Riešenia kategórie B

B-II-1 Hľadanie permutácií

Zopakujme si, čo nám hovorilo zadanie. Máme postupnosť čísel dĺžky n, v ktorej chceme hľadať za sebou idúcu permutáciu zadaných m čísel. Našou úlohou je spočítať, koľko takýchto permutácií sa v zadanej postupnosti nachádza.

Keďže hľadané slová musia ísť za sebou a žiadne nemôžeme vynechať (môžeme meniť iba ich poradie), je jasné, že sa nám stačí pozerať na každú za sebou idúcu m-ticu slov v zadanej postupnosti. Teda najskôr sa pozrieme na prvé až m-té slovo stránky, potom na druhé až $(m + 1)$., a tak ďalej. Ak pre každú m-ticu slov stránky dokážeme povedať, či je permutáciou hľadaných slov, tak ľahko spočítame výsledok.

Naviac takýchto za sebou idúcich m-tíc je v postupnosti dĺžky n pomerne málo. Posledná, ktorú ešte budeme skúšať, začína na pozícii $n - m + 1$. Neskoršie začiatky nemá zmysel skúšať, lebo už by sme mali menej ako m prvkov. Skontrolovať teda treba každú z $n - m + 1$ možných m-tíc.

Teraz potrebujeme vymyslieť spôsob, ktorým pre každú za sebou idúcu m-ticu overíme, či je táto m-tica permutáciou hľadaných slov.

Samozrejme, najľahšie riešenie je vyskúšať každú možnú permutáciu skúšanej m-tice a overiť, či sa nerovná postupnosti slov, ktoré sme dostali zadané. Uvedomme si však, že ak by užívateľ zadal tých istých m slov v inom poradí, náš program musí dať rovnaký výsledok, lebo pri permutáciách nám na poradí nezáleží. To ale znamená, že náš program musí fungovať aj keď sú slová, ktoré hľadáme, zadané od najmenšieho po najväčšie. A aj keby neboli, vieme si ich predsa ľahko usporiadať.

Vďaka tomuto pozorovaniu už nemusíme skúšať všetky permutácie. Vieme totiž, že hľadáme tú, ktorá je usporiadaná. Náš algoritmus teda najskôr usporiada m-ticu slov, ktoré dostal od používateľa. Keď potom overuje, či je nejaká m-tica čísel z postupnosti permutáciou hľadaných slov, stačí mu tieto slová taktiež usporiadať a porovnať, či sú tieto dve usporiadané postupnosti rovnaké. Toto vedie k riešeniu s časovou zložitosťou $O(n \cdot m \log m)$. Za takéto riešenie ste mohli získať 5 bodov.

Toto riešenie však vieme ešte trochu vylepšiť. To čo sme ešte nevyužili je fakt, že čísla na vstupe sú z rozsahu 1 až k, pričom hodnota k vôbec nie je veľká, najviac má veľkosť 1 000 000.

Existuje totiž aj jednoduchší spôsob ako zistiť, či je jedna m-tica čísel permutáciou inej. Musí predsa platiť, že v oboch m-ticiach je rovnako veľa výskytov čísla 1, rovnako veľa výskytov čísla 2, a tak ďalej až po k. Tento fakt vieme veľmi ľahko využiť. Najskôr si zoberieme zadaných m slov a do poľa veľkosti k si pre každé číslo spočítame, koľkokrát sa nachádza v tejto množine slov. Toto vieme spraviť jedným prechodom cez tieto čísla v čase $O(m)$ – číslo spracujeme tak, že zväčšíme hodnotu na príslušnej pozícii nášho poľa.

Keď overujeme nejakú m-ticu slov, vieme s nimi spraviť tú istú operáciu, tiež v čase $O(m)$, akurát si výsledok zapíšeme do iného poľa. Následne potrebujeme overiť, či sú dve polia rovnaké. A to spravíme najľahšie tak, že sa postupne pozrieme na každú pozíciu a porovnáme, či sú naše polia na tejto pozícii rovnaké. Ak má pole veľkosť k, zaberie nám to čas $O(k)$.

Toto celé musíme spraviť $n - m + 1$ krát, čo vedie k výslednej zložitosti $O(n \cdot (m + k))$, čo je s prihliadnutím na limity zo zadania o logaritmus lepšie ako predchádzajúce riešenie.

