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

Vzorové riešenia 1. kola letnej časti

Šandyna

1. Žrúti

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

Keďže predátorov je vo všetkých kolách rovnako veľa, každé kolo zožerú rovnako veľa koristi (pokiaľ nevymreli od hladu). Množstvo vzniknutej koristi však závisí od toho, koľko jej je. Čím viac koristi je, tým viac jej vznikne. To však znamená, že pokiaľ v prvom kole množstvo koristi stúpne, musí stúpnuť aj v druhom kole a každom neskoršom, pretože ich ubudne rovnako veľa ale pribudne ich viac. Podobne, pokiaľ v prvom kole množstvo koristi klesne, musí jej množstvo klesnúť aj v ďalších kolách.

Na vyriešenie tejto úlohy nám teda stačí odsimulovať jediné kolo ekosystému a pozrieť sa, koľko koristi sme mali predtým a koľko máme po kole.

Ak máme teda na začiatku a predátorov a b kusov koristi, tak za jedno kolo nám najprv zmizne $a \cdot p$ kusov koristi, čiže nám ich ostalo $b - (a \cdot p)$ kusov. Táto korisť sa rozmnoží a budeme mať $(b - (a \cdot p)) \cdot k$ kusov koristi. Porovnáme teda b a $(b - (a \cdot p)) \cdot k$. Podľa toho, ktoré číslo je väčšie alebo či sú čísla rovnaké ľahko určíme, čo sa stane s ekosystémom. Musíme sa ale uistiť, že používame dostatočne veľké premenné, aby sa nám do nich číslo zmestilo.

Časová a pamäťová zložitosť

Riešenie úlohy zahŕňa len spočítanie a porovnanie niekoľkých čísel, ktorých je vždy rovnako veľa, bez ohľadu na vstupné dáta. Časová zložitosť bude konštantná, čiže $O(1)$. Taktiež si nemusíme pamätať nič iné, len daných pár čísel. Pamäťová zložitosť bude teda tiež konštantná: $O(1)$.

Listing programu (C++)

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

int main() {
    long long pocet_predatorov, pocet_koristi, p, k;
    cin >> pocet_predatorov >> pocet_koristi >> p >> k;
    long long zozrata_korist = pocet_predatorov * p;
    long long nova_korist = (pocet_koristi - zozrata_korist) * k;
    if(nova_korist == pocet_koristi) {
        cout << "rovnovaha\n";
    } else if(nova_korist > pocet_koristi) {
        cout << "rastie\n";
    } else if(nova_korist < pocet_koristi) {
        cout << "vymrie\n";
    }
}
```

Listing programu (Python)

```python
a, b, p, k = map(int, input().split())
c = (b - a * p) * k
if (c > b):
    print('rastie')
elif (c < b):
    print('vymrie')
else:
    print('rovnovaha')
```

Siegrift

2. Zajova paranoja

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

Zajo chce mať posteľ umiestnenú v bezpečí. To sú také časti územia, cez ktoré nemieri žiaden ostreľovač. Všimneme si, že priestor medzi dvomi susednými ostreľovačmi je vždy bezpečný. Podobne územie medzi **ne**susednými ostreľovačmi je vždy nebezpečné, lebo sa medzi nimi nachádza nejaký iný ostreľovač.

Pomalšie riešenie

Pre každú dvojicu susedných ostreľovačov zo severu, prejdeme každú dvojicu susedných ostreľovačov z východu. Šírku medzery medzi ostreľovačmi zo severu označme a, šírku medzery medzi ostreľovačmi z východu označme b. Časť miestnosti ohraničená týmito dvojicami je tvorená obdĺžnikom so stranami a a b. Obsah tohto obdĺžnika je zrejme $a \cdot b$. Pri prechádzaní všetkými týmito možnosťami, si počítame maximálny obsah, ktorý sme doposiaľ našli. Nakonci iba vypíšeme toto číslo. Takéto riešenie má časovú zložitosť $O(m \cdot n)$, lebo týchto dvojíc je rádovo $m \cdot n$ a každú spracujeme v konštantnom čase. Pretože si pamätáme celý vstup, pamäťová zložitosť je stále $O(m + n)$.

Trik na ošetrenie okrajov

Treba si ešte dať pozor na to, že ideálne miesto pre posteľ môže byť aj pri okrajoch miestnosti. Toto treba ošetriť buď manuálne (pridať podmienky), alebo si pomôžeme trikom. Vyrobíme si dvoch ostreľovačov v každom smere takých, že budú na okrajoch miestnosti. Riešenie bude rovnako správne a rýchle, ale sprehľadníme si tým náš kód.

Všimnime si tiež, že pri všetkých premenných okrem výslednej plochy nám stačí použiť v C++ typ `int`, pretože podľa zadania sa nám do neho všetko zmestí. Pri výslednej ploche ale násobíme dve čísla, ktoré môžu byť veľké až 10^6 a vtedy by nám už `int` nestačil.

Listing programu (C++)

```
#include <iostream>
#include <vector>   // namiesto vectora mozeme pouzit aj obycajne pole
using namespace std;

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

    // zadefinujeme vstupne polia aby sa nam tam zmestili 'okraje'
    vector<int> sever(m + 2);    // pozicie okien na severe
    vector<int> vychod(n + 2);  // pozicie okien na vychode
    // nastavime 'okraje'
    sever[0] = vychod[0] = 0;
    sever[m + 1] = x;
    vychod[m + 1] = y;

    for (int i = 1; i <= m; ++i) cin >> sever[i];
    for (int i = 1; i <= n; ++i) cin >> vychod[i];

    long long odpoved = -1;     // cast izby vacsia ako -1 sa urcite najde

    for (int sever_i = 1; sever_i < sever.size(); ++sever_i) {
        for (int vychod_i = 1; vychod_i < vychod.size(); ++vychod_i) {

            int dlzka_sever = sever[sever_i] - sever[sever_i - 1];
            int dlzka_vychod = vychod[vychod_i] - vychod[vychod_i - 1];
            odpoved = max(odpoved, (long long) dlzka_sever * dlzka_vychod);

        }
    }

    cout << odpoved << endl;

    return 0;
}
```

Vzorák

Na plný počet bodov bolo nutné spraviť ešte jeden myšlienkový krok. Naše pôvodné riešenie prechádzalo zbytočne veľa možnosťami. Stačilo si uvedomiť, že najväčšie územie sa vždy nachádza na prieniku území medzi dvomi najvzdialenejšími ostreľovačmi na východe a dvomi najvzdialenejšími na severe. Počítanie maximálneho územia je preto ekvivalentné s hľadaním maximálnej vzdialenosti medzi ostreľovačmi zo severu a z východu. Hľadané územie je potom rovné súčinu týchto čísel. Časová zložitosť tohto riešenia je teda $O(m + n)$, pretože stačí raz prejsť cez obe polia. Pamäťová zložitosť je stále $O(m+n)$, lebo si musíme pamätať vstup (tú už preto nezlepšíme).

Listing programu (C++)

```
#include <iostream>
#include <vector>   // namiesto vectora mozeme pouzit aj obycajne pole
using namespace std;

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

    // zadefinujeme vstupne polia aby sa nam tam zmestile 'okraje'
```

```
    vector<int> sever(m + 2);    // pozicie okien na severe
    vector<int> vychod(n + 2);  // pozicie okien na vychode
    // nastavime 'okraje'
    sever[0] = vychod[0] = 0;
    sever[m + 1] = x;
    vychod[m + 1] = y;

    for (int i = 1; i <= m; ++i) cin >> sever[i];
    for (int i = 1; i <= n; ++i) cin >> vychod[i];

    // vypocitame maximalnu vzdialenost medzi susednymi ostrelovacmi na severe
    long long sever_max = -1;
    for (int i = 0; i < sever.size(); ++i) {
        sever_max = max(sever_max, (long long) sever[i] - sever[i-1]);
    }

    // vypocitame maximalnu vzdialenost medzi susednymi ostrelovacmi na vychode
    long long vychod_max = -1;
    for (int i = 0; i < vychod.size(); ++i) {
        vychod_max = max(vychod_max, (long long) vychod[i] - vychod[i-1]);
    }

    // uzemie je velkostou sucinu tychto cisel
    cout << sever_max * vychod_max << endl;

    return 0;
}
```

Listing programu (Python)

```
# rozsekame vstupny riadok podla medzier, skonvertujeme na inty a ulozime do premennych.
x, y, m, n = map(int, input().split())

# pridame 'okraje' miestnosti na severnej strane
sever = [0] + list(map(int, input().split())) + [x]

# pridame 'okraje' miestnosti na vychodnej strane
vychod = [0] + list(map(int, input().split())) + [y]

# inicializujeme premenne na nieco male
sever_max, vychod_max = -1, -1

# zistime maximum na severe
for i in range(1, len(sever)):
    sever_max = max(sever_max, sever[i] - sever[i-1])

# zistime maximum na vychode
for i in range(1, len(vychod)):
    vychod_max = max(vychod_max, vychod[i] - vychod[i-1])

# odpovedou je obldznik zo zistenymi rozmermi
print(sever_max * vychod_max)
```

Čerešnička na torte

Na získanie plného počtu bodov stačilo predošlé riešenie, avšak aj to sa dá ešte zlepšiť. Už nevieme zlepšiť časovú zložitosť (lebo musíme aspoň načítať vstup), vieme však zlepšiť pamäťovú zložitosť. Na vyriešenie problému totiž hľadáme maximum z nejakého počtu čísel, a to vieme robiť popri tom, ako načítavame vstup. Takéto riešenie má zložitosti $O(m+n)$ časovú a $O(1)$ pamäťovú.

Listing programu (C++)

```
#include <iostream>
#include <vector>   // namiesto vectora mozeme pouzit aj obycajne pole
using namespace std;

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

    long long sever_max = -1;
    long long vychod_max = -1;

    // potrebujeme si pamatat predchadzajuce hodnoty, aby
    // sme vedeli zistit medzeru medzi nim a sucasnym ostrelovacom
    // na zaciatku ich nastavime na zaciatok miestnosti
    long long sever_pred = 0, vychod_pred = 0;

    // najdeme maximum na severe
    for(int i = 0; i < m; ++i) {
        long long teraz;
        cin >> teraz;
        // upravime maximum
        sever_max = max(sever_max, teraz - sever_pred);
        // cislo, ktore sme prave spracuvali, bude v dalsom prechode cyklom predchadzajuce
        sever_pred = teraz;
    }

    // najdeme maximum na vychode
    for(int i = 0; i < n; ++i) {
```

```
        long long teraz;
        cin >> teraz;
        // upravime maximum
        vychod_max = max(vychod_max, teraz - vychod_pred);
        // cislo, ktore sme prave spracuvali, bude v dalsom prechode cyklom predchadzajuce
        vychod_pred = teraz;
    }

