na jednotlivé plagáty narážať. Majme nejaké dva plagáty. Pravý okraj prvého nech je ďalej od nutného obdĺžnika, ako pravý okraj druhého. Ak na oba tieto obdĺžniky narazíme zľava, tak zaručene musíme najprv naraziť na druhý, až potom na prvý, lebo je ďalej.

Z tohto je zrejmé, v akom poradí budeme na jednotlivé plagáty narážať. Najprv narazíme na tie, ktorých okraje sú bližšie, až potom na tie, ktorých okraje sú ďalej.

Plagáty si teda usporiadame osobitne podľa ľavých, pravých, horných a spodných okrajov. Následne si v každom z týchto usporiadaných polí budeme udržiavať ukazovateľ (index), ktorý určuje, kde sa nachádza daný okraj pumpovacieho obdĺžnika.

Na začiatku budú tieto indexy ukazovať na okraje nutného obdĺžnika.

Ďalej si budeme v každom smere udržiavať ďalší ukazovateľ, ktorý nazvime "naťahovací". Tieto ukazovatele budeme posúvať vždy, keď sa pumpovací obdĺžnik v niektorom smere čiastočne prekrýva s nejakým plagátom.

Ukazovatele pumpovacieho obdĺžnika budú "dobiehať" naťahovacie ukazovatele. Pri každom posunutí ukazovateľov pumpovacieho obdĺžnika narazíme na okraje nejakého nového plagátu. Skontrolujeme teda, či sa s ním pumpovací obdĺžnik čiastočne neprekrýva a ak áno, tak zase posunieme naťahovacie indexy.

Takýmto spôsobom postupne napumpujeme pumpovací obdĺžnik z nutného na postačujúci a tak nájdeme správne riešenie.

Pozrime sa ešte na časovú a pamäťovú zložitosť tohto riešenia. Na začiatku si musíme zoradiť okraje plagátov, čo nám dáva časovú zložitosť najmenej $O(n \log n)$. V nasledujúcej fáze postupne posúvame ukazovatele. Každý okraj nejakého plagátu však každým ukazovateľom navštívime najviac raz, a tak je časová zložitosť tejto fázy $O(n)$. Celková časová zložitosť je teda $O(n \log n)$. Pamäťová zložitosť je $O(n)$, pretože aj v tomto riešení si musíme len pamätať pozície a rozmery jednotlivých plagátov a nič navyše.

Listing programu (C++)

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

using namespace std;

struct rect {
    int x1, y1, x2, y2;
};

rect cover(rect r1, rect r2) {
    return {
        min(r1.x1, r2.x1),
        min(r1.y1, r2.y1),
        max(r1.x2, r2.x2),
        max(r1.y2, r2.y2),
    };
}

bool overlap(rect r1, rect r2) {
    int left = r2.x2 - r1.x1;
    int right = r2.x1 - r1.x2;
    int top = r2.y2 - r1.y1;
    int bottom = r2.y1 - r1.y2;
    return !(left <= 0 || right >= 0 || top <= 0 || bottom >= 0);
}

struct edge {
    int pos, rect_index;

    bool operator < (const edge &other) const {
        return pos < other.pos;
    }
};

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

    vector<rect> rects;
    vector<edge> x_edges;
    vector<edge> y_edges;

    for (int i = 0; i <= n; i++) {
        int x, y, w, h;
        cin >> x >> y >> w >> h;

        rect r = {x, y, x+w, y+h};
        rects.push_back(r);
        x_edges.push_back({r.x1, i});
        x_edges.push_back({r.x2, i});
        y_edges.push_back({r.y1, i});
        y_edges.push_back({r.y2, i});
```

```
    }

    sort(x_edges.begin(), x_edges.end());
    sort(y_edges.begin(), y_edges.end());

    rect target = rects.back();

    rect bounding_idx = {
        (int)(lower_bound(x_edges.begin(), x_edges.end(), edge{target.x1, -1}) - x_edges.begin()),
        (int)(lower_bound(y_edges.begin(), y_edges.end(), edge{target.y1, -1}) - y_edges.begin()),
        (int)(lower_bound(x_edges.begin(), x_edges.end(), edge{target.x2, -1}) - x_edges.begin()),
        (int)(lower_bound(y_edges.begin(), y_edges.end(), edge{target.y2, -1}) - y_edges.begin()),
    };

    for (rect r: rects) {
        if (overlap(target, r)) {
            target = cover(target, r);
        }
    }

    vector<rect> overlaps;

    do {
        overlaps.clear();

        rect bounding = {
            x_edges[bounding_idx.x1].pos,
            y_edges[bounding_idx.y1].pos,
            x_edges[bounding_idx.x2].pos,
            y_edges[bounding_idx.y2].pos,
        };

        if (bounding.x1 != target.x1) {
            bounding_idx.x1--;
            overlaps.push_back(rects[x_edges[bounding_idx.x1].rect_index]);
        }
        if (bounding.y1 != target.y1) {
            bounding_idx.y1--;
            overlaps.push_back(rects[y_edges[bounding_idx.y1].rect_index]);
        }
        if (bounding.x2 != target.x2) {
            bounding_idx.x2++;
            overlaps.push_back(rects[x_edges[bounding_idx.x2].rect_index]);
        }
        if (bounding.y2 != target.y2) {
            bounding_idx.y2++;
            overlaps.push_back(rects[y_edges[bounding_idx.y2].rect_index]);
        }

        for (rect &r : overlaps) {
            if (overlap(target, r)) {
                target = cover(target, r);
            }
        }
    } while (int(overlaps.size()) > 0);    // kym sa target meni

    int x = target.x1;
    int y = target.y1;
    int w = target.x2 - target.x1;
    int h = target.y2 - target.y1;

    cout << x << " " << y << " " << w << " " << h << endl;
}
```

Zajo

6. Obnova bytového parku

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

V úlohe hľadáme dva výsledky, ktoré aj vypisujeme – súvislú oblasť veľkosti k a najmenšiu možnú výšku najvyššej budovy v takejto oblasti – označme ju H. Ak ale poznáme jeden z týchto dvoch výsledkov, tak ten druhý sa k nemu dá dopočítať celkom ľahko.

Ak poznáme "ideálnu výšku", stačí postupne skúšať prehľadávať mesto do šírky alebo do hĺbky[3]. Zo všetkých (ešte nenavštívených) políčok postupne spúšťame prehľadávania, kým nenájdeme oblasť veľkosti aspoň k. Prehľadávanie navyše navštevuje len také susedné políčka, ktorých budovy sú nanajvýš tak vysoké ako "ideálna výška" H.

Naopak, keď poznáme tú správnu oblasť, poľahky vyrátame, aká je v nej tá najväčšia výška. Stačí prejsť všetky políčka danej oblasti a zapamätať si výšku toho najvyššieho z nich.

Toto umožňuje dva prístupy k príkladu: prvý, v ktorom sa snažíme nájsť tú "ideálnu" výšku a druhý, kedy sa pokúšame skonštruovať nejakú správnu oblasť. V tomto vzoráku sa budeme zaoberať tým prvým prístupom a na konci si len krátko spomenieme, ako by sa dalo na riešenie prísť z druhej strany.

[3]Ak sú vám tieto pojmy cudzie, na internete nájdete množstvo materiálov na témy *breadth-first-search* a *depth-first-search*. V češtine/slovenčine si môžete pozrieť napríklad časť kuchárky českého KSP alebo poznámky k prednáške zo sústredenia Prask (od str. 20).

Jednoduché hľadanie najmenšej výšky

Postupne budeme skúšať pre všetky možné výšky h (od najmenšej po najväčšiu) nasledovný postup:

1. Vieme nájsť oblasť veľkosti aspoň k, kde je najvyššia budova vysoká nanajvýš h? (Toto vieme zistiť práve spomínanými prehľadávaniami.)
2. Ak nie, skúsme to isté, pre výšku o jedna väčšiu.

Takýmto spôsobom po chvíli prídeme k tej najmenšej vyhovujúcej výške – k výške H. Na záver len vypíšeme jej prislúchajúcu oblasť a máme vyhrané.

Akú má toto riešenie časovú zložitosť? Robíme prehľadávanie na mriežke rozmerov $r \times s$ pre výšky, až kým nenájdeme tú správnu. V najhoršom prípade bude riešenie potrebovať $O(r \cdot s \cdot h_{max})$ operácií, čo je podľa zadania najviac $10^6 \cdot 10^9 = 10^{15}$. Je to síce oveľa viac, než si v bežnom programe môžeme dovoliť, no riešenie zvládne vyriešiť menšie vstupy a aj vstupy, kde je h_{max} malé.

Pamäťová zložitosť algoritmu je $O(r \cdot s)$, keďže si musíme pamätať výšku každej budovy a aj to, ktoré políčka sme pri prehľadávaní navštívili.

Ako toto riešenie zlepšiť? Naša úloha má príjemnú vlastnosť, ktorú sme zatiaľ nepoužili: Pre všetky výšky menšie ako ideálna výška H dostatočne veľkú oblasť nájsť nevieme, a naopak, pre všetky výšky väčšie ako H takú oblasť určite nájsť vieme.

Toto nám tvorí ideálne podmienky pre použitie binárneho vyhľadávania.

Binárne hľadanie najmenšej výšky H

Myšlienka binárneho vyhľadávania je veľmi jednoduchá. Pre nejakú hodnotu výšky h sa pýtame otázku: Vieme nájsť súvislú oblasť veľkosti aspoň k, kde je najväčšia budova vysoká nanajvýš h?

- Ak áno, hľadaná výška je $\leq h$.
- Ak nie, hľadaná výška je $> h$.

Na začiatku si teda zvolíme interval, v ktorom budeme vyhľadávať: `(left, right] = (-1, max_h]`. Jeho *ľavá hranica* bude označovať najväčšiu výšku, o ktorej vieme, že sa pre ňu *nedá* nájsť dostatočne veľká oblasť. Opačne, *pravá hranica* intervalu bude označovať najnižšiu výšku, o ktorej vieme, že sa pre ňu *dá* nájsť dostatočne veľká oblasť. Následne sa opakovane pýtame danú otázku pre výšku $h = (right+left)/2$ – výšku presne uprostred intervalu.

- Ak sa dá nájsť dostatočne veľká oblasť, posunieme pravú hranicu intervalu na h.
- Ak sa nedá nájsť dostatočne veľká oblasť, posunieme ľavú hranicu intervalu na h.

Prehľadávanie skončíme, ak zostane v intervale len jediná hodnota – jeho pravá hranica – výška H.

To, že sme si zvolili práve polootvorený interval (jeden z jeho hraničných prvkov v ňom nie je) nie je náhoda. Vďaka takejto voľbe sa vieme ľahko rozhodnúť, kedy kam posunúť ktorú hranicu – predsa tak aby zostali zachované definície `left` a `right`. Tiež sa vyhneme rôznym chybám, keďže pivot (prostredný prvok – h) nám rozdelí interval opäť na dva polootvorené – `(left, h]` a `(h, right]` – také, že ich prienik je prázdny a ich zjednotením dostaneme práve `(left, right]`. Vďaka tomu žiadnu možnosť nikdy nebudeme uvažovať dvakrát a žiadnu tiež nikdy nevynecháme. Ako bonus dostávame, že dĺžka intervalu `(left, right]` je práve `right-left`.

Listing programu (C++)

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

#define INF 1000000000

using namespace std;

int zmena_riadku[4] = {-1, 0, 1, 0};
int zmena_stlpca[4] = {0, -1, 0, 1};

int cislo_pokusu = 0;

vector<vector<int> > vysky;
vector<vector<int> > naposledy_navstivene;

// pokusime sa zafarbit oblast z policka (zacni_r, zacni_s), pricom navstivene
```

```
// policka musia mat vysku najviac `vyska` a oblast bude mat najviac plochu `max_plocha`
int skus_bfs(int vyska, int zacni_r, int zacni_s, int max_plocha = INF) {
    cislo_pokusu++;
    queue<int> q;
    q.push(zacni_r); q.push(zacni_s);
    naposledy_navstivene[zacni_r][zacni_s] = cislo_pokusu;
    int najdena_velkost = 1;

    while (!q.empty()) {
        int riadok = q.front(); q.pop();
        int stlpec = q.front(); q.pop();

        for (int smer = 0; smer < 4; smer++) {
            int vedla_r = riadok + zmena_riadku[smer];
            int vedla_s = stlpec + zmena_stlpca[smer];

            bool prilis_vysoko = (vysky[vedla_r][vedla_s] > vyska);
            bool uz_navstivene = (naposledy_navstivene[vedla_r][vedla_s] == cislo_pokusu);
            if (prilis_vysoko || uz_navstivene)
                continue;

            q.push(vedla_r); q.push(vedla_s);
            naposledy_navstivene[vedla_r][vedla_s] = cislo_pokusu;
            najdena_velkost++;

            if (najdena_velkost == max_plocha)
                return najdena_velkost;
        }
    }
    return najdena_velkost;
}

bool da_sa_vyska(int vyska, int velkost_riesenia, bool presna_velkost = false) {
    int prvy_pokus_tejto_vysky = cislo_pokusu + 1;

    for (int r = 0; r < vysky.size(); r++) {
        for (int s = 0; s < vysky[r].size(); s++) {
            bool prilis_vysoko = (vysky[r][s] > vyska);
            bool uz_navstivene = (naposledy_navstivene[r][s] >= prvy_pokus_tejto_vysky);
            if(prilis_vysoko || uz_navstivene)
                continue;

            int zafarbena_plocha = skus_bfs(vyska, r, s, ((presna_velkost) ? velkost_riesenia : INF));

            if (zafarbena_plocha >= velkost_riesenia)
                return true;
        }
    }
    return false;
}

int main() {
    int velkost_riesenia, velkost_r, velkost_s;
    cin >> velkost_riesenia >> velkost_r >> velkost_s;
    vysky.resize(velkost_r+2, vector<int>(velkost_s+2, INF));
    naposledy_navstivene.resize(velkost_r+2, vector<int>(velkost_s+2, 0));

    for (int r = 1; r < velkost_r+1; r++) {
        for (int s = 1; s < velkost_s+1; s++) {
            cin >> vysky[r][s];
        }
    }

    // Riesenie sa nachadza v intervale (left, right>
    // left - navyssia vyska o ktorej isto vieme, ze nie je riesenim
    // right - najnizsia vyska o ktorej vieme, ze je riesenim
    int left = -1, right = INF;

    // Binarne vyhladavame najnizsiu vysku
    while (right - left > 1) {
        int piv = (left + right) / 2;
        if (da_sa_vyska(piv, velkost_riesenia)) {
            right = piv;
        } else {
            left = piv;
        }
    }
    int min_vyska = right;

    // Oznacime si oblast presne s velkostou velkost_riesenia
    da_sa_vyska(min_vyska, velkost_riesenia, true);

    // Vypisujeme najdene riesenie
    cout << min_vyska << endl;
    for (int r = 1; r < velkost_r+1; r++) {
        for (int s = 1; s < velkost_s+1; s++) {
            if (naposledy_navstivene[r][s] == cislo_pokusu) {
                cout << "*";
            } else {
                cout << ".";
            }
        }
        cout << endl;
    }

    return 0;
}
```