K vzorovému riešeniu nám chýbajú už len dve pozorovania. Musíme sa zamyslieť, čo robíme nadbytočne. Pozrime sa bližšie na to, ako postupuje predchádzajúci algoritmus. Najskôr si do poľa spočíta, koľkokrát sa každé číslo nachádza v hľadaných slovách. Označme si toto pole ako `Hladane_slova[]`. Potom si náš algoritmus zoberie prvú m-ticu čísel z postupnosti, ktorá sa nachádza na pozíciách 1 až m. Pre túto m-ticu si tiež spočíta pole výskytov (pole s názvom `Aktualne_slova[]`) a tieto dve polia porovná.

V ďalšom kroku chce spracovať ďalšiu m-ticu, tú, ktorá je na pozíciách 2 až $m+1$. Opäť pre ňu bude vyrábať pole výskytov. Táto m-tica je však skoro taká istá ako tá predchádajúca. Na rozdiel od nej akurát pribudol prvok z pozície $m + 1$ a odbudol prvok z pozície 1. Ak sú však tieto m-tice skoro rovnaké, tak to znamená, že pole `Aktualne_slova[]` sa zmení iba veľmi málo.

Namiesto toho, aby sme opäť v čase $O(m)$ toto pole vytvárali, tak ho iba upravíme o jedno slovo, ktoré tam už nie je, a o slovo, ktoré pribudlo. To zvládneme v konštantnom čase. Následne môžeme opäť porovnať polia `Hladane_slova[]` a `Aktualne_slova[]`.

Tento postup samozrejme vieme opakovať. Pole `Aktualne_slova[]` vieme opäť jednoducho upraviť, aby zodpovedalo m-tici na pozíciách 3 až $m + 2$ a rovnakým spôsobom môžeme postupovať až do konca. To vedie k riešeniu s časovou zložitosťou $O(n \cdot k)$, za ktoré ste mohli získať až 8 bodov.

Poslednú vec, ktorú musíme vylepšiť, je porovnávanie našich dvoch polí. Tam vieme využiť veľmi podobnú myšlienku. Ak sa naša skúšaná m-tica zakaždým zmení iba o 2 prvky, nemôže sa veľmi zmeniť ani to, ako podobné sú si porovnávané polia. Okrem toho, že si budeme pamätať koľko krát sa nejaké číslo vyskytuje v našej skúšanej m-tici, budeme si pamätať aj to, na koľkých pozíciách sú polia `Hladane_slova[]` a `Aktualne_slova[]` rovnaké.

Ak túto hodnotu dokážeme počítať správne a ešte aj rýchlo, tak vieme aj ľahko porovnať, či sa naše polia rovnajú. Stačí sa pozrieť, či sa táto hodnota rovná k – teda či sú naše dve polia rovnaké na každej pozícii. Označme si túto hodnotu ako `pocet_rovnakych`.

A opäť, táto hodnota sa nemôže zmeniť o viac ako 2 pri každom posunutí. Vždy, keď meníme číslo v poli `Aktualne_slova[]`, porovnáme ho najskôr s príslušnou hodnotou v `Hladane_slova[]` Ak sa na tejto pozícii rovnali pred tým, ako sme spravili zmenu, znamená to, že musíme o 1 zmenšiť hodnotu `pocet_rovnakych`, lebo budeme mať o jednu zhodu menej. Ak sa rovnali potom ako sme urobili zmenu, tak nám naopak hodnota `pocet_rovnakych` o 1 stúpne. V inom prípade sa nezmení.

A to je celé. Spojením týchto princípov vieme každú, okrem prvej m-tice porovnať v čase $O(1)$. Výsledné riešenie má preto časovú zložitosť $O(n + m + k)$.