    // musime este manualne skontrolovat druhy kraj(koniec) miestnosti
    // premenne 'sever_pred' a 'vychod_pred' obsahuju posledne nacitane cislo
    sever_max = max(sever_max, x - sever_pred);
    vychod_max = max(vychod_max, y - vychod_pred);

    // vypiseme obsah hladaneho uzemia
    cout << sever_max * vychod_max << endl;

    return 0;
}
```

Kubo

3. Zanedbané zastávky

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

Aj v tejto sérií sa tretia úloha dala vyriešiť prekvapivo jednoduchým algoritmom so zaujímavou myšlienkou.

Je zjavné, že niekedy nám nastane deň, kedy pri niektorej linke nebude prerábaná ani jedna zastávka. Keďže každý deň prerobíme aspoň jednu zastávku, po n dňoch budú určite všetky hotové.

Podobne sa môžeme pozrieť na nejakú konkrétnu linku s i zastávkami. Podobnú úvahu ako pre celé Kocúrkovo vieme spraviť aj pre túto linku. Na trase tejto linky najneskôr v i-ty deň nebude prerábaná ani jedna zastávka. Označme si k počet zastávok na *najkratšej* linke. Na tejto linke sa najneskôr k ty deň nebude prerábať zastávka, tým pádom odpoveď na úlohu bude najviac k.

Teraz si ukážeme že plán rekonštrukcie dá naozaj spraviť tak, že prvá linka bez prerábanej zastávky sa zjaví až v k-ty deň.

Úvaha

Pre lepšiu čitateľnosť si očíslujeme zastávky 0 až $n-1$ namiesto 1 až n.

Predstavme si, že od prvých dvoch Kocúrkovských zastávok jazdia linky dĺžky k. Prvá linka jazdí medzi zastávkami 0 a $k-1$. Keďže máme k dní, k zastávok a chceme každý deň rekonštruovať aspoň jednu zastávku, tak môžeme napríklad rekonštruovať i-tu zastávku v deň číslo i.

Teraz sa pozrime na druhú linku, ktorá jazdí medzi zastávkami 1 až k. Rekonštrukciu zastávok 1 až $k-1$ už máme naplánovanú z prvej linky, a chýba v nej len deň číslo 0. Nemáme teda inú možnosť, ako rekonštruovať zastávku k v deň 0.

Podobne, ak by sme mali linku aj medzi zastávkami 2 a $k+1$, tak zastávka $k+1$ by musela byť rekonštruovaná v ten istý deň ako zastávka 1, čiže v deň 1. V tomto vzore vieme ľahko pokračovať aj ďalej, vo všeobecnosti sa zastávka i bude rekonštruovať v deň $i \bmod k$, čo znamená zvyšok po delení i číslom k.

Takýto plán nám zaručí, že ľubovoľných k po sebe idúcich zastávok bude rekonštruovaných v každom dni od 0 až po $k-1$ v nejakom poradí. Tým pádom ľubovoľná linka dĺžky aspoň k bude mať počas každého z prvých k dní rekonštruovanú aspoň jednu zastávku.

Implementácia

Zo vstupu potrebujeme okrem počtu zastávok n a počtu liniek m len dĺžku najkratšej linky k. Počet dní na výstupe potom bude práve k. Plán rekonštrukcie vyrobíme jednoducho. Zastávku číslo i budeme prerábať v dni $i \bmod k$.

Časová zložitosť takéhoto riešenia je $O(m)$ na načítanie vstupu a $O(n)$ na vypísanie odpovede, dokopy $O(n+m)$.

Pamäťová zložitosť je $O(1)$. Vo vzorovom kóde kóde v C++ si môžeme všimnúť, že na výpočet najkratšej dĺžky linky a vypísanie plánu rekonštrukcie nepotrebujeme žiadne polia [1].

Listing programu (C++)

```
#include <iostream>
#include <algorithm> // tu je definovane min

using namespace std;

int main() {

    int n, m;
    cin >> n >> m;
```

[1] Kód v Pythone ich pre prehľadnosť používa.

```
    // prve ohranicenie riesenia
    int k = n;

    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        // ak mame kratsiu linku nez doteraz tak zapiseme jej dlzku
        k = min(k, b-a+1);
    }

    cout << k << endl;

    for (int i=0; i<n; i++) {
        if (i != 0) {
            cout << " ";
        }
        cout << i % k;
    }
    cout << endl;
}
```

Listing programu (Python)

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

# nacitame konecne zastavky, ulozime ich ako dvojice v liste
konecne_zastavky = [input().split() for _ in range(m)]

# tieto dvojice postupne odcitame od seba
# dostaneme dlzky liniek a z nich vezmeme najkratsiu
k = min([int(b) - int(a) + 1 for a, b in konecne_zastavky])

print(k)

plan = [i % k for i in range(n)]

print(' '.join(map(str, plan)))
```

MikX

4. Zakázané ohňostroje

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

Na začiatku si ukážeme jedno korektné riešenie, ktoré síce správne vyrieši každý vstup, no je **príliš** pomalé pri veľkých rozmeroch. Na toto riešenie ďalšie nenaväzujú, preto, ak chcete, môžete toto preskočiť (bude označené hviezdičkou). Následne si ukážeme, ako bolo možné využiť spomínané obmedzenia na jednotlivé sady a na záver si predstavíme algoritmus, ktorý rýchlo a správne vyrieši všetky sady vstupov.

Pri implementácii všetkých riešení používame dva triky často používané pri úlohách s mriežkami: zarážky a polia zmien súradníc k susedom. Ak ste sa s týmito trikmi doteraz nestretli, viac sa o nich dozviete v Kuchárke[2].

Bratislava ako bludisko (*)

Predstavme si Bratislavu, ktorú dostaneme na vstupe, ako bludisko. Ako nájsť cestu z bludiska? Jedna z možností je predstaviť si, že máme k dispozícii niť, ktorou si značíme kade chodíme[3]. Na každej križovatke odbočíme doprava. Akonáhle prídeme do slepej uličky alebo na miesto, kde sme už boli, vrátime sa späť a pôjdeme odbočkou, ktorou sme ešte nešli a zároveň je najviac napravo. Tento postup opakujeme, kým sa z bludiska nedostaneme von.

Pokúsme sa aplikovať tento postup na hľadanie najlacnejšej cesty na intrák. Potrebujeme si pri každom odbočovaní rátať sumu doteraz zaplatených pokút, a že či sme prešli cez nejaké kontroly. Ako niť použijeme pole booleanov `vis`. Na riešenie použijeme rekurzívnu[4] funkciu, ktorá bude prehľadávať všetky cesty. Nech sme na políčku (x, y). Ak je toto cieľové políčko, tak sme vyhrali a stačí nám obnoviť globálnu premennú, v ktorej si udržiavame cenu doteraz najlacnejšej nájdenej cesty do cieľa. V opačnom prípade skontrolujeme, že či toto políčko náhodou nie je Dunaj alebo políčko, na ktorom sa nachádza naša niť. V takom prípade nechcem ďalej pokračovať v ceste a vraciame sa späť. Inak nastavíme `vis[x][y] = true` a skúšame sa rekurzívne zavolať na všetkých susedov tohto políčka. Vracanie sa po niti predstavujú návraty z rekurzívnych volaní. Pri návrate z rekurzívneho volania sa pokúsime ísť ďalším smerom. Ak už sme prešli všetky smery, tak nastavíme `vis[x][y] = false`, čo prestavuje to, že niť sa už na tomto políčku nebude nachádzať, lebo ideme späť. Tento algoritmus sa volá aj **backtracking**.

Časová a pamäťová zložitosť

Riešenie backtrackingom je pekné a jednoduché, ale má jednu nevýhodu – je **pomalé** (samozrejme aj pomalé,

[2] https://ksp.sk/kucharka/mriezky_implementacia
[3] Ariadnina niť
[4] https://ksp.sk/kucharka/rekurzia

správne riešenie dostane nejaké body). Cesta, pri ktorej Emo najmenej minie môže byť ozaj ktorákoľvek, a teda musíme vyskúšať **všetky**, ktoré existujú. Na každej križovatke rozoberieme 4 možnosti. Pre každú možnosť rozoberieme na ďalšej križovatke 4 možnosti, atď. Časová zložitosť tohto riešenia je teda $O(4^{rs})$. Pamäťová zložitosť je $O(r \cdot s)$ – veľkosť mapy Bratislavy a hĺbka zásobníka pri rekurzii.

Výsledný program by mohol vyzerať takto:

Listing programu (C++)

```
// B = mapa Bratislavy
// vis = nasa niť, pole vis urcuje ci sme na krizovatke uz boli
// surX = suradnica x, surY obdobne
// vysledok = cena doteraz najlacnejsej cesty
// pair<> I = suradnice internatu

long long vysledok = NEKONECNO;

void spocitaj(int surX, int surY, long long cena, bool presli_kontrolu) {
        // Ak som tu uz na tejto ceste boli alebo tu je dunaja.
        if (vis[surX][surY] || B[surX][surY] == DUNAJ) return;

        // Dosli sme uz do ciela?
        if (surX == I.first && surY == I.second) {
                // Ako vysledok zoberieme minimum z doterajsieho vysledku
                // a terajsej ceny, ku ktorej pripocitame pokutu
                // za ohnostroje v pripade, ze sme presli cez nejake kontroly.
                vysledok = min(vysledok, cena + (presli_kontrolu ? Po : 0));
                return;
        }

        // Zaznacime si, ze sme tu boli.
        vis[surX][surY] = true;

        // Odbocime postupne do kazdeho smeru.
        for(int i=0; i<4; ++i) {
                // Od nasho policka ideme smerom i, teda x zvysime o dx[i] a y o dy[i].
                // Cenu zvysime o pokutu za tento smer, co nam urci funkcia zisti_cenu.
                // Ked jej hodime krizovatku na ktorej sme a smer ktorym chceme ist.
                // Tiez musime updatovat flag s kontrolami.
                spocitaj(surX + dx[i],
                        surY + dy[i],
                        cena + zisti_cenu(B[surX][surY], i),
                        presli_kontrolu || (B[surX][surY] == KONTROLA));
        }

        // Odchadzame z krizovatky, zaznacime, ze sme tu este neboli,
        // nech sem vieme prijst znova.
        vis[surX][surY] = false;
}
```

Predstava Bratislavy ako grafu

Bratislava sa dá predstaviť ako orientovaný ohodnotený graf[5]. Križovatky sú vrcholy. Hrana z križovatky a vedie do križovatky b, ak sú tieto križovatky susedné. Ohodnotenia hrán sú pokuty, ktoré Emo zaplatí, ak prejde medzi týmito dvoma križovatkami. Hrany sú orientované, lebo hoci sa Emo vie dostať aj opačne, môže to byť za inú pokutu. Ak je odbočenie zadarmo (v smere prikázaného smeru, prípadne z križovatky bez obmedzení), pridáme hranu s nulovou cenou. Z Dunaja hrany nepridávame. Kontroly zatiaľ neuvažujeme.