Časová zložitosť nám klesla na $O(r \cdot s \cdot \log(h_{max}))$, lebo pri vyhľadávaní sa vždy spýtame najviac $\log_2(dlzka\ intervalu)$ otázok. Pamäťová zložitosť ostáva $O(r \cdot s)$, lebo si pamätáme tie isté údaje ako v jednoduchšom riešení.

Pohľad z opačnej strany

Bez väčších omáčok si vysvetlime, ako sa na to dalo dívať inak. Predstavíme si, že v meste ešte nie je postavená žiadna budova. Preto ich tam začneme postupne pridávať od najmenšej budovy po najvyššiu. Popri tom budeme sledovať, či po pridaní budovy nevznikla súvislá oblasť budov veľkosti aspoň k. Ak vznikla, tvrdíme, že táto oblasť je samotným riešením problému.

Dokážme si to sporom. Ak by existovala nejaká iná dostatočne veľká súvislá oblasť, ale najvyššia budova v nej by bola nižšia, tak to znamená, že túto oblasť sme museli nájsť už skôr. Keďže sme ale pridávali budovy od najnižšej, tak sme museli pridať už všetky budovy tvoriace túto oblasť. Takúto oblasť by sme teda už objavili skôr, čo je v spor s tvrdením, že pridaním poslednej budovy vznikla dostatočne veľká oblasť prvýkrát.

Potrebujeme ešte vyriešiť, ako zisťovať, či už existuje súvislá oblasť veľkosti aspoň k. Na také niečo je ako stavaný algoritmus Union-find, pri ktorom si udržujeme súvislé oblasti budov. Vždy keď pridávame novú budovu, zistíme, do ktorej oblasti ju máme pridať a či prípadne nespojila nejaké skupiny dokopy. Pre jednotlivé oblasti si pamätáme ich veľkosti a ak má nejaká z nich veľkosť aspoň k, môžeme náš algoritmus zastaviť.

Takéto riešenie je rovnako dobré ako binárne vyhľadávanie najmenšej výšky, keďže musíme usporiadať všetkých $r \cdot s$ budov podľa veľkosti. Samotný algoritmus union-find už nespotrebuje viac času ako toto triedenie a preto je výsledná časová zložitosť $O(r \cdot s \cdot log(r \cdot s))$ a pamäťová zložitosť ostáva $O(r \cdot s)$.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
#include <algorithm>
#include <queue>

using namespace std;

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

int r, c, k;

int id_podla_pozicie(int y, int x){return y*c + x;}

void pozicia_podla_id(int id, int &y, int &x){y = id / c; x = id % c;}

vector<vector<int> > mapa_vysok;
vector<pair<int, pair<int,int> > > policka_od_najnizsieho;

vector<int> otec, hlbka_komponentu, velkost;

void nacitaj_vstup(){
    scanf("␣%d␣%d␣%d",&k, &r, &c);
    mapa_vysok.resize(r, vector<int>(c, 0));
    otec.resize(r*c, 0);
    velkost.resize(r*c, 1);

    for(int y=0; y<r; y++){
        for(int x=0; x<c; x++){
            int vyska;
            scanf("␣%d", &vyska);
            mapa_vysok[y][x] = vyska;
            policka_od_najnizsieho.push_back({vyska, {y, x}});
            int id = id_podla_pozicie(y, x);
            otec[id] = id;
        }
    }
}

int sef(int a){
    if (otec[a] == a) return a;
    return otec[a] = sef(otec[a]);
}

int spoj(int a, int b){
    int vacsi = sef(a), mensi = sef(b);
    if (vacsi == mensi) return velkost[vacsi];

    if (velkost[vacsi] < velkost[mensi]) swap(vacsi, mensi);

    otec[mensi] = vacsi;
    velkost[vacsi] += velkost[mensi];
    return velkost[vacsi];
}

void najnizsi_komponent_velkosti_k(int &id_komponentu, int &vyska){
    for(int i=0; i<r*c; i++){
        vyska = policka_od_najnizsieho[i].first;
        int y = policka_od_najnizsieho[i].second.first;
        int x = policka_od_najnizsieho[i].second.second;
        int id1 = id_podla_pozicie(y, x);
```

```
        for(int j=0; j<4; j++){
            int nx = x + dx[j];
            int ny = y + dy[j];
            if (nx < 0 || nx >= c || ny < 0 || ny >= r) continue;
            if (mapa_vysok[ny][nx] > vyska) continue;

            int id2 = id_podla_pozicie(ny, nx);

            int velkost = spoj(id1, id2);
            if (velkost >= k){
                id_komponentu = sef(id1);
                return;
            }
        }
    }
}

vector< vector< char > > riesenie;

void BFS_presne_velkosti_k(int id_zaciatku){
    int zaciatok_x, zaciatok_y;
    pozicia_podla_id(id_zaciatku, zaciatok_y, zaciatok_x);
    queue<int> q;

    int pocet_vyfarbenych = 1;
    riesenie[zaciatok_y][zaciatok_x] = '*';
    q.push(zaciatok_x); q.push(zaciatok_y);

    while(!q.empty() && pocet_vyfarbenych < k){
        int x = q.front(); q.pop();
        int y = q.front(); q.pop();

        for(int j=0; j<4; j++){
            int nx = x + dx[j];
            int ny = y + dy[j];

            if (pocet_vyfarbenych == k) break;
            if (nx < 0 || nx >= c || ny < 0 || ny >= r) continue;
            if (sef(id_podla_pozicie(ny, nx)) != id_zaciatku) continue;
            if (riesenie[ny][nx] == '*') continue;

            pocet_vyfarbenych++;
            riesenie[ny][nx] = '*';
            q.push(nx); q.push(ny);
        }
    }
}

int main(){
    nacitaj_vstup();

    sort(policka_od_najnizsieho.begin(), policka_od_najnizsieho.end());

    int komponent, vyska;
    najnizsi_komponent_velkosti_k(komponent, vyska);

    riesenie.resize(r, vector<char>(c, '.'));
    BFS_presne_velkosti_k(komponent);