Listing programu (C++)

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

int main() {
    int k = 11;
    int n,m;
    scanf("%d %d", &n, &m);
    vector<int> Stranka;
    Stranka.resize(n);
    vector<int> Hladane_slova;
    Hladane_slova.resize(k, 0);
    for (int i = 0; i < n; i++)
        scanf(" %d", &Stranka[i]);
    for (int i = 0; i < m; i++) {
        int x;
        scanf(" %d", &x);
        Hladane_slova[x]++;
    }
    int pocet_rovnakych = 0;
    int riesenie = 0;
    vector<int> Aktualne_slova;
    Aktualne_slova.resize(k, 0);
    // Porovnavat budeme vzdy Aktualne_slova a Hladane_slova.
    // Ak sa budu rovnat, tak mame zhodu. V premennej pocet_rovnakych
    // si pamatame kolko indexov tychto poli ma rovnaku hodnotu.
    for (int i = 0; i < k; i++)
        if(Aktualne_slova[i] == Hladane_slova[i])
            pocet_rovnakych++;
    // spracujeme prvych m cisel zo stranky
    for (int i = 0; i < m; i++) {
        int slovo = Stranka[i];
        if (Aktualne_slova[slovo] == Hladane_slova[slovo])
            pocet_rovnakych--;
        Aktualne_slova[slovo]++;
        if (Aktualne_slova[slovo] == Hladane_slova[slovo])
            pocet_rovnakych++;
    }
    if (pocet_rovnakych == k)
        riesenie++;
    // postupne sa pozerame na dalsie m-tice
    for (int i = m; i < n; i++) {
        // pridame slovo Stranka[i]
        int slovo = Stranka[i];
        if (Aktualne_slova[slovo] == Hladane_slova[slovo])
            pocet_rovnakych--;
        Aktualne_slova[slovo]++;
        if (Aktualne_slova[slovo] == Hladane_slova[slovo])
            pocet_rovnakych++;
        // odoberieme slovo Stranka[i-m]
```