V tejto interpretácii je našou úlohou nájsť najlacnejšiu cestu v grafe, na čo existujú rozumne rýchle algoritmy.

Čo ak pokuty neexistujú?

V prvej sade vstupov sa pokuty nenachádzali. V našej predstave to znamená, že všetky ceny v grafe sú nulové, a teda celková cena bude buď `0` ak existuje cesta na intrák, prípadne `-1` ak neexistuje.

Ako zistiť, či nejaká cesta existuje? Jedna z možností je použiť algoritmus prehľadávania do hĺbky. Tento algoritmus sa podobá backtrackingu, ktorý sme si vysvetlili pri pomalom riešení. Jediný rozdiel je, že keď pri vynáraní sa z rekurzie opúšťame nejaký vrchol, necháme ho označený ako navštívený. Toto nám zaručí, že každý vrchol navštívime iba raz a časová zložitosť bude $O(r \cdot s)$. Takéto zrýchlenie si môžeme dovoliť urobiť vďaka tomu, že teraz na nájdenie najlacnejšej cesty nepotrebujeme vyskúšať všetky možnosti, keďže každá cesta je rovnako dobrá (má cenu nula). Ako a prečo to celé funguje sa dočítate v Kuchárke[6].

Inou rovnako rýchlou možnosťou je použiť iný základný grafový algoritmus – prehľadávanie do šírky[7] (známe tiež pod anglickým názvom *breadth-first search*, BFS).

Čo sa zmení, keď pridáme pokuty veľkosti 1?

Nakoľko už budeme chcieť nájsť aj reálne najlacnejšiu a nielen tak hocijakú cestu, prehľadávanie do hĺbky

[5] https://ksp.sk/kucharka/grafy_uvod
[6] https://ksp.sk/kucharka/dfs
[7] https://ksp.sk/kucharka/bfs

nám už nepomôže. Prehľadávanie do šírky však vieme rozumne vylepšiť tak, aby nám pre dva druhy cien – jednu nulovú a druhú nejakú inú (napr. 1) hľadalo najlacnejšiu cestu.

Princíp prehľadávania do šírky sa opiera o to, že sa používa fronta. Vďaka nej vieme nájsť najlacnejšiu cestu v grafe, kde sú všetky hrany rovnakej dĺžky. Pri BFS platí, že v každom momente sú vrcholy vo fronte usporiadané podľa ceny, ktorou sa do vrcholu vieme dostať.

Predstavme si teraz, že nebudeme využívať frontu, ale **obojsmernú frontu**. Táto dátová štruktúra nám umožňuje pridávať prvky aj na začiatok, aj na koniec. Ak vedie do vrcholu nulová hrana, pridáme ho na začiatok. Ak doňho vedie jednotková hrana, pridáme ho na koniec. Všimnite si, že takýmto spôsobom udržiavame vrcholy vo fronte utriedené podľa dĺžky najkrajšej cesty.

Tento algoritmus sa tiež zvykne označovať ako **0/1 BFS**. Časová zložitosť je rovnaká ako pri klasickom BFS, $O(r \cdot s)$, pamäťová takisto.

Ale pokuty predsa môžu byť ľubovoľné. . .

Teraz sa pozrime na prípad, keď dĺžky hrán môžu byť ľubovoľné. Potrebujeme dátovú štruktúru, do ktorej vieme rýchlo pridávať prvky a vyberať z nej najmenší prvok. V prípade nulových a jednotkových hrán nám obojsmerná fronta postačila[8]. Vo všeobecnom prípade budeme potrebovať zbraň vyššieho kalibru. Vhodnou dátovou štruktúrou je napríklad halda. Použitím haldy vlastne získame Dijkstrov algoritmus[9] na hľadanie najlacnejšej cesty. Dostaneme tak rýchle riešenie v časovej zložitosti $O(rs \cdot \log(rs))$ (počet vrcholov je $r \cdot s$ a počet hrán je $4 \cdot r \cdot s$). Pamäťová zložitosť je $O(r \cdot s)$ – veľkosť mapy a v najhoršom prípade aj veľkosť haldy.

Čo s kontrolami?

Stále sme si však od testovača zarobili len 4 body zo 6-tich. Potrebujeme ešte vyriešiť ako naložiť s križovatkami, kde prebiehajú kontroly.

Najjednoduchšie riešenie je uvedomiť si nasledovné: Emo buď dostane pokutu za ohňostroj, alebo nedostane. Celé prechádzanie bludiskom môžeme extrahovať do funkcie a túto dvakrát zavolať – raz s tým, že nebudeme chodiť cez žiadne kontroly (a nezaplatíme pokutu za ohňostroj) a raz s tým, že môžeme prejsť cez ľubovoľne veľa kontrol, ale na konci pripočítame veľkosť pokuty za ohňostroj. Časovú zložitosť to nezhorší. Celé to robíme len dvakrát, čo je konštanta. Navyše, v podstate rozoberáme len dva jednoduché prípady.

Na demonštráciu tohto riešenia uvádzam zdrojový kód, ktorý podrobnejšie ukazuje, čo sme si práve opísali. Funkcii `spocitaj` realizuje Dijkstrov algoritmus. Premenná `flag` hovorí, či má ísť aj cez kontroly alebo sa k nim má správať rovnako ako ku križovatkám s Dunajom.

Listing programu (C++)

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

#define ll long long

#define DUNAJ '~'
#define EMO 'E'
#define KONTROLA 'K'
#define INTRAK 'I'
#define KONTROLA_JE_ZLA false
#define KONTROLA_JE_OK true
#define NEKONECNO 2000000047000000047LL

using namespace std;

// pair x-ovej, y-ovej suradnice a ceny
struct vec {
        int surX, surY;
        ll cena;

        // nejake konstruktory, ktore nam ulahcia pracu s 'vec'
        vec(int _surX, int _surY, ll _cena) : surX(_surX), surY(_surY), cena(_cena) {}
        vec(pair<int, int> _surXY, ll _cena) : surX(_surXY.first), surY(_surXY.second), cena(_cena) {}

        // aby halda usporaduvala od najnizsej .cena-y
        bool operator< (const vec & dalsi) const {
                return dalsi.cena < this->cena;
        }
};

const char krizovatky[] = {'L', 'P', 'H', 'D'};
// vsimnite si, ze dx[] a dy[] koresponduju so smermi v krizovatky[]
const int dx[] = {0, 0, -1, 1}; // dx[i] je posunutie v x-ovej suradnici pri smere i
const int dy[] = {-1, 1, 0, 0}; // podobne dy predstavuje posunutie v y-ovej suradnici
// pokuty[i] = pokuta za porusenie prik. smeru krizovatky[i]
ll pokuty[] = {0LL, 0LL, 0LL, 0LL};
```

[8] Namiesto obojsmernej fronty sme mohli použiť aj dva zásobníky, aby sme udržiavali prvky utriedené podľa ceny

[9] https://ksp.sk/kucharka/dijkstra

```
ll Po; // pokuta za ohnostroj
int r, s;
pair<int, int> E, I; // suradnice Ema a Intraku
vector< vector<char> > B; // mapa Bratislavy

ll cena(char policko, int smer) {
	if (policko == krizovatky[smer])
		return 0LL;

	for(int i=0; i<4; ++i)
		if (policko == krizovatky[i])
			return pokuty[i];

	return 0LL;
}

// dijkstra
ll spocitaj(bool flag) {
	vector< vector<bool> > vis(r+2); // visited? aby som sa viac krat nepozeral na to iste
	for(int i=0; i<r+2; ++i) vis[i].resize(s+2, false);

	priority_queue< vec > PQ;
	// Emo sa zatial nepohol a nic nezaplatil
	PQ.push(vec(E, 0LL));
	while (!PQ.empty()) {
		vec p = PQ.top(); PQ.pop();

		if (vis[p.surX][p.surY]) continue;
		vis[p.surX][p.surY] = true;

		if (B[p.surX][p.surY] == DUNAJ) continue;
		if (flag == KONTROLA_JE_ZLA && B[p.surX][p.surY] == KONTROLA) continue;

		// dosli sme uz do ciela? ak hej tak isto s najlepsou cenou
		if (p.surX == I.first && p.surY == I.second)
			return p.cena;

		for(int i=0; i<4; ++i)
			// od nasho policka ideme smerom i, teda x zvysime o dx[i] a y o dy[i]
			// cenu zvysime o pokutu za tento smer, co nam urci funkcia cena ked
			// jej hodime krizovatku na ktorej sme a smer ktorym chceme ist
			PQ.push(vec(p.surX + dx[i],
				p.surY + dy[i],
				p.cena + cena(B[p.surX][p.surY], i)));
	}

	// ak sme tu, nevieme sa na intrak dostat
	return NEKONECNO;
}

int main() {

	// nacitavanie pokut a rozmerov mapy
	cin >> Po;
	for(int i=0; i<4; ++i) cin >> pokuty[i];
	cin >> r >> s;

	// mapu Bratislavy si zvacsime a inicializujeme na dunaj, cim pridame 'okraje'
	B.resize(r+2); for(int i=0; i<r+2; ++i) B[i].resize(s+2, DUNAJ);
	// postupne ju tam pridavame, predstavujeme si ju dokolecka obkolesenu dunajom
	for(int i=1; i<r+1; ++i) for(int j=1; j<s+1; ++j) {
		cin >> B[i][j];

		// zapamatame si ak sme narazili na Ema alebo Intrak
		if (B[i][j] == EMO)
			E = {i, j};
		else if (B[i][j] == INTRAK)
			I = {i, j};
	}

	// Emo bud neprejde kontrolou alebo prejde niekolkymi a zaplati pokutu za ohnostroj
	ll vysledok = min(spocitaj(KONTROLA_JE_ZLA), Po + spocitaj(KONTROLA_JE_OK));

	// ak je vysledok nekonecno, riesenie neexistuje, inak vypisem na co som prisiel
	cout << (vysledok < NEKONECNO ? vysledok : -1LL) << endl;

	return 0;