    printf("%d\n", vyska);
    for(int y=0; y<r; y++){
        for(int x=0; x<c; x++){
            printf("%c", riesenie[y][x]);
        }
        printf("\n");
    }
}
```

Buj

7. Osamelý a jednobunkový

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

Povedzme, že chceme zodpovedať otázku určenú vrcholom v a číslom k – teda chceme nájsť počet k-potomkov k-predka vrcholu v. To možno rozdeliť na dve podotázky:

- Zadaný je vrchol v_1 a číslo k_1. Existuje k_1-predok vrcholu v_1? Ak áno, ktorý vrchol to je?
- Zadaný je vrchol v_2 a číslo k_2. Koľko k_2-potomkov má vrchol v_2?

Pomalé riešenie

Na zodpovedanie prvej podotázky môžeme postupovať nasledovne:

- Ak $k = 0$, tak k-predok vrcholu v je samotný vrchol v.
- V opačnom prípade je to ten istý vrchol, ako $(k-1)$-predok otca v – a toho vieme nájsť rekurzívne.

Zistiť odpoveď pre druhú podotázku vieme takto:

- Ak $k = 0$, tak je odpoveď zrejme 1 – jediný 0-potomok vrcholu v je samotný vrchol v.
- V opačnom prípade nech $d_1, d_2, \ldots, d_i$ sú všetky deti vrcholu v. Označme $r_1, r_2, \ldots, r_i$ postupne počty $(k-1)$-potomkov týchto vrcholov. Ľahko si možno rozmyslieť, že v má práve $r_1 + r_2 + \ldots + r_i$ k-potomkov. No a hodnoty $r_1, r_2, \ldots, r_i$ vieme vypočítať rekurzívne.

Prvá fáza algoritmu môže mať až toľko krokov, koľko je hĺbka vrcholu v, keďže postupne zisťujeme jeho 1-predka, 2-predka, ... A to môže byť až lineárne od n.

Druhá fáza algoritmu môže mať až toľko krokov, koľko je počet vrcholov v podstrome vrcholu v – a tých môže byť až lineárne veľa od n.

Časová zložitosť na zodpovedanie jednej otázky je teda $O(n)$, celkovo máme preto časovú zložitosť $O(nq)$. Pamäťová zložitosť je $O(n)$ – pamätáme si iba reprezentáciu stromu.

Listing programu (C++)

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

struct Problem {

  int n, koren; // pocet vrcholov, cislo korenoveho vrchola
  vector<int> otec; // otec[i] je rodic vrchola i
  vector<vector<int> > synovia; // synovia[i] je zoznam potomkov vrchola i

  // nacitame pocet vrcholov, strom
  void nacitaj () {
    cin >> n;
    otec.resize(n);
    synovia.resize(n);
    for (int i = 0; i < n; i++) {
      cin >> otec[i];
      otec[i]--;
      if (otec[i] == -1) {
        koren = i;
      }
      else {
        synovia[otec[i]].push_back(i);
      }
    }
  }

  // vrati k-predka vrcholu v
  int predok (int v, int k) {
    if (k == 0) {
      return v;
    }
    if (v == koren) {
      return -1;
    }
    return predok(otec[v], k - 1);
  }

  // vrati pocet k-potomkov vrcholu v
  int prehladaj (int v, int k) {
    if (k == 0) {
      return 1;
    }
    int vysledok = 0;
    for (int i = 0; i < (int) synovia[v].size(); i++) {
      int syn = synovia[v][i];
      vysledok += prehladaj(syn, k - 1);
    }
    return vysledok;
  }

  // vrati odpoved na otazku "kolko k-potomkov ma k-predok vrcholu v?"
  int zodpovedaj (int v, int k) {
    int kPredok = predok(v, k);
    if (kPredok == -1) {
      return 0;
    }
    int vysledok = prehladaj(kPredok, k);
    return vysledok;
  }

  // odpovieme na vsetky otazky
  void zodpovedajVsetko () {
    int q; // pocet otazok
    cin >> q;
    for (int otazka = 0; otazka < q; otazka++) {
      int v, k;
      cin >> v >> k;
      v--;
      int vysledok = zodpovedaj(v, k);
      cout << vysledok << "\n";
    }
  }

  // spravi vsetko -- nacita strom a odpovie na otazky
```

```
  void spravVsetko () {
    nacitaj();
    zodpovedajVsetko();
  }

};

int main () {
  int t;
  cin >> t;
  for (int test = 0; test < t; test++) {
    Problem p;
    p.spravVsetko();
  }
  return 0;
}
```

Rýchlejšie zodpovedanie prvej podotázky

Predstavme si, že by sme si nepamätali pre každý vrchol iba jeho 1-predka, ale aj jeho 2-predka. Potom by sme k-predka vrcholu v vedeli nájsť zhruba dvakrát rýchlejšie:

- Ak $k \geq 2$, tak sa rekurzívne zavoláme – zrejme hľadáme $(k-2)$-predka vrcholu x, ktorý je 2-predkom vrcholu v.
- Ak $k = 1$, tak odpoveďou je 1-predok vrcholu v.
- V poslednom prípade $k = 0$, a odpoveďou je samotný vrchol v.

Ďalej si predstavme, že si nepamätáme iba 2-predkov, ale aj ďalších predkov s takými číslami, ktoré sú mocninami dvojky – $4, 8, 16, \ldots$. Takto si pre každý vrchol pamätáme rádovo $\log n$ čísel. Žiadny vrchol totiž nemá viac ako n predkov, a nemá preto zmysel počítať jeho (2^i)-predka pre $2^i > n$.

Zodpovedanie otázky má veľmi podobnú formu.

- Ak $k > 0$, nájdeme najväčšie číslo tvaru 2^i, ktoré je menšie ako k. Ak vrchol v nemá (2^i)-predka, tak nemôže mať ani k-predka. V opačnom prípade vieme v konštantnom čase pozrieť, ktorý vrchol je (2^i)-predok v. Rekurzívne nájdeme $(k-2^i)$-predka tohto vrcholu.
- Ak $k = 0$, tak odpoveď je zrejme v.

Navyše si všimnime, že ak v rekurzívnom volaní skočíme z v na jeho (2^i)-predka, potom v nasledujúcom rekurzívnom volaní určite môžeme skočiť najviac na (2^{i-1})-predka.

Ak by sme totiž znovu skočili hore aspoň o 2^i, skončili by sme vo vrchole, ktorý je aspoň $(2^i + 2^i)$-predkom v – potom určite $2^{i+1} \leq k$. Ale to by sme vedeli skočiť o 2^{i+1} už v prvom volaní.

Takže, ak označíme i najväčšie číslo také, že v má (2^i)-predka, stačí nám pri hľadaní k-predka skúšať $2^i, 2^{i-1}, 2^{i-2}, \ldots, 1$. Každé raz a v tomto poradí. Dostávame tak algoritmus, ktorý má časovú zložitosť $O(\log n)$.

Listing programu (C++)

```
int n; // pocet vrcholov stromu
vector<vector<int> > predok; // predok[i][j] je (2^j)-predok vrcholu i
vector<int> hlbka; // hlbka[i] je hlbka vrcholu i

// vrati k-predka vrcholu v alebo -1 ak neexistuje
int kPredok (int v, int k) {
  if (k > hlbka[v]) { // taky daleky predok neexistuje
    return -1;
  }
  int kolkyPredok = 1;
  for (int i = 1; i < (int) predok[v].size(); i++) {
    kolkyPredok *= 2;
  }
  for (int i = (int) predok[v].size() - 1; i >= 0; i--) {
    if (k >= kolkyPredok) {
      k -= kolkyPredok;
      v = predok[v][i];
    }
    kolkyPredok /= 2;
  }
  return v;
}
```

Ešte sme ale vôbec neriešili, ako rýchlo vypočítať týchto mocninových predkov. Predstavme si, že chceme nájsť (2^i)-predka vrcholu v. To ale nie je nič iné, ako jeho $(2^{i-1} + 2^{i-1})$-predok. Takže môžeme postupovať tak, že najprv nájdeme jeho (2^{i-1})-predka – nech to je x. Potom nájdeme (2^{i-1})-predka vrcholu x – výsledný vrchol je práve (2^i)-predok vrcholu v.

Na začiatku pre každý vrchol vypočítame jeho 1-predka (napríklad prehľadávaním do hĺbky). Potom vieme pre každý vrchol vypočítať jeho 2-predka, potom pre každý vrchol jeho 4-predka, potom pre každý vrchol jeho 8-predka, a tak ďalej.

Časová zložitosť tohto predpočítania je teda $O(n \log n)$, a pamäťová takisto, keďže si všetkých týchto predkov musíme pamätať.

Listing programu (C++)

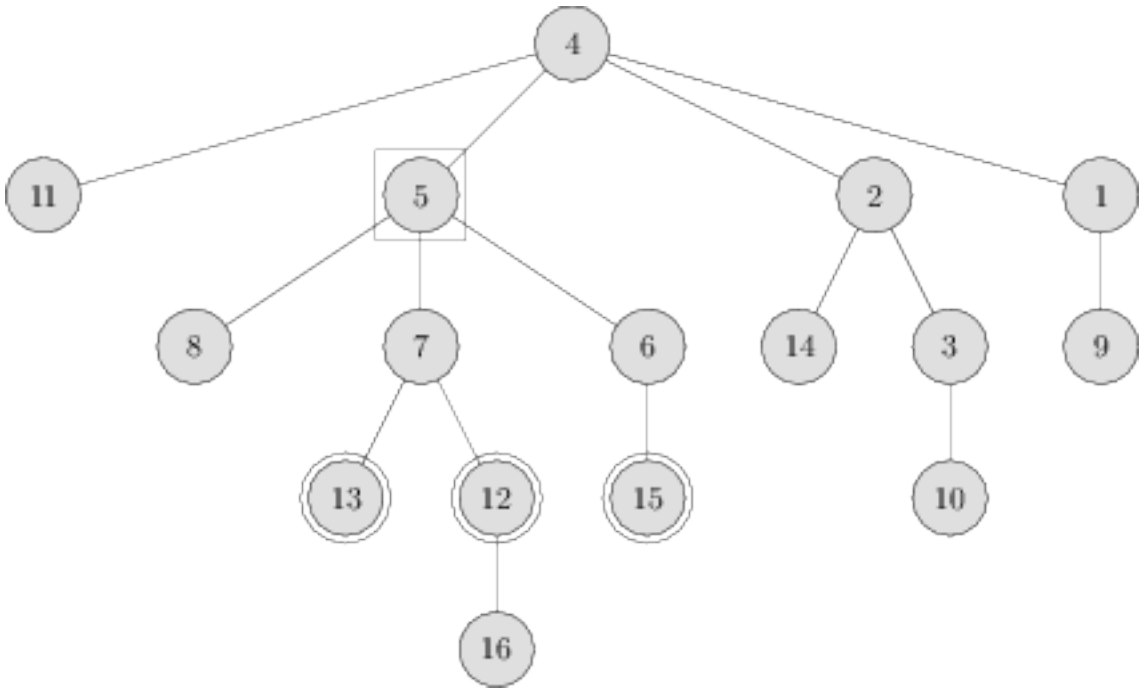
```
int n; // pocet vrcholov stromu
vector<vector<int> > predok; // predok[i][j] je (2^j)-predok vrcholu i
                             // na zaciatku predok[i] obsahuje jedine otca vrcholu i

// pre kazdy vrchol spocitame jeho 2^i predkov
void predratajPredkov () {
  bool este = true;
  int i = 0;
  while (este) {
    este = false;
    for (int v = 0; v < n; v++) {
      if ((int) predok[v].size() <= i) { // 2^i predok vrcholu v neexistuje
        continue;
      }
      int medziPredok = predok[v][i];
      if ((int) predok[medziPredok].size() <= i) { // 2^i predok medziPredka neexistuje
        continue;
      }
      int koncovyPredok = predok[medziPredok][i];
      predok[v].push_back(koncovyPredok);
      este = true; // ma zmysel pokracovat jedine vtedy, ked sme pre nejaky vrchol
                   // nasli jeho 2^i predka
    }
    i++;
  }
}
```