```
        slovo = Stranka[i-m];
        if (Aktualne_slova[slovo] == Hladane_slova[slovo])
            pocet_rovnakych--;
        Aktualne_slova[slovo]--;
        if (Aktualne_slova[slovo] == Hladane_slova[slovo])
            pocet_rovnakych++;
        if (pocet_rovnakych == k)
            riesenie++;
    }
    printf("%d\n",riesenie);
}
```

B-II-2 Pokémoni 2

Poďme Peťovi aj po druhýkrát pomôcť chytať pokémonov. Zmena oproti domácemu kolu je hlavne v tom, ako získava pokélopty. Napríklad si môžeme všimnúť, že na začiatku nemá k dispozícii žiadne. Je teda jasné, že kým nepríde k prvému pokéstopu nemá na výber: všetkých pokémonov, ktorých stretne, musí nechať ísť.

Rozdiel je tiež v tom, že v pokéstopoch je iba obmedzený počet pokélôpt, ale Peťo ich môže pri sebe niesť koľko chce. Je teda jasné, že vždy, keď príde k pokéstopu, zoberie si odtiaľ všetky dostupné pokélopty. Nemá dôvod to nespraviť, každá ďalšia pokélopta je ďalší pokémon, ktorého môže potenciálne chytiť.

Ako však zistíme, ktorých pokémonov má chytiť? Skúsme ich na začiatok pochytať ľubovoľným spôsobom a potom sa ho budeme snažiť napraviť. Dobrý spôsob môže byť pažravý – vždy keď máme voľnú pokéloptu a stretneme pokémona tak ho chytíme. Na jeho silu sa ani nepozrieme.

Takáto stratégia zo začiatku funguje vcelku dobre. Postupne navštevujeme pokéstopy, ktoré nás zásobujú pokéloptami a chytáme všetkých pokémonov, ktorých stretneme. Problém nastane, keď prvýkrát narazíme na pokémona, ale k dispozícii už nemáme žiadnu prázdnu pokéloptu. V tomto momente totiž nastane otázka, či by nebolo lepšie chytiť tohto pokémona namiesto nejakého iného, ktorého sme chytili pred tým.

Veľmi dôležité pozorovanie je to, že ak chceme chytiť nejakého pokémona, môžeme to spraviť len pokéloptami z predchádzajúcich pokéstopov. A vieme, že tie sme použili na chytenie všetkých predchádzajúcich pokémonov. My sa však na rozdiel od Peťa vieme vracať do minulosti. Kľudne si môžeme povedať, že niektorého pokémona vypustíme (jednoducho budeme tvrdiť, že sme ho vlastne nikdy nechytili), čím sa nám uvoľní pokélopta, ktorou môžeme chytiť pokémona pred ktorým stojíme.

Jediná otázka je, že kedy to chceme urobiť a ktorého pokémona vypustiť. Odpoveď je však pomerne jasná. Ak sa rozhodneme nejakého pokémona vypustiť z lopty (čiže nechytiť), tak samozrejme, že najviac sa oplatí vybrať toho najslabšieho. Výsledná sila našich pokémonov totiž klesne najmenej. Tiež je jasné, že ak je aktuálny pokémon slabší ako najslabší pokémon, ktorého sme zatiaľ chytili, tak sa nám chytať neoplatí. Ak je však aspoň o 1 silnejší ako náš najslabší pokémon, tak keď ho vypustíme a nového pokémona chytíme namiesto neho, spoločná sila našich pokémonov sa zvýši aspoň o 1.

Myšlienka nášho riešenia teda vyzerá nasledovne. Postupne prechádzame udalosti, ktoré sa daný deň Peťovi udiali. Ak narazíme na pokéstop, zoberieme si z neho všetky pokélopty. Ak narazíme na pokémona a zároveň máme k dispozícii prázdnu pokéloptu, tak ho chytíme. Pritom si pamätáme, akých silných pokémonov sme zatiaľ chytili.

Ak stretneme pokémona a už nemáme pokéloptu, do ktorej by sme ho chytili, porovnáme jeho silu so silou najslabšieho zatiaľ chyteného pokémona. Ak je nový pokémon slabší, necháme ho ísť. Ak je však silnejší, tak nášho najslabšieho pokémona vypustíme (budeme sa tváriť, že sme ho nikdy nechytili), čím sa nám uvoľní pokélopta, ktorou chytíme nového pokémona.

Na konci budeme mať zoznam pokémonov, ktorých sme chytili. Súčet síl týchto pokémonov je hľadaným výsledkom.

Jediná vec, ktorá nie je z predchádzajúceho popisu jasná, je spôsob, akým nájdeme silu najslabšieho pokémona, ktorého sme zatiaľ chytili. Najľahšie riešenie je pamätať si sily chytených pokémonov v poli. Vždy keď hľadáme najslabšieho pokémona, toto pole prejdeme a zistíme, na ktorej pozícii sa nachádza. Ak sa ho rozhodneme nahradiť novým pokémonom, jednoducho prepíšeme hodnotu na tejto pozícii. Uvedomme si, že nikdy nenazbierame viac ako n pokémonov, preto nám táto operácia potrvá najviac čas $O(n)$. Takéto riešenie má potom časovú zložitosť $O(n^2)$ a mohli ste zaň získať 7 bodov.

Halda

Výrazne lepšie riešenie však vieme dosiahnuť ak na hľadanie najmenšej sily použijeme dátovú štruktúru halda (poprípade ešte silnejší nástroj vo forme vyváženého vyhľadávacieho stromu).

Halda je dátová štruktúra, v ktorej si vieme mať uloženú nejakú množinu čísel – nech táto množina obsahuje n čísel. Nad touto množinou potom vieme vykonávať tri operácie:

- `top()` – povedz mi hodnotu najmenšieho čísla v tejto množine v čase $O(1)$
- `push(x)` – do množiny čísel pridaj číslo x v čase $O(\log n)$
- `pop()` – odstráň z množiny najmenšie číslo v čase $O(\log n)$

Všimnite si, že odstraňovať vieme vždy iba najmenšie číslo množiny. Ako vidíme, halda vie robiť presne to čo potrebujeme. Ukladať do nej budeme sily pokémonov, ktorých sme chytili. Ak chceme chytiť nového pokémona, použijeme funkciu `push()`, ktorou jeho silu pridáme do tejto množiny. Následne sa vieme veľmi rýchlo spýtať na najslabšieho pokémona, ktorého sme chytili, pomocou `top()` a ak ho chceme vypustiť, tak použijeme `pop()`. A to všetko v najviac logaritmickom čase. Časová zložitosť takéhoto riešenia je teda $O(n \log n)$.

Väčšina programovacích jazykov má svoju implementáciu haldy, ktorú môžeme priamo použiť. Napríklad v C++ ju nájdeme v knižnici STL pod menom `priority_queue`, v Pythone chceme použiť metódy `heappush` a `heappop` z modulu `heapq`, a tak ďalej.

V nižšie uvedenom programe si môžete pozrieť použitie haldy v C++. STL implementácia haldy hľadá najväčší prvok, nie najmenší, to však vieme ľahko zmeniť.

Listing programu (C++)

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

int main() {
    int n;
    scanf("%d", &n);
    int prazdne_pokelopty = 0;
    int vysledok = 0;
    // Ak by som chcel maximovu haldu, stacilo by napisat
    // priority_queue<int> chyteny_pokemoni;
    priority_queue<int, vector<int>, greater<int> > chyteny_pokemoni;
    for (int i = 0; i < n; i++) {
        int x;
        scanf("%d", &x);
        // narazil som na pokestop
        if (x < 0) {
            prazdne_pokelopty -= x;
            continue;
        }
        // narazil som na pokemona
        if (prazdne_pokelopty > 0) {
            chyteny_pokemoni.push(x);
            prazdne_pokelopty--;
            vysledok += x;
        }
        else if(!chyteny_pokemoni.empty()) {
            // ak je najslabsi chyteny pokemon slabsi
            if (chyteny_pokemoni.top() < x) {
                vysledok = vysledok - chyteny_pokemoni.top() + x;
                chyteny_pokemoni.pop();
                chyteny_pokemoni.push(x);
            }
        }
    }
    printf("%d\n", vysledok);
}
```