}
```

Iný pohľad na kontroly

Rovnako rýchle riešenie je vytvoriť si **2 vrcholy pre každú križovatku**. Jeden typ reprezentuje križovatky s tým, že sme na ceste k nim ešte žiadnou kontrolou neprešli a druhý reprezentuje, že sme už aspoň cez jednu kontrolu prešli. Keďže nevieme, či bude optimálne ísť cez kontroly, alebo nie, potrebujeme si pamätať oba vrcholy. V 3D sa to dá predstaviť ako keby sme urobili kópiu nášho grafu a položili ju nad náš graf. Takto dostaneme dvojúrovňový graf. Dolná úroveň sú vrcholy, ktoré hovoria, že sme ešte neprešli žiadnou kontrolou a horná úroveň sú vrcholy, ktoré hovoria, že už sme prešli aspoň jednou kontrolou. Vrcholy v rámci jednej úrovne budú pospájané hranami rovnakých cien, ako sme si popisovali (na oboch úrovniach). Jediná výnimka budú križovatky s kontrolami v dolnej úrovni. Ak sa dostaneme do križovatky s kontrolou v dolnej úrovni, tak potom

pokračujeme v našej ceste v hornej úrovni. Za prechod medzi úrovňami zaplatíme cenu pokuty za ohňostroj. Ďalej už pokračujeme iba po hornej vrstve.

Na tomto grafe spočítame Dijkstrovým algoritmom najlacnejšiu cestu na lacnejší vrchol internátu. Zaujíma nás totiž tá lacnejšia z týchto dvoch najlacnejších ciest na jednotlivé vrcholy. Je nám jedno na akej úrovni grafu skončíme.

5. Obezlička

Hodobox
(max. 8 b za popis, 7 b za program)

Stručný obsah podčlánkov je nasledovný:

- Hrubá sila: riešenie overovaním všetkých štvorcových plôch z ich ľavého horného rohu $O(n^5)$.
- Dá sa to aj múdrejšie: overovanie štvorcov zo stredu $O(n^4)$.
- Pozrime sa na to bližšie: správny odhad časovej zložitosti riešenia $O(n^2)$.

Hrubá sila

V tomto najjednoduchšom riešení by sme sa postupne chceli pozrieť na všetky štvorcové oblasti tabule, zistiť, ktoré z nich sú nepočmáranými pluskami a na základe toho zistiť, koľko bodov Roman dostane.

Každý štvorec (každé plusko) je na tabuli určený jeho **ľavým horným rohom a dĺžkou strany**. Ak si teda fixne zvolíme jedno políčko (i, j), chceme postupne skúšať dĺžky strany $3, 5, 7, ...$ (teda štvorce s ľavým horným políčkom (i, j) a pravým dolným políčkom postupne $(i + 2, j + 2), (i + 4, j + 4), ...$)

Ak máme vybraný štvorec s rohovými políčkami (i, j) a $(i + x, j + x)$, vieme ľahko overiť, či je tento štvorec nepočmáraným pluskom. Stačí prejsť jeho všetky políčka v cykle a overiť, že tie, ktoré sú v strednom riadku (teda $(i + x/2, j), (i + x/2, j + 1), (i + x/2, j + 2), ..., (i + x/2, j + x)$) alebo v stredom stĺpci (teda $(i, j + x/2), (i + 1, j + x/2), ...(i + x, j + x/2)$) musia byť popísané (`#`), a ostatné nepopísané (`.`).

Ako ale zistiť počet bonusových bodov? V zadaní sa písalo: "Za plus veľkosti v dostane Roman v bodov, no každé plus veľkosti aspoň 2 obsahuje vo svojom strede menšie pluská a za tie už Roman body nedstane."

Pre správny výpočet výsledku nám však stačí len zistiť počet štvorcov, ktoré sú pluskami. Neveríte?

Ak je na tabuli plusko veľkosti 1, nájdeme práve jeden štvorec (so stranou dĺžky 3), ktorý ho obsahuje. Ak je plusko veľkosti 2, vytvára tak na tabuli dve štvorcové oblasti, ktoré sú pluskami – jedna oblasť je štvorec so stranou dĺžky 5 a druhá oblasť je menší štvorec so stranou dĺžky 3, v strede väčšieho štvorca. Plusko veľkosti 3 nám vytvorí 3 štvorcové oblasti ktoré sú pluskami atď. Trik je v tom, že v tomto riešení dáme každému štvorcu, ktorý je pluskom len **jeden bod**. Väčšie pluská preto ohodnotíme presne toľkými bodmi, ako si to vyžadovalo zadanie.

Listing programu (C++)

```
#include <iostream>

using namespace std;

char tabula[2000][2000];

int je_stvorec_plusko(int r, int c, int v)
{
    for(int i=0; i<v; ++i)
        for(int j=0; j<v; ++j)
            //ak nie sme v strednom riadku/stĺpci a políčko je popísané
            if(i != v/2 && j != v/2 && tabula[r+i][c+j] == '#')
                return 0;
            //ak sme v strednom riadku/stĺpci a políčko je nepopísané
            else if ( (i == v/2 || j == v/2) && tabula[r+i][c+j] == '.')
                return 0;
    return 1;
}

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

    for(int i=0; i<n; ++i)
        for(int j=0; j<n; ++j)
            cin >> tabula[i][j];

    int body = 0;
    for(int i=0; i<n; ++i)
    {
        for(int j=0; j<n; ++j)
        {
            for(int v=3; i+v<=n && j+v<=n; v+=2)
            {
                body += je_stvorec_plusko(i, j, v);
            }
```

```
        }
    }

    cout << body << "\n";
}
```

Políčok je na tabuli n^2 a dĺžka strany každého štvorca môže byť rádovo až n. Overiť, či je štvorec nepočmárané plus nám v najhoršom prípade trvá toľko krokov ako je jeho obsah, čo je v najhoršom prípade n^2. Dokopy je teda počet krokov zhora ohraničený $n^2 \cdot n \cdot n^2$, teda časová zložitosť bude $O(n^5)$.

Pamätať si pritom musíme celú tabuľu, takže potrebná pamäť bude kvadratická v závislosti od rozmeru tabule – $O(n^2)$.

Riešenie vieme ešte zlepšiť niekoľkými trikmi. O veľa štvorcoch napríklad veľmi rýchlo zistíme, že sú počmárané; vtedy môžeme rovno prestať kontrolovať zvyšok štvorca. Dá sa ukázať, že toto vylepšenie zníži časovú zložitosť na $O(n^4)$. Takisto sa ľahko môže stať, že veľa štvorcov na tabuli spĺňa jednu z podmienok (každý štvorec na prázdnej tabuli má všetky nie-stredné riadky a stĺpce správne, a každý štvorec na úplne popísanej tabuli má všetky stredné riadky a stĺpce správne). Aby nás takéto vstupy menej trápili, mohli by sme napríklad na začiatok vyskúšať zopár náhodných políčok v strednom riadku/stĺpci a niekoľko náhodných políčok mimo, či naše plusko nekazia. Existuje viacero takýchto podobných vylepšení, ktoré mohli riešenie urýchliť, a získať teda trochu viac bodov na testovacích vstupoch. Napriek tomu takéto optimalizácie väčšinou nemenia časovú zložitosť algoritmu, keďže pri odhade časovej zložitosti uvažujeme ten najhorší možný prípad.

Dá sa to aj múdrejšie?

Vyššie uvedené riešenie sa síce dalo všelijakými ad-hoc trikmi vylepšovať, nevie sa však nijak zbaviť základnej myšlienky, že pre každé z n^2 políčok skúša rádovo n dĺžok strán štvorcov, a tie (v najhoršom prípade) celé prekontroluje. Pre n blížiace sa k 2000 je teda riešenie príliš pomalé.

Zväčšovanie štvorcov z ľavého horného rohu má principiálne ten problém, že ak zmeníme dĺžku strany štvorca, ktorý chceme overiť, veľa políčok v ňom bude zohrávať inú úlohu. Popísané políčka, ktoré nám pri jednej dĺžke strany zavadzali, môžu byť pri ďalšej dĺžke strany práve tými potrebnými políčkami v strednom riadku alebo stĺpci, a naopak. Kvôli tomu musíme pri zmene dĺžky strany štvorca prakticky všetky políčka znova overovať. Jednoduchšie povedané, vždy keď zmeníme dĺžku strany štvorca ktorý chceme overiť, a vždy keď zmeníme políčko, ktoré skúšame ako ľavý horný roh, zahadzujeme všetku doteraz získanú informáciu, a preto musíme overovať veľa políčok, aby sme zistili, či nový štvorec je alebo nie je nepočmárané plus.

Vieme nejakým spôsobom upraviť náš prvý algoritmus tak, aby sme využili predošlé informácie, ktoré máme z overovania menších štvorcov?

Vieme – stačí si štvorec definovať pomocou súradníc jeho stredného políčka a dĺžky jeho strany. Opäť budeme teda overovať všetky štvorce – najprv si zvolíme stred štvorca (i, j) a potom budeme zväčšovať jeho dĺžku strany $v = 3, 5, 7, ..., n$. Keď vieme, že štvorec so stranou dĺžky v je nepočmárané plus, a chceme overiť štvorec so stranou dĺžky $v + 1$, máme už zaručené, že celé "vnútro" je správne, a stačí nám teda overiť "nové" políčka – teda okraje štvorca. Keď ale zistíme, že štvorec je počmárané plusko, všetky väčšie štvorce s týmto stredom budú tiež počmárané a môžeme tak prestať overovať štvorce s týmto stredom.

Listing programu (C++)

```cpp
#include <iostream>

using namespace std;

char tabula[2000][2000];

bool je_stvorec_plusko(int r, int c, int v)
{
    //pozrieme sa či nové políčka v strednom riadku a stĺpci sú popísané
    if(tabula[r+v][c]!='#' || tabula[r-v][c]!='#' || tabula[r][c-v]!='#' || tabula[r][c+v]!='#')
        return false;