Rýchlejšie zodpovedanie druhej podotázky

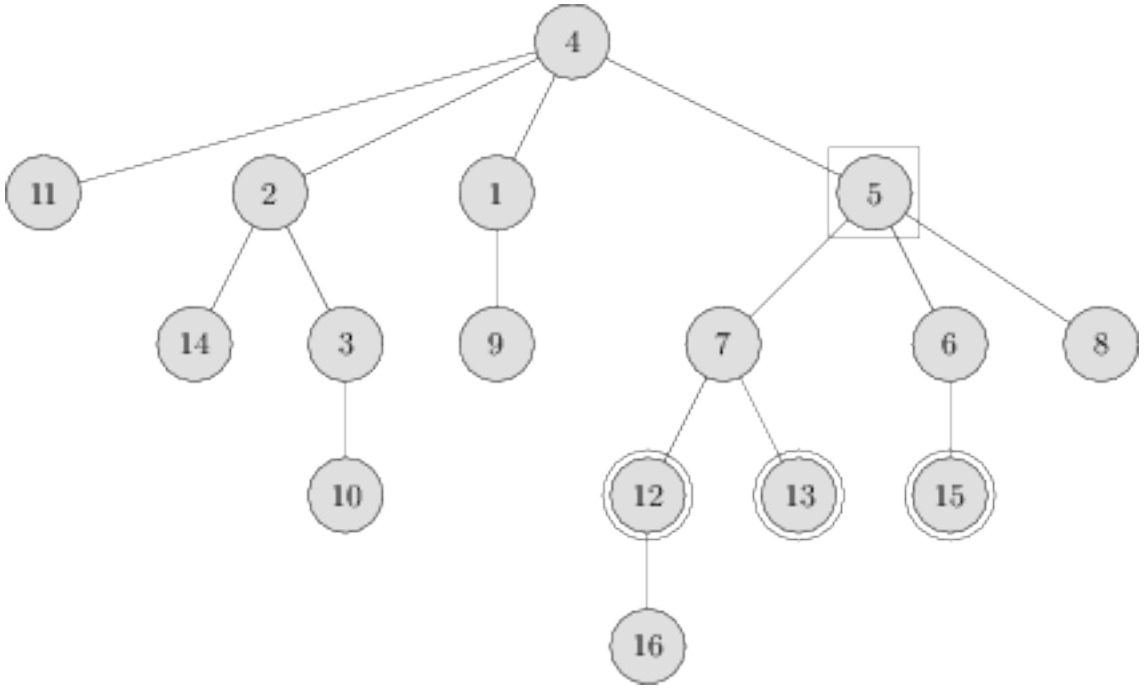
Rozdeľme si všetky vrcholy do vrstiev podľa toho, akú majú vzdialenosť od koreňa stromu. Na prvej vrstve sa nachádza iba koreň, na druhej sú jeho deti, na tretej sú zas ich deti, ... Pre každý vrchol v označme $h(v)$ číslo vrstvy, na ktorej sa nachádza.

Zrejme k-potomkov vrcholu v stačí hľadať medzi vrcholmi na vrstve $h(v)+k$. Tých ale stále môže byť rádovo n.

Po chvíli kreslenia stromov a pozorovania rôznych k-potomkov rôznych vrcholov v odpozorujeme, že tí potomkovia akosi vždy tvoria vo vrstve súvislý úsek. V nasledujúcom obrázku sú napríklad zobrazení 2-potomkovia vrcholu 5:



Ten istý strom môže ale vyzerať úplne inak:



Uvedomíme si, čo určuje poradie vrcholov vo vrstve – pre každý vrchol máme totiž zoznam jeho synov, ale týchto synov môžeme zľava doprava nakresliť v ľubovoľnom poradí. Zvoľme si teda pre každý vrchol v nejaké poradie jeho synov.

Nahliadneme, že ak a, b sú synovia vrcholu v a a je pred b, tak v rámci každej vrstvy je každý potomok a v tej vrstve pred každým potomkom b.

Na základe týchto pozorovaní vieme poradie vo vrstvách zostrojiť jednoduchým prehľadávaním do hĺbky. Keď **prvýkrát navštívime** (ďalej *objavíme*) vrchol v, tak si pre neho zapamätáme, koľký v poradí objavený vrchol to je. Potom postupne navštívime jeho synov v poradí, ktoré sme si zvolili. V rámci každej vrstvy sú potom vrcholy usporiadané podľa toho, kedy sme ich objavili.

A keď už vieme, že vo vrstve budú vrcholy usporiadané podľa času ich objavenia, tak ich do vrstvy vieme pridať hneď keď ich objavíme.

Listing programu (C++)

```
vector<int> zac, kon; // zac[i] (kon[i]) je zaciatocne (koncove) cislo vrchola i

vector<vector<int> > synovia; // synovia[i] je zoznam potomkov vrchola i
vector<vector<int> > vrstvy; // vrstvy[i] je zoznam vrcholov, ktore su vzdialene od korena presne i

// spracovanie podstromu vrcholu v, ktory sedi na vrstve s cislom h
void dfs (int v, int h) {
  zac[v] = cas;
  cas++;

  if ((int) vrstvy.size() <= h) {
    vrstvy.push_back(vector<int>());
  }
  vrstvy[h].push_back(v);

  for (int i = 0; i < (int) synovia[v].size(); i++) {
    int syn = synovia[v][i];
    dfs(syn, h + 1);
  }

  kon[v] = cas;
  cas++;
}
```

Teraz už máme vrcholy v každej vrstve usporiadané tak, že keď hľadáme k-potomkov niektorého vrcholu v, tvoria vo vrstve súvislý úsek. Ako ale zistíme, kde ten úsek začína a kde končí?

Zoberme si ľubovoľnú vrstvu, a nech x je niektorý vrchol na nej. Rozmyslite si, že ak x je potomkom v, tak sme ho určite objavili neskôr ako sme objavili v.

Pozrime sa na úsek potomkov v na tejto vrstve. Prvý vrchol x_1 v tomto úseku sme objavili neskôr, ako v – a spomedzi všetkých takých vrcholov sme ho objavili najskôr. Vieme ho teda vo vrstve binárne vyhľadať, nakoľko vrcholy na nej sú usporiadané podľa času ich objavenia.

Takto sme našli začiatok úseku, ako ale nájdeme jeho koniec? Podobne – pre každý vrchol si tiež zistíme, kedy sme ho **naposledy navštívili** (ďalej *opustili*).

Opäť si zoberme ľubovoľnú vrstvu, a nech x je niektorý vrchol na nej. Potom ak x je potomkom v, tak sme ho určite opustili skôr, ako sme opustili v.

Rozmyslite si, že v každej vrstve je usporiadanie vrcholov podľa času prvej návštevy rovnaké, ako usporiadanie podľa času poslednej návštevy.

Pozrime sa teda na úsek potomkov v na tejto vrstve. Posledný vrchol x_2 v tomto úseku bol opustený skôr, ako sme opustili v – a spomedzi všetkých takých vrcholov sme ho opustili najneskôr. Vieme ho teda vo vrstve binárne vyhľadať.

No a ak prvý vrchol v úseku je i-ty, a posledný vrchol v úseku je j-ty, počet vrcholov vo vnútri úseku je $j - i + 1$.

Rekapitulácia

Zrekapitulujme si naše riešenie. Najprv v čase $O(n \log n)$ predpočítame pre každý vrchol jeho $1, 2, 4, 8, \ldots$ predkov. V čase $O(n)$ prehľadáme graf do hĺbky, a pre každý vrchol zistíme čas jeho prvej návštevy a čas poslednej návštevy. Pritom zostrojujeme zoznamy vrcholov v každej vrstve. Celkovo máme čas $O(n \log n)$ na predpočítanie.

Keď odpovedáme na otázku určenú vrcholom v a číslom k, v čase $O(\log n)$ nájdeme k-predka vrcholu v. Potom vo vrstve $h(v)$ binárne vyhľadáme začiatok úseku potomkov k-predka v, a tiež koniec tohto úseku – to nám tiež zaberie čas $O(\log n)$. Na otázkach teda minieme $O(q \log n)$ času.

Celková časová zložitosť je potom $O((n + q) \log n)$. Pamäťová zložitosť je $O(n \log n)$.

Rýchlejšie a úspornejšie

Vráťme sa späť k prvej podotázke. V nej sme dostali zadaný vrchol v a číslo k, a chceli sme vedieť, či existuje k-predok vrcholu v a ak áno, ktorý vrchol to je. Označme ho x.

Teraz už ale pre každý vrchol vieme, kedy sme ho prvýkrát navštívili, a kedy sme ho naposledy navštívili. A tiež vieme povedať, v ktorej vrstve sa nachádza. To využijeme na vytvorenie lepšieho riešenia.

Vieme totiž, že x musí spĺňať nasledovné:

- Musí ležať na vrstve s číslom $h(v) - k$.
- Objavili sme ho skôr, ako vrchol v.
- Opustili sme ho neskôr, ako v.

Potom vieme x vo vrstve binárne vyhľadať.

Týmto sme si ušetrili $n \log n$ operácii potrebných na predpočítanie, a s tým spojených $n \log n$ pamäte. Dostávame tak celkovú časovú zložitosť $O(n + q \log n)$, a pamäťovú zložitosť $O(n)$.