B-II-3 Bludisko

Na hľadanie najkratšej cesty v takomto bludisku je vhodné použiť *prehľadávanie do šírky*. Pri prehľadávaní do šírky postupne nájdeme všetky políčka, na ktoré sa vieme dostať na 0 krokov, 1 krok, 2 kroky, a tak

ďalej. Vždy, keď nejaké políčko spracúvame, tak vyskúšame všetky možnosti, ako z neho spraviť krok ďalej. Niektoré tieto možnosti povedú na políčka, kde sme už boli, ale niektoré povedú na nové políčka. Vždy, keď takto „objavíme“ nové políčko, zaradíme ho do fronty, v ktorej čakajú políčka, ktoré ešte treba spracovať. Podrobnejší popis prehľadávania do šírky nájdete napríklad vo vzorových riešeniach úlohy B-I-2 v 28. ročníku Olympiády v informatike.

Pomocou prehľadávania do šírky vieme teda (v čase priamo úmernom veľkosti bludiska) zodpovedať nasledujúce otázky: „Ak začínam na tomto políčku, na ktoré iné políčka sa viem dostať? A ako dlho mi to na ktoré z nich bude trvať?“

V našej súťažnej úlohe nám však takéto niečo ešte nestačí – potrebujeme sa zamyslieť nad tým, ako do algoritmu prehľadávania do šírky doplniť prechod cez stenu. Ukážeme si dve riešenia: jedno použije malý ale šikovný trik, druhé bude viac technické.

Riešenie s malým trikom

Prehľadávanie do šírky spustíme nie raz, ale dvakrát. Prvýkrát ho spustíme normálne, teda z pozície, kde začína Peťka. Tým sa dozvieme, na ktoré políčka bludiska sa vie dostať bez použitia elixíru. Druhé prehľadávanie (úplne nezávislé od prvého) spustíme z políčka, na ktorom je východ. Toto prehľadávanie nám pre každé políčko povie, na koľko krokov sa naň dá (bez použitia elixíru) dostať od východu bludiska. Alebo, ekvivalentne, na koľko krokov sa z ktorého políčka vieme dostať k východu.

No a tieto dve informácie dokopy nám už stačia na to, aby sme vedeli vyriešiť našu pôvodnú úlohu. Ako? Prvou možnosťou je, že Peťka elixír nepoužije. Riešenie pre túto možnosť už máme spočítané v rámci toho, čo nám vrátilo prvé prehľadávanie do šírky. Druhou možnosťou je, že Peťka pôjde cez práve jednu stenu. Postupne vyskúšame všetky možnosti, ktorá stena to bude.

No a vyskúšať konkrétnu stenu vieme ľahko v konštantnom čase. Stačí zistiť dve veci: ako najrýchlejšie sa vieme dostať od Peťky na niektoré políčko susediace s touto stenou a ako rýchlo sa dá od niektorého políčka susediaceho s touto stenou dostať k východu. Súčet týchto dvoch hodnôt, plus dva za kroky cez samotnú stenu, nám povie optimálne riešenie, ak by Peťka išla cez túto konkrétnu stenu. (A samozrejme, ak sa od Peťky alebo od východu k tejto stene vôbec nedá dostať, tak ju len spokojne odignorujeme, riešenie nám neovplyvní.) Najmenší, takto dosiahnutý súčet, je potom výslednou odpoveďou.