    //skontrolujeme či nové vonkajšie riadky a stĺpce sú nepopísané
    for(int i=1; i<=v; ++i)
    {
        if(tabula[r+i][c-v]=='#'||tabula[r+i][c+v]=='#')
            return false;
        if(tabula[r+v][c-i]=='#'||tabula[r+v][c+i]=='#')
            return false;
        if(tabula[r-i][c-v]=='#'||tabula[r-i][c+v]=='#')
            return false;
        if(tabula[r-v][c-i]=='#'||tabula[r-v][c+i]=='#')
            return false;
    }

    return true;
}
```

```
int main()
{
    //vstup môže mať až 4,000,000 znakov, zrýchlime načítavanie
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);

    int n, body = 0;
    cin >> n;

    for(int i=0; i<n; ++i)
        for(int j=0; j<n; ++j)
            cin >> tabula[i][j];

    for(int i=0; i<n; ++i)
    {
        for(int j=0; j<n; ++j)
        {
            if(tabula[i][j]=='.') continue;
            //zväčšujeme dĺžku strany o 1 kým rám štvorca so stredom v [i][k] sedí
                        int v = 1;
                while(i+v<n && j+v<n && i-v>=0 && j-v>=0 && je_stvorec_plusko(i,j,v)) ++v;
            body += v-1;
        }
    }

    cout << body << "\n";

    return 0;
}
```

Ako vieme zatiaľ zhora odhadnúť zložitosť tohto riešenia? Pre každé políčko, ktorých je n^2, máme cyklus ktorý beží (v teoretickom, najhoršom možnom prípade) najviac n-krát, a zakaždým kontroluje nové políčka štvorca, teda okraj štvorca. Ten má rádovo n políčok. Dokopy teda dostávame horný odhad $O(n^2 \cdot n \cdot n) = O(n^4)$.

Pozrime sa na to bližsie

Napriek tomu, že náš odhad $O(n^4)$ je správny (algoritmus skutočne urobí na tabuli so stranou n najviac rádovo n^4 operácií), tento odhad **nie je tesný**. V tejto časti si ukážeme, že v skutočnosti je vyššie popísaný algoritmus omnoho rýchlejší.

Pri políčkach, ktoré nie sú stredom žiadneho nepočmáraného plus, naše overovanie skončí veľmi rýchlo – najneskôr po skontrolovaní prvých deviatich políčok, teda v konštantnom čase. Overovanie takýchto políčok teda **dokopy** zaberie $O(n^2)$ času.

Keď skúšame políčko, ktoré je stredom nepočmáraného plus s ramenom v (ale už nie väčšieho), skontrolujeme štvorec so stranou $2v+1$, čo nám zaberie $(2v+1)^2$ krokov. Následne ešte skúsime plusko zväčšiť, ale nepodarí sa nám to a prestaneme. V najhoršom prípade teda prejdeme všetky políčka štvorca so stranou $2v+3$. Dokopy teda urobíme rádovo v^2 krokov.

Inak povedané, ak sú na tabuli pluská veľkostí $v_1, v_2, ..., v_k$ (pričom počítajme len tie najväčsie; nepočítajme tie, ktoré sú v strede väčšieho pluska), na ich overenie bude tento program potrebovať rádovo ${v_1}^2 + {v_2}^2 + ... + {v_k}^2$ krokov.

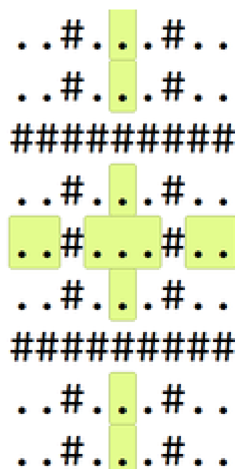
Teda náš algoritmus vykoná rádovo toľko operácií, koľko je súčet plôch nepočmáraných plusiek na tabuli. Má teda zložitosť opísateľnú ako $O(n^2 + \text{plocha plusiek})$ ($O(n^2)$ kvôli načítaniu vstupu a políčkam, ktoré nie sú stredom žiadneho pluska). A toto je pre náš algoritmus veľmi prospešné – pluská si totiž, z hľadiska algoritmu, navzájom zavadzajú. Chceme už vlastne len dokázať, že nepočmárané pluská sa nevedia ľubovoľne prekrývať a na tabuli ich teda bude relatívne málo.

Zoberme si teda ľubovolné políčko (i, j). Pozrieme sa naň toľkokrát, do koľkých nepočmáraných plusiek patrí. Ak je na (i, j) znak `#`, môže to byť buď stred pluska (teda patrí do práve jedného), alebo byť hornou/dolnou stranou stredného stĺpca, resp. ľavou/pravou stranou stredného riadku. V tejto situácii môže zároveň patriť do najviac jedného ďalšieho nepočmáraného pluska.

```
..#....
..#..#.
#######
..#..#.
..#....
.###...
..#....
```

Na tomto obrázku sú farebne vyznačené tie #, ktoré patria do dvoch nepočmáraných plusiek (a teda náš algoritmus sa na ne pozrie dvakrát). Žiadnu mriežku však ako okraj platného pluska nebude skúšať viac ako dvakrát.

Podobná úvaha platí aj pre nepopísané políčka – každé nepopísané políčko môže patriť do najviac štyroch nepočmáraných plusiek (do ľavej hornej časti jedného, pravej dolnej časti druhého, ...).



Na tomto obrázku sú farbene vyznačené tie bodky, ktoré patria do viacerých nepočmáraných plusiek. Pritom všetky patria do práve dvoch plusiek s výnimkou bodky v úplnom strede obrázku, ktorá patrí do všetkých štyroch plusiek.

Keďže pri overovaní jedného pluska sa na každé jeho políčko pozrieme len raz, a každé políčko v najhoršom prípade patrí do štyroch rôznych nepočmáraných plusiek, vyplýva z toho, že na každé políčko tabule sa pri overovaní pozrieme najviac štyrikrát. Políčok je n^2, a teda časová zložitosť je $O(n^2)$.

Opäť si pamätáme celú tabuľu, teda pamäť je stále $O(n^2)$

Baklažán

6. O introvertných a extrovertných holuboch

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

Vždy, keď Dávidovi priletí nejaký holub s extroverciou e, musí ho Dávid nasťahovať do holubníka, v ktorom je presne e iných holubov. Niekedy sa môže stať, že Dávid má na výber viacero holubníkov s presne e holubmi.

Všimnime si, že všetky holubníky s e holubmi sa pre účely našej úlohy správajú úplne rovnako (do každého z nich môžeme strčiť holuba s extroverciou e a ak to urobíme, stane sa z neho holubník s $e + 1$ holubmi). To znamená, že vždy keď má Dávid na výber z viacerých holubníkov, môže si vybrať ľubovoľný z nich a určite tým nič nepokazí.

Priamočiare riešenie

Asi najpriamočiarejšie je celú situáciu simulovať. V jednom poli si budeme pre každý holubník pamätať, koľko je v ňom holubov. Vždy, keď priletí holub, nájdeme prvý holubník s vhodným počtom iných holubov a strčíme ho tam. Budeme si musieť pamätať n holubníkov (pre prípad, že by nám priletelo n holubov s extroverciou 0), teda pamäťová zložitosť bude $O(n)$. Hľadanie vhodného holubníka nám pri každom holubovi bude trvať v najhoršom prípade $O(n)$, teda časová zložitosť bude $O(n^2)$.

Šetrenie pamäťou

Ľahký spôsob, ako toto riešenie vylepšiť, je nepamätať si n holubníkov, ale len toľko, koľko potrebujeme. V najhoršom prípade to síce nepomôže (lebo v najhoršom prípade aj tak potrebujeme všetkých n holubníkov), ale vyrieši to tretiu sadu testovacích vstupov, kde máme sľúbené, že stačí použiť nanajvýš 1 000 holubníkov.

Samozrejme, na začiatku nevieme, koľko holubníkov budeme potrebovať. Preto si nebudeme pamätať žiaden. Vždy, keď priletí holub s extroverciou 0, otvoríme mu nový holubník a začneme si pamätať počet holubov v ňom. Na tento účel sa veľmi hodí pole s možnosťou pridávania prvkov na koniec, napríklad `vector` v C++, alebo `ArrayList` v Jave.

Listing programu (C++)

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

```
int main()
{
  int n;
  scanf("%d", &n);
  vector<int> holubnik;
  for(int i=0; i<n; i++)
  {
    int e;
    scanf("%d", &e);
    if(e == 0)
    {
      printf("%d", holubnik.size()+1);
      holubnik.push_back(1);
    }
    else
    {
      for(int j=0; j<holubnik.size(); j++)
      {
        if(holubnik[j] == e)
        {
          printf("%d", j+1);
          holubnik[j]++;
          break;
        }
      }
    }
    if(i+1 < n) printf(" ");
    else printf("\n");
  }
  return 0;
}
```

Rýchlejšie riešenie

V predošlom riešení sme každého nového holuba strkali do prvého vhodného holubníka. To okrem iného zaručuje, že počty holubov v jednotlivých holubníkoch tvorili v každom momente nerastúcu postupnosť. Teda nikdy sa nemohlo stať, že by v i-tom holubníku bolo menej holubov ako v $i+1$-vom.

V tomto riešení budeme holuby strkať do holubníkov rovnakým spôsobom, ale budeme si o holubníkoch pamätať inú informáciu. Konkrétne, pre každé číslo x od 0 po $n-1$ si budeme pamätať číslo prvého holubníka obsahujúceho x holubov.

Trochu problém je s takými číslami x, pre ktoré neexistuje holubník s presne x holubmi. Tento problém sa dá ošetriť rôznymi spôsobmi. Implementačne asi najjednoduchší z nich je pamätať si pre každé x číslo prvého holubníka, ktorý obsahuje x **alebo menej** holubov. Toto číslo má rozumnú hodnotu pre každé x z rozsahu 0 až $n-1$ a ako sa ukáže, veľmi ľahko sa aktualizuje.

Na začiatku si teda budeme pamätať n jednotiek.

Vždy, keď priletí holub s nejakou extroverciou e, pozrieme sa na prvý holubník obsahujúci e alebo menej holubov. Aspoň jeden holubník určite obsahuje presne e holubov (inak by sa holuby nedali dobre ubytovať). Počty holubov v holubníkoch tvoria nerastúcu postupnosť, teda prvý holubník s e alebo menej holubmi určite obsahuje presne e holubov. To znamená, že tam môžeme strčiť nového holuba. Následne si musíme aktualizovať naše informácie: číslo prvého holubníka obsahujúceho e alebo menej holubov sa nám zvýšilo o 1. Žiadne iné z čísel, ktoré si pamätáme, sa nezmenilo.

Takto vieme každého holuba spracovať v konštantnom čase. Časová zložitosť tohto riešenia je teda $O(n)$, pamäťová je tiež $O(n)$.

Listing programu (C++)

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

int main()
{
  int n;
  scanf("%d", &n);
  vector<int> ma_najviac_x(n, 1);
  for(int i=0; i<n; i++)
  {
    int e;
    scanf("%d", &e);
    printf("%d", ma_najviac_x[e]);
    ma_najviac_x[e]++;
    if(i+1 < n) printf(" ");
    else printf("\n");
  }
  return 0;
}
```

Šetrenie pamäťou

Aj v tomto riešení sa dá ušetriť nejaká pamäť. V podstate iba inak ošetríme čísla x, pre ktoré neexistuje holubník s x holubmi. Doteraz sme si pre takéto čísla pamätali prvý holubník s menej ako x holubmi. Teraz si pre ne nebudeme pamätať vôbec nič. Počas ubytovávania holubov sa nám totiž nikdy nestane, že by sme potrebovali zistiť číslo prvého holubníka s x alebo menej holubmi, ak neexistuje holubník s presne x holubmi.