Listing programu (C++)

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

struct Problem {

  int n, koren; // pocet vrcholov, cislo korenoveho vrchola
  vector<int> otec; // otec[i] je rodic vrchola i
  vector<vector<int> > synovia; // synovia[i] je zoznam potomkov vrchola i

  vector<int> zac, kon; // zac[i] (kon[i]) je zaciatocne (koncove) cislo vrchola i
  vector<vector<int> > vrstvy; // vrstvy[i] je zoznam vrcholov, ktore su vzdialene od korena presne i
  vector<int> hlbka; // hlbka[i] je hlbka vrcholu i

  // nacitame pocet vrcholov, strom a inicializujeme zac, kon
  void nacitaj () {
    cin >> n;
    otec.resize(n);
    synovia.resize(n);
    zac.resize(n);
    kon.resize(n);
    hlbka.resize(n);
    for (int i = 0; i < n; i++) {
      cin >> otec[i];
      otec[i]--;
      if (otec[i] == -1) {
        koren = i;
      }
      else {
        synovia[otec[i]].push_back(i);
      }
    }
  }
```

```
  // tymto prehladame graf do hlbky a spocitame zac, kon, vrstvy, hlbka
  void dfs (int v, int h, int& prveVolne) {
    zac[v] = prveVolne;
    prveVolne++;
    hlbka[v] = h;

    if ((int) vrstvy.size() <= h) {
      vrstvy.push_back(vector<int>());
    }
    vrstvy[h].push_back(v);

    for (int i = 0; i < (int) synovia[v].size(); i++) {
      int syn = synovia[v][i];
      dfs(syn, h + 1, prveVolne);
    }
    kon[v] = prveVolne;
    prveVolne++;
  }

  // tymto spocitame cele zac, kon, vrstvy
  void dfs () {
    int prveVolne = 0;
    dfs(koren, 0, prveVolne);
  }

  // vrati najvacsie i take, ze zac[vrstva[i]] < limit
  int binarneVyhladaj (int limit, vector<int>& vrstva) {
    int l = -1;
    int r = (int) vrstva.size();
    while (r - l > 1) {
      int s = (l + r) / 2;
      if (zac[vrstva[s]] < limit) {
        l = s;
      }
      else {
        r = s;
      }
    }
    return l;
  }

  // vrati k-predka vrchola v
  int predok (int v, int k) {
    int h = hlbka[v] - k;
    if (h < 0) {
      return -1;
    }
    int i = binarneVyhladaj(zac[v], vrstvy[h]);
    return vrstvy[h][i];
  }

  // vrati pocet k-potomkov vrchola v
  int kPotomkov (int v, int k) {
    int h = hlbka[v] + k;
    if (h >= (int) vrstvy.size()) {
      return 0;
    }
    int prvy = binarneVyhladaj(zac[v], vrstvy[h]) + 1;
    int posledny = binarneVyhladaj(kon[v], vrstvy[h]);
    return posledny - prvy + 1;
  }

  // vrati odpoved na otazku "kolko k-potomkov ma k-predok vrcholu v?"
  int zodpovedaj (int v, int k) {
    int kPredok = predok(v, k);
    if (kPredok == -1) {
      return 0;
    }
    int vysledok = kPotomkov(kPredok, k);
    return vysledok;
  }

  // odpovieme na vsetky otazky
  void zodpovedajVsetko () {
    int q; // pocet otazok
    cin >> q;
    for (int otazka = 0; otazka < q; otazka++) {
      int v, k;
      cin >> v >> k;
      v--;
      int vysledok = zodpovedaj(v, k);
      cout << vysledok << "\n";
    }
  }

  // spravi vsetko -- nacita strom a odpovie na otazky
  void spravVsetko () {
    nacitaj();
    dfs();
    zodpovedajVsetko();
  }

};

int main () {
  int t;
  cin >> t;
  for (int test = 0; test < t; test++) {
    Problem p;
```

```
    p.spravVsetko();
  }
  return 0;
}
```

Žaba

8. Oprášený startup

(max. 0 b za popis, 25 b za program)

Dúfam, že so mnou budete súhlasiť v tom, že táto optimalizačná úloha bola naozaj zaujímavá. Nestáva sa predsa každý deň, že píšete program, ktorého výstupom je iný program. Otázkou však je, ako napísať k takejto úlohe vzorák.

Nakoniec som sa rozhodol, že vám v tomto vzoráku ukážem hlavné myšlienky, ktorými sa dalo riadiť pri postupnom získavaní bodov. Najprv si ukážeme niekoľko jednoduchších prístupov, potom si predstavíme pomerne zaujímavý algoritmus slúžiaci na kódovanie znakov a nakoniec to všetko skombinujeme do výsledného riešenia. To samozrejme nebude najlepšie možné, ale kvôli prehľadnosti vzorového riešenia nebudeme uplatňovať príliš veľa optimalizácií.

S niečím začať treba

Prečo teda nespraviť prvú a najľahšiu vec, čo nám napadne! Nemáme čo stratiť a naprogramovanie ľahkého riešenia nám zaberie veľmi málo času. Otestujeme si ním, či správne rozumieme príkazom, ktoré Dávidkov procesor používa, a keď budeme neskôr robiť niečo zložitejšie, budeme si vedieť porovnať naše riešenia a zistiť, ako veľmi sme sa zlepšili.

Ak sa pozrieme na príkazy, ktoré používa Dávidkov procesor, vidíme, že vypisovať budeme príkazom `OUT`. Ten podľa zadania zoberie vrchné číslo na zásobníku a vypíše toľký znak z ASCII tabuľky. Ak teda chceme napríklad vypísať znak `„a"`, ktorý je v ASCII tabuľke 97-my, tak najskôr musíme dostať na zásobník číslo 97, čo spravíme príkazom `PUSH 97`. Takto vieme dostať jednoduchý program, ktorý vypíše znak `„a"` a nič iné.

Listing programu (Text)

```
PUSH 97
OUT
```

A toto predsa vieme spraviť s ľubovoľným znakom v texte. Pozrieme sa na to, koľký v poradí je tento znak v ASCII tabuľke, vložíme také číslo do zásobníka a potom ho z neho rovno vypíšeme. Môžeme preto napísať jednoduchý program v `C++`, ktorý načíta text a vytvorí z neho program pre Dávidkov procesor, ktorý daný text vypíše.

Všimnite si, ako v tomto programe šikovne využívame fakt, že `C++` vie s premennou typu `char` pracovať ako s číslom z ASCII tabuľky. Ak preto necháme vypísať `char` ako `int`, vypíše sa žiadaná hodnota.

Listing programu (C++)

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

int main() {
    char c;
    while(scanf("%c", &c) == 1) {
        printf("PUSH %d\n", c);
        printf("OUT\n");
    }
}
```

Programovanie v Dávidkovom procesore

Ukázali sme si, ako vieme spraviť prvé, veľmi jednoduché riešenie, ktoré pre každý znak na vstupe použije dva príkazy Dávidkovho procesora. Ďalej potrebujeme prísť s niečím, čo tento počet príkazov výrazne zmenší. To, že musíme pridať na zásobník konkrétnu hodnotu pre každý znak asi neovplyvníme (zatiaľ). Mohli by sme však radšej použiť ostatné príkazy Dávidkovho procesora na to, aby sme zmenšili počet príkazov potrebných na vypisovanie.

Čo spravíme v `C++`, ak chceme vypísať obsah nejakého poľa (zásobníka)? Použijeme `for`-cyklus. Hľadáme preto spôsob, akým naprogramovať `for`-cyklus v Dávidkovom procesore. A keďže `for`-cyklus nie je nič iné ako vracanie sa na tie isté riadky, budeme chcieť použiť príkaz `JGZ`.

Rozmyslime si najskôr slovne, čo chceme spraviť: "Vypíš znak zodpovedajúci vrchnému číslu na zásobníku a toto číslo odstráň. Ak zásobník nie je prázdny, vráť sa na príkaz výpisu (začiatok)."

Hneď si však všimneme, že nevieme zistiť, či je zásobník prázdny alebo nie. Vieme iba zisťovať, či je na vrchu zásobníka číslo väčšie ako 0. A keďže 0 sa medzi číslami na zásobníku nachádzať nebude (lebo to nie je znak, ktorý by sa nám objavil v texte), môže 0 slúžiť ako vhodná zarážka na koniec zásobníka. Potrebujeme preto overiť, či je na vrchu zásobníka 0 a ak nie je, tak pokračovať vo vypisovaní.

Posledný problém, ktorý musíme vyriešiť, je, že príkaz `JGZ` odstráni dve vrchné čísla zo zásobníka. Musíme si preto vytvoriť kópiu vrchného čísla, ktorú môžeme bez problémov odstrániť. Takúto kópiu vytvoríme pomocou príkazu `READ`.

Listing programu (Text)

```
OUT
PUSH -1
READ      // nakopíruj číslo na pozícii -1 (teda to najvrchnejšie) na vrch zásobníka
PUSH -5   // o koľko príkazov sa chcem vrátiť, ak číslo navrchu nie je 0
JGZ       // ak druhé číslo zvrchu zásobníka nie je 0, tak program
          // sa vráti o 5 inštrukcií späť, čo je práve príkaz OUT
```

To, čo sme spravili, je vlastne programovanie pomocou príkazov Dávidkovho procesora. Poriadne si pozrite vyššie uvedený program. Niečo podobné ešte párkrát uvidíte a v podstate nič zložitejšie potrebovať nebudeme.

Náš program teda do zásobníka postupne pridá jedno číslo za každý znak textu a potom vykoná vyššie uvedený kúsok programu, ktorý znaky vypíše. Netreba ešte zabudnúť na to, že čísla znakov musíme na zásobník pridávať v obrátenom poradí, teda prvý znak textu musí byť pridaný ako posledný.

Štyri v jednom

Koľko príkazov použil náš posledný program? Potreboval jeden príkaz za každé písmeno na vstupe a potom ešte päť príkazov na vypísanie. Asi je jasné, že ak chceme vymyslieť ešte lepšie riešenie, musíme zmenšiť počet čísel, ktoré na začiatku vložíme do zásobníka.

To znamená, že jedno číslo musí zastupovať (kódovať) viacero znakov. Uvedomme si, že čísla, ktoré môžeme vkladať do zásobníka môžu mať veľkosť až $2^{31} - 1$. Naše riešenia však zatiaľ používali iba čísla, ktoré nemohli byť väčšie ako 2^7 (počet znakov v ASCII tabuľke). Pokúsime sa preto nájsť spôsob, akým zakódovať viacero znakov do jedného čísla a samozrejme aj spôsob, ako tento postup obrátiť (ako z čísla zistiť znaky, ktoré kóduje).

Predstavme si, že chceme zakódovať dva znaky – znak `„a"` s ASCII číslom 97 a znak `„b"` s ASCII číslom 98, pričom `„a"` má byť vypísané skôr ako `„b"`. Použime na to číslo $12\,641 = 128 \cdot 98 + 97$. Dôvod, prečo je takéto kódovanie dobré je ten, že z čísla 12 641 vieme ľahko zistiť, ktorý znak má byť vypísaný prvý. Bude to predsa zvyšok čísla 12 641 po delení 128. Druhé číslo na vypísanie bude celočíselný podiel týchto dvoch čísel a na počítanie podielu a zvyšku po delení máme pre Dávidkov procesor príkaz `DIV`.