Časová zložitosť tohto riešenia je $O(rs)$, teda lineárna od plochy bludiska: najskôr spustíme dve prehľadávania a pri každom z nich nanajvýš raz spracujeme každé políčko bludiska, a potom práve raz spracujeme každú stenu v bludisku.

Listing programu (Python)

```python
from queue import Queue # fronta

nekonecno = 10**9

def prehladaj(mapa, start):
    # prehľadaj do šírky danú mapu z daného vrcholu
    # na výstup vráť pre každý dosiahnuteľný vrchol vzdialenosť doň

    R, S = len(mapa), len(mapa[0])
    sr, ss = start

    # na začiatku sú všetky vzdialenosti nekonečné, len pre štart je to 0
    vzdialenost = [ [ nekonecno for s in range(S) ] for r in range(R) ]
    vzdialenost[sr][ss] = 0

    Q = Queue()
    Q.put( (sr,ss) )

    while not Q.empty():
        # vyberieme z fronty ďalšie políčko a spracujeme ho
        cr, cs = Q.get()
        for nr, ns in [ (cr+1,cs), (cr-1,cs), (cr,cs+1), (cr,cs-1) ]:
            if nr < 0 or nr >= R or ns < 0 or ns >= S: continue  # mimo mapy nesmieme
            if mapa[nr][ns] in '#X': continue                    # do steny tiež nesmieme
            if vzdialenost[nr][ns] != nekonecno: continue        # toto políčko sme už videli
            vzdialenost[nr][ns] = vzdialenost[cr][cs] + 1
            Q.put( (nr,ns) )

    return vzdialenost

# načítame vstup
```

```
R, S = [ int(_) for _ in input().split() ]
mapa = [ input() for r in range(R) ]

# nájdeme súradnice Peťky a východu
pr, ps, vr, vs = None, None, None, None
for r in range(R):
    for s in range(S):
        if mapa[r][s] == 'P': pr, ps = r, s
        if mapa[r][s] == 'V': vr, vs = r, s

# spustíme dve prehľadávania: od Peťky a od východu
Pvzdialenost = prehladaj( mapa, (pr,ps) )
Vvzdialenost = prehladaj( mapa, (vr,vs) )

# prezrieme všetky možnosti pre najkratšiu cestu
odpoved = Pvzdialenost[vr][vs]

for r in range(R):
    for s in range(S):
        if mapa[r][s] == 'X':
            # skúsime ísť cez túto stenu
            pv, vv = nekonecno, nekonecno
            for sr, ss in [ (r+1,s), (r-1,s), (r,s+1), (r,s-1) ]:
                pv = min( pv, Pvzdialenost[sr][ss] )
                vv = min( vv, Vvzdialenost[sr][ss] )
            odpoved = min( odpoved, pv+vv+2 )

print(odpoved if odpoved < nekonecno else 'smola')
```

Technickejšie riešenie

Toto druhé riešenie je možno o malý chlp menej pekné, ale má aj svoje výhody: netreba naň žiadne triky a vieme ho ľahšie zovšeobecniť.

Na predchádzajúce riešenie sa môžeme dívať tak, že hľadáme najkratšiu cestu v grafe, ktorého vrcholmi sú jednotlivé políčka bludiska. V tomto druhom riešení budeme tiež hľadať najkratšiu cestu, ale v inom grafe.

Vrcholmi nášho grafu budú jednoducho všetky možné situácie, ktoré môžu nastať, keď Peťka chodí po bludisku. Každú takúto situáciu vieme popísať tromi číslami: súradnice políčka, na ktorom Peťka práve stojí, a počet elixírov, ktoré ešte má (nula alebo jeden).

Hrany v takomto grafe predstavujú akcie, ktoré môže Peťka spraviť. Niektoré hrany budú predstavovať krok na susedné voľné políčko. Iné hrany budú predstavovať použitie elixíru a následný krok do steny.

Celé riešenie teda vyzerá tak, že si postavíme vyššie popísaný graf a následne naň pustíme prehľadávanie do šírky. Tým nájdeme najkratšiu cestu zo stavu „Peťka je na začiatku a má jeden elixír“ do niektorého zo stavov v ktorých je Peťka na políčku, kde je východ z bludiska.