To však prináša niekoľko technických problémov. Prvým problémom je, že už nám nestačí obyčajné pole indexované podľa počtu holubov v holubníku, keďže takéto pole by bolo príliš veľké. Stále však chceme byť schopní pre dané x rýchlo nájsť prvý holubník s x holubmi. Preto použijeme *mapu* založenú na vyvažovanom vyhľadávacom strome. Vo väčšine rozumných programovacích jazykov sú takéto stromy už implementované, v C++ je to `std::map`, v Jave `TreeMap`[10]. V takejto mape vieme k hodnote pre dané x pristupovať v čase $O(\log m)$, kde m je počet prvkov v mape. V rovnakom čase vieme z mapy aj mazať a vkladať do nej nové prvky.

Druhý problém je aktualizácia našej informácie. Keď nám priletí holub s extroverciou e a ubytujeme ho, zanikne nám jeden holubník s e holubmi a vznikne nám holubník s $e+1$ holubmi. Musíme teda skontrolovať, či sme predtým mali nejaký holubník s $e+1$ holubmi. Ak nie, treba do našej mapy pridať informáciu o tom, že takýto holubník máme (aj s jeho číslom).

Ďalej potrebujeme overiť, či ešte stále máme nejaké holubníky s e holubmi. Ak áno, potom musíme aktualizovať číslo prvého z nich (teda zvýšiť ho o 1 oproti doterajšej hodnote). Ak nie, potom treba z mapy vyhodiť záznam o holubníkoch s e holubmi. Aby sme zistili, či máme holubník s e holubmi, stačí nám pozrieť sa na číslo prvého holubníka s menej ako e holubmi. Ak je to holubník hneď za holubníkom, do ktorého sme práve nasťahovali $e+1$-vého holuba, znamená to, že žiadne holubníky s e holubmi už nemáme. Ak je prvý holubník s menej ako e holubmi až niekde ďalej, potom ešte nejaké holubníky s e holubmi máme.

Polohu prvého holubníka s menej ako e holubmi vieme za pomoci našej mapy nájsť v čase $O(\log m)$, kde m je počet záznamov v mape.

Listing programu (C++)

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

int main()
{
  int n;
  scanf("%d", &n);
  map<int, int> prvy_presne_x;
  prvy_presne_x[0] = 1;
  for(int i = 0; i<n; i++)
  {
    int e;
    scanf("%d", &e);
    printf("%d", prvy_presne_x[e]);
    if(prvy_presne_x.count(e+1) == 0)
    {
      prvy_presne_x[e+1] = prvy_presne_x[e];
    }
    prvy_presne_x[e]++;
    if(e > 0)
    {
      map<int, int>::iterator it = prvy_presne_x.find(e);
      it--;
      if(it->second == prvy_presne_x[e])
      {
        prvy_presne_x.erase(e);
      }
    }
    if(i+1 < n) printf(" ");
    else printf("\n");
  }
  return 0;
}
```

S odhadom pamäťovej zložitosti to bude tentoraz veselé. Pre každé číslo x také, že v aspoň jednom holubníku je presne x holubov, si pamätáme jeden záznam v mape. Takýchto čísel x je však v každom momente najviac rádovo $\sqrt{n}$. Ak totiž máme k rôznych nezáporných celých čísel, ich súčet je minimálne $0+1+\cdots+(k-1) = \frac{k(k-1)}{2}$, čo je zhruba polovica z k^2. To znamená, že môžeme mať najviac zhruba $\sqrt{2n}$ rôznych x, pre ktoré existuje holubník s x holubmi, inak by sme dokopy v našich holubníkoch museli mať viac než n holubov. Okrem mapy máme už iba konštantne veľa premenných, teda pamäťová zložitosť nášho algoritmu je $O(\sqrt{n})$.

Každého holuba spracovávame v čase $O(\log m)$, pričom v mape nikdy nebude viac ako rádovo $\sqrt{n}$ čísel, teda môžeme povedať, že spracovanie jedného holuba nám bude trvať čas $O(\log \sqrt{n}) = O(\log n)$. To znamená, že celý algoritmus pobeží v čase $O(n \log n)$.

[10] Možno sa pýtate, prečo nepoužijeme hešmapu, keď je rýchlejšia. Dôvod je taký, že pri aktualizovaní informácie budeme chcieť cez našu mapu vedieť iterovať podľa veľkosti x.

Buj

7. Olympiáda vo vyhľadávaní v texte

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

V texte budeme označovať veľkosť vstupu ako n. Teda dĺžka textu aj dĺžka slova sú $O(n)$. Zjednoduší nám to analýzu časovej zložitosti – neberieme dĺžku textu a dĺžku slova ako dve rôzne premenné.

Naivné riešenie

Samotné zadanie nám dáva návod, ako získať nejaké body za túto úlohu. Implementujeme naivný algoritmus, a počítame si počet porovnaní.

Listing programu (C++)

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

int main () {
  string text, slovo;
  cin >> text >> slovo;

  long long vysledok = 0;
  for (int i = 0; i < (int) text.size(); i++) {
    for (int j = 0; j < (int) slovo.size(); j++) {
      if (i + j >= (int) text.size()) {
        break;
      }
      vysledok++;
      if (text[i + j] != slovo[j]) {
        break;
      }
    }
  }
  cout << vysledok << "\n";
  return 0;
}
```

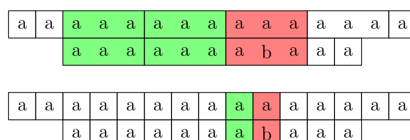
Ďalej si ukážeme tri riešenia. Prvé dve na seba nadväzujú, a síce nie sú z časového hľadiska optimálne, ale majú zaujímavú myšlienku, ktorá sa často dá uplatniť aj v iných úlohách. Kľudne ale môžete začať čítať od tretieho (optimálneho) riešenia.

Šikovnejšie porovnávanie

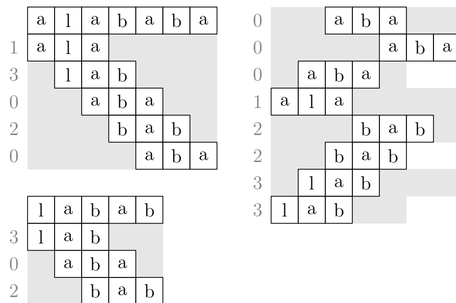
Predstavme si, že by sme vedeli rýchlo porovnávať nie len znak textu so znakom slova, ale aj súvislý *blok* textu dĺžky a so súvislým blokom slova rovnakej dĺžky. Potom by sme vedeli pre každú začiatočnú pozíciu rýchlejšie zistiť, koľko porovnaní od nej spravíme:

1. Porovnávame bloky, kým sa zhodujú. Každé blokové porovnanie zodpovedá a jednotkovým porovnaniam, teda vždy, keď sa bloky zhodujú, zvýšime výsledok o a. Týchto blokových porovnaní nemôže byť viac ako $O(\frac{n}{a})$.

2. Keď narazíme na nezhodu blokov (alebo keď do konca textu alebo slova ostáva menej ako a znakov), začneme porovnávať znak po znaku. Týchto porovnaní spravíme najviac a, lebo vieme, že sa blok textu s blokom slova na niektorom znaku nezhoduje.



Ako ale možno bloky porovnávať rýchlejšie, ako znak po znaku? Kľúčovým pozorovaním je, že týchto blokov je iba $O(n)$. Môžeme si preto dovoliť ich na začiatku lexikograficky usporiadať, a každému bloku priradiť počet lexikograficky menších blokov – jeho *číslo*. Namiesto porovnávania obsahov blokov nám teraz stačí porovnávať ich čísla, ktoré sú veľké $O(n)$ – takže môžeme predpokladať, že sa nám zmestia do bežnej premennej a vieme ich porovnávať v konštantnom čase. To je pre dostatočne veľké bloky oveľa rýchlejšie, ako porovnávať ich znak po znaku.



Spravíme tak $O(\frac{n}{a}+a)$ operácii pre každú začiatočnú pozíciu – pre všetky pozície to potom trvá $O(n\cdot(\frac{n}{a}+a))$.

Ešte musíme započítať čas strávený zisťovaním čísel blokov. Máme ich $O(n)$, a porovnanie dvoch blokov trvá $O(a)$ – takže triedenie nám bude trvať $O(n \log n \cdot a)$.

Vhodná voľba a je $a = \sqrt{n}$, pre ktorú dostávame celkovú časovú zložitosť $O(n\sqrt{n}\log n)$. Pamäťová zložitosť je $O(n\sqrt{n})$ – najviac si toho pamätáme v zozname blokov, ktorých je $O(n)$, a každý má dĺžku $O(\sqrt{n})$.

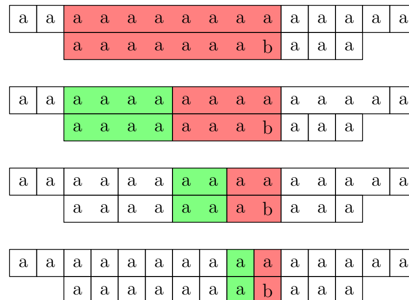
Listing programu (C++)

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

struct Blok {
  string obsah;
  int kto, zaciatok;

  Blok (string a, int b, int c) : obsah(a), kto(b), zaciatok(c) {}

  bool operator< (const Blok b) const { // pouziva sa pri sorte
    return obsah < b.obsah;
  }
};

int main () {
  string retazec[2]; // retazec[0] je text, retazec[1] je slovo
  cin >> retazec[0] >> retazec[1];
  int dlzkaBloku = ceil(sqrt(min(retazec[0].size(), retazec[1].size())));

  // ocislujeme bloky
  vector<Blok> zoznam;
  vector<int> cisloBloku[2]; // cisloBloku[0] su cisla blokov v texte, cisloBloku[1] zas v slove
  for (int kto = 0; kto < 2; kto++) {
    int dlzka = (int) retazec[kto].size() - dlzkaBloku + 1;
    cisloBloku[kto].resize(dlzka);
    for (int i = 0; i < dlzka; i++) {
      zoznam.push_back(Blok(retazec[kto].substr(i, dlzkaBloku), kto, i));
    }
  }
  sort(zoznam.begin(), zoznam.end());
  int pocetMensich = 0;
  for (int i = 0; i < (int) zoznam.size(); i++) {
    if (i > 0) {
      if (zoznam[i - 1] < zoznam[i]) {
        pocetMensich++;
      }
    }
    cisloBloku[zoznam[i].kto][zoznam[i].zaciatok] = pocetMensich;
  }