Nemusíme sa však obmedziť tým, že do jedného čísla zakódujeme len dva znaky. Takéto čísla budú totiž nanajvýš veľkosti 2^{14}. Pridajme preto rovnakým spôsobom ďalšie dve čísla. Zo štyroch znakov a, b, c a d (toto sú premenné a nie konkrétne znaky) potom vieme spraviť jedno číslo menšie ako 2^{28} nasledovne:

$$a + 128 \cdot b + 128^2 \cdot c + 128^3 \cdot d$$

Sami si rozmyslite, že postupným delením tohto čísla číslom 128 budeme ako zvyšky dostávať čísla zodpovedajúce znakom a, b, c a d.

Program, ktorý kóduje náš text takýmto spôsobom je o niečo komplikovanejší. Musíme si dať pozor, aby sme zakódovali čísla v správnom poradí a musíme vymyslieť program používajúci príkazy Dávidkovho procesora, ktorý bude zadané čísla deliť 128 a vypisovať ich na výstup. Nie je to však až také ťažké a nižšie si môžete pozrieť jedno možné riešenie.

Listing programu (C++)

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

int main() {
    vector<char> Text;
    char c;
    while (scanf("%c", &c) == 1) Text.push_back(c);
    while (Text.size() % 4 != 0) Text.push_back(0); // doplním pole Text o nejaké 0, aby mal správnu dĺžku
    reverse(Text.begin(), Text.end());

    printf("PUSH␣0\n"); // zarážka na spodok zásobníka

    for(int i = 0; i < Text.size()/4; i++)
        printf("PUSH␣%d\n",128*128*128*Text[4*i] + 128*128*Text[4*i+1] + 128*Text[4*i+2] + Text[4*i+3]);

    printf("PUSH␣128\n");
```

```
    printf("DIV\n");
    printf("OUT\n");
    printf("PUSH -1\n");
    printf("READ\n");
    printf("PUSH -7\n");
    printf("JGZ\n");
    // ak sa program dostane až sem, znamená to, že číslo
    // na vrchu sme vydelili už 4 krát a musíme ho zásobníka vyhodiť
    printf("ADD\n");
    printf("PUSH -1\n");
    printf("READ\n");
    printf("PUSH -12\n");
    printf("JGZ\n");
}
```

Tok bitov

Pokúsime sa ešte viac vylepšiť naše riešenie. Dobrý nápad je pozrieť sa na jednotlivé texty, ktoré máme zadané. Možno nás napadne nejaké lepšie riešenie, našité priamo na ne.

A naozaj, ak sa pozrieme na text `1.in`, zistíme, že obsahuje iba písmená „`c`" (dokonca neobsahuje ani koniec riadku). Je zjavne zbytočné do zásobníka vkladať veľa hodnôt, ktoré nám aj tak povedia to, čo už dopredu vieme – vypíš písmeno „`c`". Namiesto toho by nám v zásobníku mohla stačiť iba jedna hodnota, ktorá hovorí, koľko písmen „`c`" ešte máme vypísať. Túto hodnotu budeme postupne znižovať až kým neklesne na 0. Vtedy program ukončíme pomocou príkazu `JGZ`.

Dostaneme jednoduchý 9-príkazový program:

Listing programu (Text)

```
PUSH 9379
PUSH 99
OUT
PUSH -1
ADD
PUSH -1
READ
PUSH -8
JGZ
```

Pozrime sa teraz na text `2.in`. V ňom sa nachádzajú iba dva druhy písmen – „`f`" a „`y`". Naším cieľom bude teda zakódovať túto postupnosť dvoch druhov písmen čo najúspornejšie. Predchádzajúce všeobecné riešenie používalo na kódovanie čísla 102 a 121. To je však strašne zbytočné. Na zakódovanie písmena „`f`" môžeme predsa použiť hodnotu 0 a ako kód „`y`" použijeme 1. Záleží len na nás, či bude náš program rozoznávať „`f`" ako 102 alebo 0.

Náš text sme si teda zmenili na postupnosť bitov (základná jednotka informácie, ktorá nadobúda iba hodnotu 0 alebo 1) – a v počítači je aj každé číslo reprezentované ako postupnosť bitov (v našom prípade 32 bitov). Namiesto 4 písmen vieme teraz do jedného čísla preto zakódovať až 32 písmen, čo je 8-násobné zlepšenie.

Napriek tomu, to nebude také jednoduché. Po prvé, potrebujeme do čísla zakódovať nejakým spôsobom túto postupnosť bitov. To síce vyzerá priamočiaro – nasekám si bity na 32-tice a jednoducho ich zmením na číslo – avšak objaví sa niekoľko problémov, ktoré budeme musieť vyriešiť. Po prvé, je nebezpečné používať všetkých 32 bitov, lebo hoci Dávidkov procesor dokáže pracovať s 2^{32} číslami, polovica z nich je záporná (rozsah čísel v zásobníku je -2^{31} až $2^{31}-1$). Tu by už naša matematika – delíme 2 a pozeráme sa na zvyšok – nefungovala tak akoby sme chceli, a preto sa obmedzíme len na prvých 31 bitov, ktoré nám zaručia, že čísla budú kladné.

Navyše však dostaneme nasledovný problém. Náš program bude postupne deliť naše číslo hodnotou 2. Kedy má však zastať? Ak sa číslo zmenší na 0, znamená to, že už som spracoval všetky bity, alebo ešte nasleduje niekoľko 0-vých bitov? Riešení tohto problému môže byť viac. Pri jednom možnom spôsobe zopakujeme Dávidkovým programom 31 delení. Mohli by sme 31-krát nakopírovať kus programu ktorý vykoná delenie alebo na to použiť ďalší (vnorený) cyklus. Buď sa nám teda zväčší počet príkazov, alebo to bude neprehľadnejšie a takisto budeme musieť ešte vyriešiť prípad, kedy niektoré číslo nekóduje 31, ale menej bitov.

Riešenie, ktoré si ale ukážeme my, bude založené na tom, že si do každého čísla spravíme značku. V podstate si vyhradíme najvyšší používaný bit (väčšinou 31-vý) ako zarážku. Zakaždým bude mať hodnotu 1. Tým pádom, pri delení dvojkou sa nám raz stane, že nám v celom čísle ostane už len tento jednotkový bit, a číslo, ktoré budeme spracovávať bude teda rovné 1. Ak nastane takýto prípad, budeme vedieť, že už sme spracovali všetky bity, ktoré kódovalo toto číslo a môžeme sa presunúť na to ďalšie.

Pozrite si nižšie uvedenú implementáciu takéhoto riešenia. Všimnite si hlavne časť, v ktorej je zadrôtovaný Dávidkov program, ktorý vydelí číslo hodnotou 2 a podľa zvyšku skočí buď do časti, ktorá vypisuje písmeno „`f`" alebo do časti pre písmeno „`y`".

Listing programu (C++)

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

using namespace std;

vector<char> Text;

int main() {
    // načítam text
    char c;
    while(scanf("%c", &c) == 1) Text.push_back(c);

    //zmením text na postupnosť 0 a 1
    string tok_bitov = "";
    for(int i = 0; i < Text.size(); i++)
        if(Text[i] == 'f') tok_bitov += "0";
        else tok_bitov += "1";
    reverse(tok_bitov.begin(), tok_bitov.end());

    printf("PUSH 0\n"); // zarážka na spodok zásobníka

    // zakódujem 30 bitov do jedného čísla, ktoré vložím na zásobník
    int kolko = 0, cislo = 1;
    for(int i = 0; i < tok_bitov.size(); i++) {
        cislo *= 2;
        cislo += tok_bitov[i] - '0';
        kolko++;
        if(kolko == 30 || i == tok_bitov.size()-1) {
            printf("PUSH %d\n",cislo);
            cislo = 1; kolko = 0;
        }
    }

    // skontrolujem, či v čísle už nezostala len 1 a nepotrebujem ďalšie číslo
    // potom skontrolujem, či na vrchu nie je 0 a nemám už skončiť
    printf("PUSH -1\n");
    printf("READ\n");
    printf("PUSH -1\n");
    printf("ADD\n");
    printf("PUSH 8\n");
    printf("JGZ\n");
    printf("MUL\n");
    printf("PUSH -1\n");
    printf("READ\n");
    printf("PUSH 3\n");
    printf("JGZ\n");
    printf("PUSH 1\n");
    printf("PUSH 1000000\n");
    printf("JGZ\n");

    // pozriem sa na posledný bit aktuálneho čísla
    printf("PUSH 2\n");
    printf("DIV\n");
    printf("PUSH 5\n"); // ak je to 1, chcem preskočiť 5 príkazov
    printf("JGZ\n");
    printf("PUSH 102\n"); // ak som neskončil, tak tam bola 0
    printf("OUT\n");
    printf("PUSH 1\n");
    printf("PUSH -23\n"); // skoč na začiatok
    printf("JGZ\n");
    printf("PUSH 121\n");
    printf("OUT\n");
    printf("PUSH 1\n");
    printf("PUSH -28\n"); // skoč na začiatok
    printf("JGZ\n");
}
```

Uvedomme si ešte, že takéto riešenie sa dá istým spôsobom zovšeobecniť. Na vstupe máme postupnosť (tok) bitov. Našou úlohou je spracovávať ich jeden po druhom a robiť rozhodnutia podľa toho, či je aktuálny bit 0 alebo 1. To, čo sme si ukázali v tejto časti, boli často len technické pomôcky, ako niečo takéto robiť pomocou príkazov Dávidkovho procesora.

Nerovnomerný výskyt a prefixový kód

Pozrime sa teraz na text `4.in`[4]. V ňom sa objavujú len tri znaky – „`p`", „`q`" a „`A`".

Prirodzene, prvá vec, ktorá nám napadne, je použiť na kódovanie znakov čísla 0, 1 a 2 (v jednom čísle by sme si pamätali znaky nie ako zvyšky po opakovanom delením dvomi, ale po delení tromi). Na tom by nebolo nič zlé, ale predsa len, zaberá to trochu veľa miesta – na každé číslo musíme využiť takmer 2 bity.

[4] Text `3.in` preskakujem úmyselne. Vieme naň síce vymyslieť vcelku jednoduché a pomerne dobré riešenie, nijak však nesúvisí so vzorovým riešením, a preto ho tu nespomeniem.