Aj toto riešenie má časovú zložitosť $O(rs)$, keďže najskôr v takomto čase postavíme graf (ktorý má nanajvýš $2rs$ vrcholov a z každého z nich vedú nanajvýš štyri hrany) a potom v takomto čase zbehne prehľadávanie do šírky.

Ako sme už uviedli vyššie, jednou z výhod tohto prístupu je možnosť jeho zovšeobecnenia. Keby napríklad Peťka mala elixírov až 7, prvé riešenie vôbec nevieme použiť, zatiaľ čo toto druhé by sme stále mohli použiť, takmer bez zmeny.

Listing programu (Python)

```
from queue import Queue # fronta

nekonecno = 10**9

def prehladaj(mapa, start):
    # prehľadaj do šírky danú mapu z daného štartového stavu (riadok, stĺpec, počet elixírov)

    R, S = len(mapa), len(mapa[0])
    sr, ss, se = start
    vzdialenost = [ [ [ nekonecno for e in range(2) ] for s in range(S) ] for r in range(R) ]
    vzdialenost[sr][ss][se] = 0
    Q = Queue()
    Q.put( (sr,ss,se) )

    while not Q.empty():
        # vyberieme z fronty ďalšie políčko a spracujeme ho
        cr, cs, ce = Q.get()
        for nr, ns in [ (cr+1,cs), (cr-1,cs), (cr,cs+1), (cr,cs-1) ]:
            if nr < 0 or nr >= R or ns < 0 or ns >= S: continue  # mimo mapy nesmieme
            if mapa[nr][ns] in '#': continue                     # do betónovej steny tiež nesmieme
```

```
            ne = ce
            if mapa[nr][ns] in 'X':                            # do tehlovej steny len s elixírom
                if ce == 0: continue                           #    ... ktorý ak nemáme, tak sem nesmieme
                ne = ce-1
            if vzdialenost[nr][ns][ne] != nekonecno: continue  # toto políčko sme už videli
            vzdialenost[nr][ns][ne] = vzdialenost[cr][cs][ce] + 1
            Q.put( (nr,ns,ne) )

    return vzdialenost

# načítame vstup
R, S = [ int(_) for _ in input().split() ]
mapa = [ input() for r in range(R) ]

# nájdeme súradnice Peťky a východu
pr, ps, vr, vs = None, None, None, None
for r in range(R):
    for s in range(S):
        if mapa[r][s] == 'P': pr, ps = r, s
        if mapa[r][s] == 'V': vr, vs = r, s

# spustíme prehľadávanie od Peťky s elixírom
Pvzdialenost = prehladaj( mapa, (pr,ps,1) )

# nájdeme a vypíšeme menšiu z možností "Peťka pri východe s elixírom" a "... bez neho"
odpoved = min( Pvzdialenost[vr][vs] )
print(odpoved if odpoved < nekonecno else 'smola')
```

B-II-4 Tabuľa II

Prejdeme si postupne riešenia všetkých podúloh.

Podúloha A: počet možností ako vpísať znamienka

Viktor vpisuje 20 znamienok. Pre každé z nich má na výber dve množnosti. Všetky výbery sú navzájom nezávislé, preto existuje presne 2^{20} možností ako vybrať znamienka.

Podúloha B: ako dosiahnuť jednoznačné výsledky

Existuje veľa správnych riešení. Jedným z najjednoduchších je napísať na tabuľu rôzne mocniny dvoch, teda napríklad čísla $1, 2, 4, 8, \cdots, 2^{18}, 2^{19}$.

Prečo má táto postupnosť čísel želanú vlastnosť?

Predstavme si, že znamienka dopĺňame postupne zľava doprava a priebežne si počítame výsledok, ktorý sme dostali. Matematickou indukciou dokážeme nasledovné tvrdenie: doplnením prvých k znamienok vieme ako výsledok vyrobiť ľubovoľné nepárne číslo od $-(2^k - 1)$ po $2^k - 1$, a to každé práve jedným spôsobom.

Pre $k = 1$ to platí: po doplnení prvého znamienka máme buď $+1$ alebo -1.