  // samotne porovnavanie
  long long vysl = 0;
  for (int i = 0; i < (int) retazec[0].size(); i++) {
    int j = 0;
    while (i + j < (int) cisloBloku[0].size() && j < (int) cisloBloku[1].size()) {
      if (cisloBloku[0][i + j] == cisloBloku[1][j]) {
        j += dlzkaBloku;
        vysl += dlzkaBloku;
      }
      else {
        break;
      }
    }
    while (i + j < (int) retazec[0].size() && j < (int) retazec[1].size()) {
      vysl++;
      if (retazec[0][i + j] != retazec[1][j]) {
        break;
      }
      j++;
    }
  }
  cout << vysl << ”\n”;
  return 0;
}
```

Na štýl binárneho vyhľadávania

Nemusíme mať ale iba jednu veľkosť blokov. Bloky si môžeme rozdeliť do viacerých úrovní podľa ich dĺžky, a na každej úrovni ich očíslovať lexikograficky. Ďalej nám opäť stačí namiesto porovnávania obsahov blokov porovnávať ich čísla. Porovnávanie začneme na najvyššej úrovni. Vždy, keď nájdeme nezhodu, začneme porovnávať bloky o úroveň nižšie. Na najnižšej úrovni budú bloky dĺžky 1, teda samotné znaky.

Potrebujeme teda vhodne zvoliť dĺžky blokov na jednotlivých úrovniach, a šikovne bloky na každej úrovni očíslovať lexikograficky. Vhodnou voľbou, ktorá vyrieši oba problémy, je mať na každej úrovni bloky práve dvakrát také dlhé, ako bloky na o 1 nižšej úrovni.



Pri porovnávaní budeme mať na každej úrovni najviac 1 zhodu – 2 zhody by totiž zodpovedali 1 zhode na úrovni vyššie. Na aktuálnu úroveň sme sa ale mohli dostať iba tak, že na vyššej úrovni nastala nezhoda. Úrovní máme $O(\log n)$, takže by sme mali iba $O(\log n)$ blokových porovnaní pre každú začiatočnú pozíciu. Každé blokové porovnanie trvá $O(1)$ (porovnávame čísla veľké $O(n)$), táto časť algoritmu teda trvá $O(n \log n)$.

A ako nájdeme lexikografické číslovanie pre každú úroveň? Tentokrát si nemôžeme dovoliť bloky usporiadať podľa ich obsahu – môžu byť totiž dlhé až $O(n)$, a normálne triedenie by trvalo až $O(n^2 \log n)$. Kľúčovým pozorovaním je, že blok na každej úrovni (okrem najnižšej úrovne) zodpovedá dvom blokom na úrovni o 1 nižšej. Ak tie už máme očíslované, vieme ľahko porovnávať ľubovoľné dva bloky na aktuálnej úrovni – najprv ich porovnáme podľa prvej polovice, a ak sa na nej zhodujú, tak ich porovnáme podľa druhej polovice. Keď vieme bloky porovnávať, tak ich vieme usporiadať a následne očíslovať.

Úrovní je $O(\log n)$, na každej triedime $O(n)$ blokov, a každé porovnanie trvá $O(1)$ (lebo dvakrát porovnávame čísla veľké $O(n)$). Preto má táto časť algoritmu časovú zložitosť $O(n \log^2 n)$.

Celková časová zložitosť je teda $O(n \log^2 n)$. Pamäťová zložitosť je $O(n \log n)$ – najviac pamäte zaberajú úrovne, ktorých je $O(\log n)$, a na každej si pamätáme čísla blokov, ktorých je $O(n)$.

Listing programu (C++)

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

struct Blok {
  int prvaPolka, druhaPolka;
  int kto, zaciatok;

  Blok (int a, int b, int c, int d) : prvaPolka(a), druhaPolka(b), kto(c), zaciatok(d) {}

  bool operator< (const Blok b) const { // pouziva sa pri sorte
    if (prvaPolka == b.prvaPolka) {
      return druhaPolka < b.druhaPolka;
    }
    return prvaPolka < b.prvaPolka;
  }
};

int main () {
  string retazec[2]; // retazec[0] je text, retazec[1] je slovo
  cin >> retazec[0] >> retazec[1];

  vector<vector<int> > cisloBloku[2]; // cisloBloku[0] su cisla blokov v texte, cisloBloku[1] zas v slove

  // ocislujeme jednotkove bloky
  for (int kto = 0; kto < 2; kto++) {
    int dlzka = retazec[kto].size();
    cisloBloku[kto].resize(dlzka);
    for (int i = 0; i < dlzka; i++) {
      int cislo = 'a' - retazec[kto][i] - 1; // jednotkove bloky maju cisla -1 az -26
      cisloBloku[kto][i].push_back(cislo);
    }
  }
  // ocislujeme postupne bloky velkosti 2, 4, 8, ...
  int level = 0;
  int krok = 2;
```

```
  while (krok <= min(retazec[0].size(), retazec[1].size())) {
    vector<Blok> zoznam;
    for (int kto = 0; kto < 2; kto++) {
      for (int i = 0; i + krok <= (int) retazec[kto].size(); i++) {
        int prvaPolka = cisloBloku[kto][i][level];
        int druhaPolka = cisloBloku[kto][i + krok / 2][level];
        zoznam.push_back(Blok(prvaPolka, druhaPolka, kto, i));
      }
    }
    sort(zoznam.begin(), zoznam.end());
    int pocetMensich = 0;
    for (int i = 0; i < (int) zoznam.size(); i++) {
      if (i > 0) {
        if (zoznam[i - 1] < zoznam[i]) {
          pocetMensich++;
        }
      }
      cisloBloku[zoznam[i].kto][zoznam[i].zaciatok].push_back(pocetMensich);
    }
    level++;
    krok *= 2;
  }

  // samotne porovnavanie
  long long vysl = 0;
  for (int i = 0; i < (int) retazec[0].size(); i++) {
    int j = 0;
    int level = (int) cisloBloku[1][0].size() - 1;
    int krok = (1 << level);
    while (level != -1) {
      if (i + j >= (int) retazec[0].size()) {
        break;
      }
      if (j >= (int) retazec[1].size()) {
        break;
      }
      if (level < (int) cisloBloku[0][i + j].size()) {
        if (level < (int) cisloBloku[1][j].size()) {
          if (cisloBloku[0][i + j][level] == cisloBloku[1][j][level]) {
            j += krok;
            vysl += krok;
          }
        }
      }
      level--;
      krok /= 2;
    }
    if (j < (int) retazec[1].size() && i + j < (int) retazec[0].size()) {
      vysl++;
    }
  }
  cout << vysl << "\n";
  return 0;
}
```

Toto riešenie sa dá vylepšiť tak, aby malo optimálnu časovú zložitosť $O(n)$ za využitia kladív ako *sufixové pole*, *LCP pole*, a *range-minimum query*. Tie výrazne presahujú rámec tejto úlohy, a uvádzať ich tu nebudeme.

Čo navrhujú Knuth, Morris a Pratt

Predstavme si, že sa nám znaky textu odkrývajú postupne. Na začiatku máme teda prázdny text, a postupne na jeho koniec pridávame znaky. Ako sa bude meniť počet porovnaní, ktoré spraví naivný algoritmus?

Pozrime sa na nejakú začiatočnú pozíciu. Od nej spravíme niekoľko porovnaní, až kým nenastane jedno z nasledujúcich:

1. Našli sme nezhodu. Potom keď na koniec textu pridáme nový znak, tak počet porovnaní, ktoré spravíme pre túto začiatočnú pozíciu, sa nezmení.
2. Nenašli sme nezhodu, a došli sme na koniec slova. Pridaním znaku na koniec textu sa počet porovnaní, ktoré spravíme, opäť nezmení – nie je s čím ďalej porovnávať, lebo sme na konci slova.
3. Nenašli sme ešte nezhodu, došli sme na koniec textu ale nie sme ešte na konci slova. Keď na koniec textu pridáme nový znak, tak určite spravíme jedno ďalšie porovnanie.

Zaujíma nás teda počet tých začiatkov, pri ktorých sme došli na koniec textu ale ešte nie sme na konci slova. Každému takému začiatku zodpovedá reťazec znakov, ktorý sme už spracovali, a ktorý sa zhodoval s prvými niekoľkými znakmi slova. Je to teda vlastný prefix slova. Takže keď pridávame znak na koniec textu, výsledok sa zvýši o počet vlastných prefixov slova, ktoré sú sufixom textu (pred pridaním nového znaku).

Toto nám umožňuje spočítať práve Knuthov-Morrisov-Prattov algoritmus[11]. Konkrétne

1. Vieme s jeho pomocou po pridaní každého znaku zistiť najdlhší prefix slova, ktorý je sufixom textu.

[11] https://www.ksp.sk/kucharka/kmp/

2. Najdlhší prefix slova, ktorý je sufixom textu, nám jednoznačne určuje všetky prefixy slova, ktoré sú sufixami textu. To vyplýva z toho, že tie sú kratšie – takže ak sú sufixami textu, musia byť aj sufixami nášho najdlhšieho prefixu.

3. Pre každý prefix slova vieme jeho najdlhší vlastný sufix taký, že je zároveň prefixom slova – takzvané *spätné linky*.

4. Ak postupne budeme uvažovať prefix, na ktorý ukazuje naša spätná linka, potom prefix, na ktorý ukazuje spätná linka predchádzajúceho prefixu, … tak vytvoríme postupnosť **všetkých** prefixov slova takých, že sú sufixom nášho najdlhšieho prefixu.

5. Nás zaujíma ich počet. Ak ho pre `i`-ty prefix slova označíme `pocet[i]`, a spätnú linku z `i`-teho prefixu ako `spat[i]`, tak `pocet[i] = pocet[spat[i]] + 1`. Takto vypočítame všetky počty.

6. Ak je najdlhším prefixom celé slovo, tak odrátame od výsledku 1, nakoľko sme ho zarátali v predchádzajúcom procese, a nemali sme ho zarátať.

Časová aj pamäťová zložitosť sú $O(n)$.