Zostaňme ale pri našej predstave toku bitov s hodnotami 0 alebo 1. Mohli by sme napríklad každému znaku priradiť nejaké dva bity (napr.: „p" by sme zakódovali ako „00", „q" ako „01" a „A" ako „10"). Potom by sme opäť dostali tok bitov, akurát by sa trochu nepríjemnejšie spracovával. Ale to nám tiež neušetrí počet bitov, ktoré použijeme. Skúsme to ešte inak. Použime na kódovanie tri, nerovnako dlhé, reťazce „1", „00" a „01".

Keď každému písmenu priradíme jeden takýto reťazec, opäť vieme celý text zmeniť na postupnosť bitov. Naviac, táto postupnosť bitov sa dá jednoducho dekódovať. Postupne prechádzame bity zľava – ak je prvý bit 1, vieme, že sme narazili na prvé písmeno a môžeme ho vypísať. Ak je prvý bit 0, tak zase vieme, že sa musíme pozrieť ešte na druhý bit a až ten nám rozhodne, čo máme vypísať. V každom momente sa preto vieme jednoznačne rozhodnúť, čo treba spraviť. Naviac, na zakódovanie jedného zo znakov použijeme iba jeden bit.

To, čo môžeme v takomto riešení ovplyvniť je, ktorým písmenám priradíme ktorý reťazec. Už pri letmom pohľade do textu `4.in` by sme si mali všimnúť, že znak „A" sa tam nachádza častejšie. Je preto logické, že práve jemu chceme priradiť reťazec „1", lebo ušetrenie jedného bitu (oproti reťazcom „01" a „00") bude častejšie. Na priradení zvyšných dvoch reťazcov nezáleží, lebo majú rovnakú dĺžku.

Následne ešte musíme trochu upraviť Dávidkov program, ktorý bude tento tok bitov spracovávať. Aj on sa totiž musí vedieť rozhodovať podľa toho, či vidí 0 alebo 1 a ak vidí 0 tak sa presunúť na ďalší bit, aby vedel, čo má vypísať. Riešenie je však veľmi podobné tomu, čo sme už videli predtým. Všimnite si, že Dávidkov program aj pre prehľadnosť rozdeľujem do častí, medzi ktorými náš program skáče a každá časť má presne určenú úlohu.

Listing programu (C++)

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

vector<char> Text;

int main() {
    // načítam text
    char c;
    while(scanf("%c", &c) == 1) Text.push_back(c);

    //zmením text na postupnosť 0 a 1
    string tok_bitov = "";
    for(int i = 0; i < Text.size(); i++)
        if(Text[i] == 'A') tok_bitov += "1";
        else if(Text[i] == 'p') tok_bitov += "00";
        else tok_bitov += "01";
    reverse(tok_bitov.begin(), tok_bitov.end());

    // zarážka na spodok zásobníka
    printf("PUSH 0\n");

    // zakódujem 30 bitov do jedného čísla, ktoré vložím na zásobník
    int kolko = 0, cislo = 1;
    for(int i=0; i < tok_bitov.size(); i++) {
        cislo *= 2;
        cislo += tok_bitov[i]-'0';
        kolko++;
        if(kolko == 30 || i == tok_bitov.size()-1) {
            printf("PUSH %d\n", cislo);
            cislo = 1; kolko = 0;
        }
    }

    // skontrolujem, či v čísle už nezostala len 1 a nepotrebujem ďalšie číslo
    // potom skontrolujem, či na vrchu nie je 0 a nemám už skončiť
    printf("PUSH -1\n");
    printf("READ\n");
    printf("PUSH -1\n");
    printf("ADD\n");
    printf("PUSH 8\n");
    printf("JGZ\n");
    printf("MUL\n");
    printf("PUSH -1\n");
    printf("READ\n");
    printf("PUSH 3\n");
    printf("JGZ\n");
    printf("PUSH 1\n");
    printf("PUSH 1000000\n");
    printf("JGZ\n");

    // pozriem sa na posledný bit aktuálneho čísla
    printf("PUSH 2\n");
    printf("DIV\n");
    printf("PUSH 21\n");
    printf("JGZ\n");
    // posledný bit bola 0, takže sa musím pozrieť na ďalší
    // najskôr sa pozriem, či som ešte neminul toto číslo
    printf("PUSH -1\n");
```