Pozrime sa teraz na indukčný krok. Nech teda vieme doplnením prvých k znamienok vyrobiť ľubovoľné nepárne číslo od $-(2^k - 1)$ po $2^k - 1$, každé práve jedným spôsobom. Čo vieme vyrobiť doplnením prvých $k+1$ znamienok?

Ak za prvých k čísel pripíšeme $+2^k$, dostaneme kladný výsledok: od $-(2^k - 1) + 2^k = 1$ až po $2^k - 1 + 2^k = 2^{k+1} - 1$. No a ak za ne namiesto toho pripíšeme -2^k, dostaneme záporný výsledok od $-(2^{k+1} - 1)$ po -1. No a to je presne to, čo sme chceli dostať: vyrobili sme všetky čísla, ktoré sme vyrobiť mali, a keďže kladné a záporné čísla sa neprekrývajú, každé číslo sme vyrobili práve jedným spôsobom.

Doplnením všetkých 20 znamienok vieme teda vyrobiť presne 2^{20} rôznych čísel, každé práve jedným spôsobom. Ide o všetky nepárne čísla od $-(2^{20} - 1)$ po $2^{20} - 1$.

Podúloha C: Vie Viktor vyrobiť 42 aj 47?

Anička má pravdu, Viktor sa musí mýliť. Totiž ak zmeníme ľubovoľné $+x$ na $-x$ alebo naopak, zmení sa výsledná hodnota celého výrazu o $2x$. A keďže všetky čísla sú celé, zmena o $2x$ nikdy nezmení paritu výsledku. Nech teda akokoľvek preklápame znamienka, všetky výsledky, ktoré vieme vyrobiť, budú mať vždy rovnakú paritu. Ak teda existuje nejaká kombinácia plusov a mínusov, ktorá vedie k výsledku 42, tak úplne všetky dosiahnuteľné výsledky musia byť párne – a teda sa určite nedá dosiahnuť výsledok 47.

Podúloha D: K niektorému výsledku musí viesť veľa možných ciest

Kľúčom k riešeniu tejto podúlohy je takzvaný Dirichletov princíp, tiež známy ako holubníkový princíp. Základná verzia tohto tvrdenia je veľmi jednoduchá: nech akokoľvek rozmiestnite $n+1$ holubov do n holubníkov, vždy bude existovať holubník, v ktorom sú aspoň dva holuby.

My použijeme trochu všeobecnejšiu verziu, a to nasledovnú: nech akokoľvek rozmiestnite 2^{20} holubov do n holubníkov, vždy bude existovať holubník, v ktorom je aspoň $2^{20}/n$ holubov. (Dôkaz je zjavný: keby ich v každom z n holubníkov bolo ostro menej ako $2^{20}/n$, tak ich je v súčte menej ako $n \cdot (2^{20}/n) = 2^{20}$, čo je spor.)

Čo budú holubníky?

Ak Anička napíše na tabuľu 20 čísel z rozsahu od 1 po 100, tak vieme, že všetky možné výsledky ležia v rozsahu od -2000 po 2000. Máme teda 2^{20} „holubov“ (možností ako vpísať znamienka), ale len 4001 „holubníkov“ (výsledkov, ktoré môžeme dostať). Niektorému výsledku teda musí zodpovedať aspoň $2^{20}/4001 \approx 262.078$ rôznych kombinácií znamienok.

Navyše si môžeme spomenúť, že vyššie sme si zdôvodnili, že všetky dosiahnuteľné výsledky majú rovnakú paritu. Holubníkov teda určite nemôže byť viac ako 2001, čo vedie k priemeru ≈ 542.026 holuba na holubník.

Anička má teda opäť pravdu, a dokonca to s číslom 107 výrazne podcenila. Namiesto 107 by pokojne mohla povedať 543 a ešte stále by mala pravdu.

Poznámka na záver: Ani vyššie uvedený výsledok ešte nie je optimálny. Jemnejšími a detailnejšími úvahami sa dá táto hranica posunúť ešte o dosť ďalej. (Napríklad si rozmyslite, že možných výsledkov vlastne vždy musí byť ostro menej ako 2001. Taktiež si rozmyslite, že keď máme veľa rôznych možných výsledkov, tak tým najväčším a najmenším spomedzi nich zodpovedá len veľmi málo rôznych kombinácií znamienok – o to viac nám ich potom ostane na zvyšné možné výsledky.)

TRIDSIATY DRUHÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2017