Listing programu (C++)

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

int precitaj (char znak, string& slovo, vector<int>& spatneLinky, int stav) {
  if (stav + 1 < (int) spatneLinky.size()) {
    if (znak == slovo[stav]) {
      return stav + 1;
    }
  }
  while (stav != 0) {
    stav = spatneLinky[stav];
    if (znak == slovo[stav]) {
      return stav + 1;
    }
  }
  return 0;
}

int main () {
  string text, slovo;
  cin >> text >> slovo;

  vector<int> spatneLinky(1, 0);
  vector<int> dlzkaChvostu(1, 0);
  for (int i = 0; i < (int) slovo.size(); i++) {
    int predch = precitaj(slovo[i], slovo, spatneLinky, i);
    spatneLinky.push_back(predch);
    dlzkaChvostu.push_back(1 + dlzkaChvostu[predch]);
  }

  long long vysl = 0;
  int stav = 0;
  for (int i = 0; i < (int) text.size(); i++) {
    if (stav != (int) slovo.size()) {
      vysl++;
    }
    vysl += dlzkaChvostu[stav];
    stav = precitaj(text[i], slovo, spatneLinky, stav);
  }
  cout << vysl << "\n";
  return 0;
}
```

misof

8. Obrana pobrežia

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

Už v úlohe 8 minulej série sme sa stretli s elementárnou operáciou, ktorá nám bude užitočná aj tu: dozvedeli sme sa, ako pomocou vektorového súčinu ľahko a presne zistiť, či sa dve úsečky pretínajú. Toto budeme potrebovať aj v tejto úlohe. Kto to už zabudol, nájde to v Kuchárke[12].

Hrubá sila

Akonáhle máme takúto funkciu, už vieme riešiť zadanú úlohu hrubou silou: postupne vyskúšame všetkých $n!$ permutácií, každú z nich skontrolujeme, a spočítame tie, pre ktoré sa žiadne dve úsečky nepretínajú. Za takéto

[12] https://ksp.sk/kucharka/skalarny_a_vektorovy_sucin

riešenie síce nebude plný počet bodov, ale ak ste za túto úlohu mali nulu, tak aj toto je od nuly lepšie. A je to naozaj pár riadkov programu – napr. v C++ máme k dispozícii funkciu `next_permutation` pomocou ktorej sa skúšanie permutácií implementuje ozaj hravo.

Chceme lepšie

My ale chceme nejaké riešenie, ktoré bude bežať v polynomiálnom čase. Ako na to?

Naše riešenie je založené na nasledujúcom pozorovaní: Zoberme sépiu, ktorá je najďalej od pobrežia. (Ak je takých viac, tak trebárs tú z nich, ktorá má najmenšie číslo.) Položme si teraz otázku: môže byť táto sépia trafená balistou číslo k? Odpoveď je jednoduchá: áno, ak sú splnené nasledujúce podmienky.

- Na úsečke medzi balistou k a našou sépiou neležia žiadne iné sépie.
- "Naľavo" od našej úsečky je rovnako veľa balíst ako sépií (a napravo teda nutne tiež).

Postupne vyskúšame všetky možnosti pre k. Vždy, keď nájdeme nejakú vyhovujúcu, tak sme vlastne pôvodný problém rozdelili na dva menšie: Nech a je počet spôsobov, ktorými vieme spárovať sépie a balisty naľavo od našej úsečky a nech b je počet spôsobov, ktorými to vieme urobiť napravo. Potom dokopy dostávame ab riešení, lebo každé riešenie vľavo vieme skombinovať s každým riešením vpravo.

No a ak sme už zvlášť pre každé k určili, koľko mu zodpovedá riešení, tak celkový počet riešení je jednoducho súčtom všetkých týchto počtov.

Vyššie popísané riešenie nás teda vedie k rekurzívnemu riešeniu našej úlohy: vyskúšame nejaké možnosti ako spraviť jednu dvojicu balista-sépia a následne sa rekurzívne zavoláme na nejaké menšie vstupy.

Ostáva ale celkom veľa nezodpovedaných otázok. Funguje to vôbec? A ak áno, má to polynomiálnu časovú zložitosť?

Funguje to vôbec?

To, že vyššie uvedené riešenie funguje, je spôsobené práve našou šikovnou voľbou sépie, ktorej pár hľadáme ako prvý. My sme si zvolili sépiu S, ktorá je najďalej od pobrežia, a spojili sme ju s nejakou balistou B na pobreží. Ak teraz zoberieme hocijakú sépiu naľavo od S a spojíme ju s balistou napravo od B (alebo naopak), bude táto nová úsečka určite križovať úsečku SB. Vďaka našej šikovnej voľbe sépie sa nám teda naozaj nakreslením úsečky SB zadanie rozpadne na dve časti ktoré sú nezávislé.

Má to polynomiálnu časovú zložitosť?

Nemá. Generuje to postupne po jednom všetky prípustné riešenia, a tých ani omylom nemusí byť len polynomiálne veľa. (Ľahko sa vyrobí napríklad vstup, ktorý bude mať riešení aspoň $2^{n/2}$. Viete dosiahnuť ešte viac?)

Ako teda dostať polynomiálne riešenie?

Jednoducho: pridáme memoizáciu. Každý konkrétny podproblém, na ktorý narazíme ("tu je podmnožina sépií a úsek balíst, koľkými spôsobmi ich vieme pospájať?") budeme riešiť len raz. Keď ho prvýkrát vyriešime, zapamätáme si jeho riešenie. Ak by sme na ten istý podproblém narazili inokedy, už nebudeme nič skúšať, len sa pozrieme na jeho zapamätané riešenie a to vrátime na výstup.

Prečo pridaním memoizácie dostaneme polynomiálne riešenie?

Kvôli konzistencii si predstavme, že naľavo aj napravo od celého vstupu je po jednej sépii (ďaleko na mori) a jednej baliste. Ľubovoľný konkrétny podproblém si teraz môžeme popísať tak, že povieme dve úsečky sépia-balista: jedna tvorí jeho ľavý a druhá jeho pravý okraj.

Všetkých podproblémov je teda nanajvýš toľko, koľko je takých dvojíc úsečiek – teda $O(n^4)$, lebo máme n možností pre každý koniec každej úsečky. V skutočnosti bude dosiahnuteľných podproblémov výrazne menej. Na dôkaz polynomiálnej časovej zložitosti nám ale stačí takýto horný odhad ich počtu.

No a konkrétny podproblém vyriešime v $O(n^2)$: najskôr overíme, či máme rovnako veľa sépií a balíst (ak nie, odpoveď je 0), potom nájdeme najvzdialenejšiu sépiu, a potom ju postupne vyskúšame spojiť s každou balistou. Keď máme konkrétnu dvojicu sépia-balista, potrebujeme ešte prejsť všetky ostatné a rozdeliť ich na ľavé a pravé (a skontrolovať, že žiadna sépia neleží na práve nakreslenej úsečke).

Celkovo teda dostávame riešenie s časovou zložitosťou $O(n^6)$.

Šlo by aj zlepšiť. Napríklad tak, že keď už máme zvolenú najvzdialenejšiu sépiu, tak si všetky ostatné balisty a sépie okolo nej polárne usporiadame, aby sme vedeli rýchlejšie hovoriť, ktoré objekty sú naľavo a ktoré napravo

od práve skúšanej úsečky sépia-balista. Zadanie však hovorí, že čokoľvek polynomiálne je dobré, a tak budeme leniví a nebudeme už nič zlepšovať.

Implementácia

V mojej implementácii som bol extra lenivý, preto si konkrétny podproblém reprezentujem nie pomocou štyroch čísel (dve balisty a dve sépie) ale pomocou dvoch 64-bitových intov: v jednom sú nastavené bity pre tie harpúny, ktoré ešte mám, a v druhom bity pre sépie. Ono je to skoro jedno, stavov mám stále rovnako a takto sa ten kód stručnejšie písal.

Namiesto 4-rozmerného poľa potom používam na memoizáciu `map`. Tým mi pribudne v časovej zložitosti ešte logaritmus navyše. Spomínal som už, že som bol extra lenivý? :)

Navyše sa ani neobťažujem priebežne kontrolovať, či mám rovnako veľa sépií a balíst. Až ak na konci vetvy rekurzie zistím, že jedny sa už minuli a druhé ešte nie, vrátim nulu.

Listing programu (C++)

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

int N;
vector<ll> B, X, Y;

map< pair<ll,ll>, int > memo;

// vektorový súčin: kladný ak (x3,y3) leží naľavo ak sa pozeráme z (x1,y1) na (x2,y2)
ll cross_product(ll x1, ll y1, ll x2, ll y2, ll x3, ll y3) {
    ll ux = x2-x1, uy = y2-y1, vx = x3-x1, vy = y3-y1;
    return ux*vy - vx*uy;
}

int vyries(ll balisty, ll sepie) {
    pair<ll,ll> key = {balisty,sepie};
    if (memo.count(key)) return memo[key]; // tento problém sme už riešili

    int &odpoved = memo[key]; // spravíme si referenciu na záznam v mape kam chceme zapísať odpoveď

    // zistíme, či už sme na konci
    if (balisty == 0 && sepie == 0) return odpoved = 1;
    if (balisty == 0 || sepie == 0) return odpoved = 0;

    // tu by sa dalo rovno kontrolovať, či máme balíst a sépií rovnako veľa
    // pridanie nasledujúceho riadku výrazne urýchli čas behu v praxi
    // if (__builtin_popcountll(balisty) != __builtin_popcountll(sepie)) return odpoved = 0;

    // nájdeme najvzdialenejšiu sépiu
    int bests = -1;
    for (int s=0; s<N; ++s) if (sepie & 1LL << s) if (bests == -1 || Y[s] > Y[bests]) bests = s;

    // vyskúšame všetky možné balisty
    odpoved = 0;

    for (int b=0; b<N; ++b) if (balisty & 1LL << b) {
        // rozdelíme zvyšok na ľavú a pravú stranu, skontrolujeme že nič neleží na úsečke
        bool ok = true;
        ll slave = 0, sprave = 0, blave = 0, bprave = 0;

        for (int i=0; i<N; ++i) if (i != bests) if (sepie & 1LL << i) {
            ll poloha = cross_product( B[b], 0, X[bests], Y[bests], X[i], Y[i] );
            if (poloha > 0) slave |= 1LL << i;
            if (poloha < 0) sprave |= 1LL << i;
            if (poloha == 0) ok = false;
        }

        for (int i=0; i<N; ++i) if (i != b) if (balisty & 1LL << i) {
            ll poloha = cross_product( B[b], 0, X[bests], Y[bests], B[i], 0 );
            if (poloha > 0) blave |= 1LL << i;
            if (poloha < 0) bprave |= 1LL << i;
        }

        if (ok) odpoved = (odpoved + vyries(blave,slave) * vyries(bprave,sprave)) % 1000000007;
    }
    return odpoved;
}

int main() {
    cin >> N;
    B.resize(N); for (auto &b:B) cin >> b;
    X.resize(N); for (auto &x:X) cin >> x;
    Y.resize(N); for (auto &y:Y) cin >> y;
    ll vsetko = (1LL << N) - 1;
    cout << vyries(vsetko,vsetko) << endl;
}
```