```
    printf("READ\n");
    printf("PUSH -1\n");
    printf("ADD\n");
    printf("PUSH 1\n");
    printf("JGZ\n");
    printf("MUL\n");
    // pozriem sa na ďalší bit
    printf("PUSH 2\n");
    printf("DIV\n");
    printf("PUSH 5\n");
    printf("JGZ\n");
    // vypíšem znak p za 00
    printf("PUSH 112\n");
    printf("OUT\n");
    printf("PUSH 1\n");
    printf("PUSH -34\n");
    printf("JGZ\n");
    // vypíšem znak q za 01
    printf("PUSH 113\n");
    printf("OUT\n");
    printf("PUSH 1\n");
    printf("PUSH -39\n");
    printf("JGZ\n");
    // vypíšem znak A za 1
    printf("PUSH 65\n");
    printf("OUT\n");
    printf("PUSH 1\n");
    printf("PUSH -44\n");
    printf("JGZ\n");
}
```

Na tomto riešení si môžeme všimnúť dve veľmi dôležité veci. Prvou je, že reťazce, ktoré priraďujeme ako kódy znakov nemusia byť rovnako dlhé. Prirodzene sa nám potom oplatí použiť kratšie reťazce pre častejšie sa vyskytujúce znaky, keďže vďaka tomu viac ušetríme. Takýto postup využíva napríklad aj Morseovka. Ak si pozriete tabuľku jej kódovania, zistíte, že dva najkratšie reťazce, bodka a čiarka, sú priradené písmenám `„E"` a `„T"`, ktoré sú (nie zhodou náhod) dve najčastejšie sa vyskytujúce písmená anglickej abecedy. Najdlhšie reťazce sú naopak priradené znakom, ktoré sa vyskytujú veľmi zriedkavo.

To však nebola jediná vec, na ktorú sme si museli dať pozor. Je potrebné tiež zaručiť, že dekódovanie je jednoznačné – teda vždy máme na základe aktuálneho bitu iba jedinú možnosť na to, čo spraviť. V Morseovke takéto niečo neplatí, ak sa pri dekódovaní objaví ako prvá bodka, nemôžeme si byť istí, či máme vypísať `„E"` (bodka), alebo prečítať ďalší znak, ktorý môže byť čiarka, čo by zodpovedalo písmenu `„A"` (bodka, čiarka). V Morseovke sa preto používa aj tretí symbol – oddeľovač.

My však oddeľovač k dispozícii nemáme (máme iba znaky 0 a 1), preto budeme používať **prefixové** kódy. To znamená, že žiaden kód nie je prefixom (začiatkom) iného kódu. Keď je splnená táto podmienka, kód sa bude dať vždy jednoznačne dekódovať.

Huffmanov kód

Zoberme si teraz ľubovoľný text, napríklad jeden z posledných štyroch – anglické a slovenské texty. Naším cieľom je vytvoriť prefixový kód, ktorý naviac zakóduje náš text na čo najmenej bitov. Uvedomte si, že takýto kód chceme našiť priamo na daný text, lebo výskyty písmen ovplyvňujú dĺžku kódových slov. V slovenčine totiž asi ťažko budeme hľadať písmená `„q"` alebo `„w"` a ak sa aj vyskytnú, môžeme im priradiť naozaj dlhý kód. V angličtine sú však tieto dve písmená oveľa častejšie a naše riešenie by to malo zobrať do úvahy. Hľadáme preto všeobecný algoritmus, ktorý dokáže pre daný text vytvoriť prefixový kód s optimálnou dĺžkou.(**Kód** je jedno konkrétne priradenie **kódových slov** (v našom prípade binárnych reťazcov) znakom textu.)

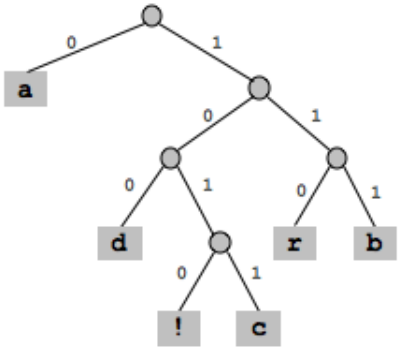
Zoberme si konkrétny text a spočítajme si písmená, ktoré obsahuje. Nech obsahuje m písmen, pričom i-te písmeno sa v tomto texte vyskytuje p_i-krát. Naviac, nech platí $p_0 \geq p_1 \geq \cdots \geq p_m$. Predstavme si tiež, že už máme nájdený optimálny prefixový kód, ktorý týmto znakom priradí binárne kódy $w_1, w_2 \ldots w_m$. Naviac, dĺžky týchto kódových slov si označme $l_1, l_2 \ldots l_m$.

Z tohto vyplýva, že na zakódovanie tohto textu potrebujeme presne $\sum_{i=1}^{m} p_i \cdot l_i$ bitov. Poďme si povedať, čo musí platiť o týchto slovách, ak majú tvoriť optimálny kód. Po prvé, musí platiť, že $l_1 \leq l_2 \leq \cdots \leq l_m$. Toto zodpovedá nášmu pozorovaniu, že dlhšie kódové slová chceme priraďovať menej častým znakom. Teraz si to však vieme aj dokázať. Ak by to totiž pre nejaké i a j neplatilo, vieme vymeniť im zodpovedajúce kódové slová, čím zmenšíme hodnotu sumy $\sum_{i=1}^{m} p_i \cdot l_i$. To je však v spore s tým, že priradenie kódovaní bolo optimálne, teda malo túto sumu minimálnu.

Veľmi užitočné bude tiež vedieť, ako si tieto kódové slová reprezentovať. Vhodnou štruktúrou bude písmenkový strom. Uvedomme si, že z každého vrcholu vychádzajú dve hrany – jedna reprezentujúca 0 a jedna reprezentujúca 1. V listoch tohto stromu sú potom jednotlivé znaky, ktoré kódujeme. Takýto strom je užitočný

na dve veci. Po prvé nám hovorí, aký kód má príslušný znak – stačí prejsť od koreňa k danému listu a zapisovať si, po akých hranách prechádzam.

Naviac nám tento strom dáva postup, ktorým rozkódovať postupnosť zadaných bitov. Začneme v koreni a vždy sa pohneme po tej hrane, ktorú nám určuje aktuálny bit postupnosti. Ak narazíme na list, tak vypíšeme príslušný znak, ktorý sme práve rozkódovali, a pokračujeme opäť od koreňa.



char	encoding
a	0
b	111
c	1011
d	100
r	110
!	1010

Na záver si ešte uvedomme nasledovné. Každý vnútorný vrchol (teda nelist) tohto stromu má práve dve hrany, ktoré z neho vychádzajú – 0 a 1. Ak by totiž mal iba jedného syna, môžeme tento vrchol odstrániť a nahradiť ho tým jediným vrcholom pod ním. Tým však skrátime kódové slová, čo je v spore s optimálnosťou.

Z toho vyplýva, že slovo w_m (s dĺžkou l_m), ktoré je najdlhšie, má suseda, ktorého kód má rovnakú dĺžku a líši sa iba v poslednom bite. Bez ujmy na všeobecnosti, nech je tento sused slovo w_{m-1} (ak by nebolo, tak ho vymeníme na túto pozíciu a nič tým nepokazíme).

Teraz si už vieme popísať algoritmus, ktorý vytvorí písmenkový strom kódujúci všetky naše znaky a toto kódovanie bude navyše optimálne. Myšlienka je naozaj jednoduchá – zakaždým si zoberieme dva najmenej často sa vyskytujúce znaky. Tie majú mať, podľa našich úvah, najdlhšie kódové slová a navyše majú byť v našom strome susedné. Vytvoríme preto vrchol, z ktorého vychádzajú hrany do dvoch listov, jeden patriaci znaku $m-1$ a druhý patriaci znaku m. Nemôžeme však na tieto znaky len tak zabudnúť. To čo sa však stalo je, že sme ich zlúčili do jedného spoločného znaku, ktorý je teraz reprezentovaný novovytvoreným vrcholom. A počet výskytov tohto spoločného znaku je predsa $p_{m-1} + p_m$. Úspešne sme preto zredukovali počet znakov, ktoré potrebujeme ešte spracovať, o jedna. A na to čo nám zostalo môžeme použiť úplne rovnakú myšlienku.

Predstaviť si to môžeme tak, že náš algoritmus má v každom kroku množinu aktívnych vrcholov, ktoré ešte nemajú priradeného žiadneho otca. Každý vrchol má naviac priradenú váhu zodpovedajúcu tomu, ako dôležitý je tento vrchol (ako často sa znaky v listoch pod týmto vrcholom vyskytujú v texte). Na začiatku náš algoritmus začína s m vrcholmi (listami), každý reprezentuje jeden znak a váha príslušného vrchola je určená podľa počtu výskytov daného znaku v texte.

Následne v každom kroku zoberie dva vrcholy s najmenšou váhou a vytvorí nový vrchol, ktorý bude slúžiť ako otec týchto dvoch vrcholov. Jeho váha bude daná súčtom váh jeho synov. Následne pôvodné dva vrcholy prestane náš algoritmus uvažovať, lebo im už priradil otca. Musí však začať uvažovať novovytvorený vrchol, ktorý otca ešte nemá. V každom kroku sa však zmenší počet vrcholov, ktoré ešte potrebuje spracovať o 1 a preto po $m-1$ krokoch mu ostane iba jediný vrchol – koreň celého stromu.

Takéto kódovanie sa nazýva Huffmanovo a skutočne je najlepšie možné, ktoré vieme dostať pri daných predpokladoch. Dôkaz optimálnosti som sa rozhodol sem neuviesť, keďže je pomerne formálny (aj keď vôbec nie ťažký). Ak by vás zaujímal, tak si ho môžete pozrieť tu: Úvod do teórie kódovania (3.3.3).

Jeho implementácia je takisto pomerne jednoduchá. Uvedomme si, že nám stačí použiť jednu haldu, v ktorej si udržiavame váhu jednotlivých vrcholov, ktoré ešte potrebujeme spracovať. Z haldy vieme jednoducho vybrať dva vrcholy s najmenšou váhou a spojiť ich. Časová zložitosť takéhoto riešenia bude preto $O(n \log n)$.

Výsledné riešenie

Vo vzoráku sme si postupne ukázali, ako sa dá programovať v Dávidkovom procesore, ukázali sme si, že pomerne dobré riešenie je zakódovať všetky znaky ako postupnosť bitov, a tú potom postupne dekódovať. Na to sme potrebovali priradiť každému znaku určité kódové slovo a chceli sme aby takéto priradenie vytvorilo čo najkratší kód.

Ukázali sme si, že hľadáme takzvaný prefixový kód (kód ktorý vieme jednoznačne dekódovať) taký, že uprednostňuje častejšie sa vyskytujúce znaky tým, že im priradzuje kratšie kódové slová. Nakoniec sme si ukázali aj konkrétny takýto algoritmus priraďovania – Huffmanov kód.

Ostáva nám už len spojiť všetky tieto veci dokopy a napísať program, ktorý pre daný text vytvorí Huffmanov kód, a následne vypíše program v jazyku Dávidkovho procesora, ktorý bude simulovať rozkódovanie Huffmanovho kódu.

Prekvapivo to však nie je až taký problém. Je nutné si dať pozor na veľa technickejších detailov, mnohé z nich sme však vyriešili v priebehu tohto textu. Najdôležitejšie je dať si pozor, aby ste postupnosť bitov zakódovali do čísel v správnom poradí, a aby ste mali vhodné zarážky – na spodku zásobníka alebo na konci čísla. Navyše, v tomto všeobecnom riešení musí náš program vedieť aj vypočítať, o koľko má skákať dopredu alebo dozadu na príkaze `JGZ`. Keď si to však trochu premyslíte, mali by ste vedieť tento problém vyriešiť.

A keďže raz vidieť je lepšie ako stokrát počuť, tak vám rovno ukážeme už hotový algoritmus, ktorý úlohu rieši. Odporúčame sa aspoň zbežne pozrieť na časť s Huffmanovým kódovaním.

Listing programu (C++)

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

#define For(i,n) for(int i=0; i<(n); i++)
#define mp(a,b) make_pair((a),(b))
typedef pair<int,int> pii;

struct node {
    int znak;
    int left, right;
    node() {
        znak = -1;
        left = -1; right = -1;
    }
};

vector<char> Text;
vector<node> Huff_tree;
vector<string> Kody;

// funkcia prechádza stromom a pamätá si kódy znakov
void prelez(int kde, string kod) {
    if(Huff_tree[kde].znak != -1) {
        Kody[Huff_tree[kde].znak] = kod;
        return;
    }
    prelez(Huff_tree[kde].left, kod+"0");
    prelez(Huff_tree[kde].right, kod+"1");
}

int main() {
    // načítam text
    char c;
    while(scanf("%c",&c) == 1) Text.push_back(c);

    // spočítam výskyty jednotlivých znakov
    vector<int> Vyskyt; Vyskyt.resize(128,0);
    For(i,Text.size()) Vyskyt[Text[i]]++;

    // vytvor strom Huffmanovho kódovania
    priority_queue<pii> Q;
    For(i,128) {
        if(Vyskyt[i] == 0) continue;
        // vložím do haldy výskyty aktuálneho znaku a spravím preň vrchol v strome
        // vkladám záporné hodnoty, aby som mal najmenšie čísla na vrchu haldy
        Q.push(mp(-Vyskyt[i], Huff_tree.size()));
        Huff_tree.push_back(node());
        Huff_tree[Huff_tree.size()-1].znak = i;
    }
    // spájam dva vrcholy s najmenšou váhou
    while(Q.size() != 1) {
        pii a=Q.top(); Q.pop();
        pii b=Q.top(); Q.pop();
        Q.push(mp(a.first+b.first, Huff_tree.size()));
        Huff_tree.push_back(node());
        Huff_tree[Huff_tree.size()-1].left = a.second;
        Huff_tree[Huff_tree.size()-1].right = b.second;
    }

    // rekurzívnou funkciou prelez() zistím aké kódy som priradil jednotlivým znakom
    Kody.resize(128,"");
    prelez(Huff_tree.size()-1,"");

    // vytvorím si postupnosť bitov, ktoré chcem zakódovať
    string tok_bitov = "";
    For(i,Text.size()) tok_bitov += Kody[Text[i]];
    reverse(tok_bitov.begin(), tok_bitov.end());

    // zarážka na spodok zásobníka
    printf("PUSH 0\n");

    // zakódujem 30 bitov do jedného čísla, ktoré vložím na zásobník
    int kolko=0, cislo=1;
```

```
    For(i, tok_bitov.size()) {
        cislo *= 2; cislo += tok_bitov[i]-'0';
        kolko++;
        if(kolko == 30 || i == tok_bitov.size()-1) {
            printf("PUSH %d\n",cislo);
            cislo=1; kolko=0;
        }
    }

    // spočítam si, na ktorých riadkoch budú začínať jednotlivé kusy kódu
    // pre jednotlivé vrcholy Huffmanovho kódu. Vďaka tomu budem počítať,
    // kam sa mám presunúť pri príkaze JGZ
    vector<int> Riadky;
    Riadky.resize(Huff_tree.size()+1,14);
    For(i,Huff_tree.size()) {
        // 14 a 5 sú počty príkazov v jednotlivých častiach výsledného kódu (viď nižšie)
        if(Huff_tree[i].znak == -1) Riadky[i] = 14;
        else Riadky[i] = 5;
    }
    for(int i=Huff_tree.size()-1; i>=0; i--) Riadky[i]+=Riadky[i+1];

    //skontrolujem, či na vrchu zásobníka nie je 0 a nemám už skončiť
    printf("PUSH -1\n");
    printf("READ\n");
    printf("PUSH -1\n");
    printf("ADD\n");
    printf("PUSH 8\n");
    printf("JGZ\n");
    printf("MUL\n");
    printf("PUSH -1\n");
    printf("READ\n");
    printf("PUSH 3\n");
    printf("JGZ\n");
    printf("PUSH 1\n");
    printf("PUSH 1000000\n");
    printf("JGZ\n");

    // Každý vrchol Huffmanovho stromu má priradenú časť programu, ktorá
    // spracováva čo sa v ňom má stať. Buď vypísať znak a skočiť na začiatok
    // alebo sa rozhodnúť podľa zvyšku po delení 2.
    for(int i=Huff_tree.size()-1; i>=0; i--) {
        if(Huff_tree[i].znak == -1) {
            //časť programu riešiaca vnútorný vrchol stromu
            printf("PUSH -1\n");
            printf("READ\n");
            printf("PUSH -1\n");
            printf("ADD\n");
            printf("PUSH 1\n");
            printf("JGZ\n");
            printf("MUL\n");
            printf("PUSH 2\n");
            printf("DIV\n");
            printf("PUSH %d\n",Riadky[Huff_tree[i].right+1]-Riadky[i]+3);
            printf("JGZ\n");
            printf("PUSH 1\n");
            printf("PUSH %d\n",Riadky[Huff_tree[i].left+1]-Riadky[i]);
            printf("JGZ\n");
        }
        else {
            //časť programu riešiaca list stromu
            printf("PUSH %d\n",Huff_tree[i].znak);
            printf("OUT\n");
            printf("PUSH 1\n");
            printf("PUSH %d\n",-5-Riadky[i+1]);
            printf("JGZ\n");
        }
    }
}
```