P-II-1 Petra lyžuje

Pro každou branku jsou dvě možnosti: buď ji Petra projde čistě, nebo tam udělá chybu. Dohromady proto existuje 2^n možných průjezdů tratí. Úlohu tedy umíme vyřešit hrubou silou tak, že postupně vygenerujeme všechny možné průjezdy a pro každý si spočítáme, kolik dohromady Petra ztratí (součet s_i za branky, kde udělala chybu) a s jakou pravděpodobností takový průjezd nastane (součin pravděpodobností zvoleného typu průjezdu brankou). Celkovou pravděpodobností výhry je pak součet pravděpodobností těch průjezdů tratí, při kterých ztratí dostatečně málo.

Řešení za 5 bodů: nanejvýš dvě libovolné chyby

Pokud víme, že Petra má náskok 25 setin a každou chybou z něj ztratí 10, je zjevné, že si může na trati dovolit nanejvýš dvě chyby – a to na libovolných brankách. Potřebujeme tedy spočítat celkovou pravděpodobnost takových průjezdů tratí.

Pravděpodobnost c čistého průjezdu umíme vypočítat jako součin $p_1 \cdot p_2 \cdots p_n$. Když už známe hodnotu c, umíme z ní vyjádřit pravděpodobnosti průjezdů s jednou i dvěma chybami.

Mějme například konkrétní branku j. Pravděpodobnost toho, že Petra celou trať přejede čistě, až na branku j, kde udělá chybu, udává vzorec $p_1 \cdot p_2 \cdots p_{j-1} \cdot (1 - p_j) \cdot p_{j+1} \cdots p_n$. Tento součin však nemusíme celý počítat znovu: jeho hodnotu umíme vyjádřit z pravděpodobnosti c tak, že ji vydělíme p_j a následně vynásobíme $1 - p_j$.

Obdobně můžeme v konstantním čase pro každé (k, ℓ) vypočítat pravděpodobnost toho, že Petra pokazí právě branky k a ℓ. Sečtením všech $\mathcal{O}(n^2)$ takto vygenerovaných pravděpodobností dostáváme odpověď.

Zlepšení časové složitosti

Výše popsané řešení mělo z pochopitelných důvodů kvadratickou časovou složitost. Hledanou odpověď však umíme vypočítat i v lineárním čase pomocí o trochu chytřejší matematiky.

Označme $x_j = (1 - p_j)/p_j$. Pak zjevně platí, že $c \cdot x_j$ je pravděpodobnost průjezdu s chybou právě na brance x_j a $c \cdot x_k \cdot x_\ell$ je pravděpodobnost průjezdu s chybami právě na brankách k a ℓ. Celková pravděpodobnost průjezdu s jednou chybou je tedy $cx_1 + cx_2 + \cdots + cx_n = c(x_1 + \cdots + x_n)$ a se dvěma chybami obdobně $c(x_1x_2 + x_1x_3 + \cdots + x_{n-1}x_n)$.

Označme si nyní $u = x_1 + \cdots + x_n$ a $v = x_1x_2 + x_1x_3 + \cdots + x_{n-1}x_n$. Pokud bychom věděli hodnoty u a v, máme vyhráno: celková pravděpodobnost, že Petra vyhraje, je $c + cu + cv$.

Hodnotu u snadno spočítáme v lineárním čase, ale na přímý výpočet v bychom potřebovali kvadratický čas. Jak to udělat rychleji? Když si roznásobíme $u^2 = (x_1 +$

$\cdots + x_n)^2$, uvidíme, že výsledek se dost podobá $2v$. Navíc jsou tam pouze členy tvaru x_i^2. Hodnotu $w = x_1^2 + x_2^2 + \cdots + x_n^2$ umíme ale také spočítat v lineárním čase, a když od u^2 odečteme w, zůstane nám $2v$ a vyhráli jsme.

Zkuste se zamyslet, jak by se toto řešení dalo zobecnit pro tři, čtyři, nebo dokonce obecně mnoho chyb. Jak bude časová složitost záviset na počtu chyb?

Řešení pro z libovolných chyb

K osmibodovému řešení vede i lehčí cesta než ta z předchozí části.

Pokud víme, že všechny s_i jsou stejné, víme, že Petra vyhraje právě tehdy, pokud na trati udělá nanejvýš $z = \lfloor c/s_1 \rfloor$ chyb. Potřebujeme tedy spočítat celkovou pravděpodobnost toho, že se toto stane. Namísto toho, abychom si ručně odvozovali vzorce, necháme počítač udělat většinu práce za nás.

Když ještě Petra stojí na startu, víme, že s pravděpodobností 1 neudělala žádnou chybu. Po první brance jsou dvě možnosti: s pravděpodobností p_1 má stále nula chyb, s pravděpodobností $1 - p_1$ má první chybu. Toto si můžeme představit jako dva nezávislé světy: s pravděpodobností p_1 se nachází v prvním z nich, s pravděpodobností $1 - p_1$ ve druhém.

V každém z těchto světů Petra následně projde druhou branou. S pravděpodobností p_2 ji přejde čistě, s pravděpodobností $1 - p_2$ tam udělá chybu. Každý možný svět se nám tím opět rozdělil na dva nové.

Po dvou brankách jsou tedy čtyři možné světy: s pravděpodobností $p_1 p_2$ je Petra ve světě, ve kterém obě branky prošla čistě, s pravděpodobností $p_1(1 - p_2)$ ve světě, kde měla chybu na druhé brance, s pravděpodobností $(1 - p_1)p_2$ je ve světě, kde měla chybu na první brance, a s pravděpodobností $(1 - p_1)(1 - p_2)$ je ve světě, kde pokazila obě branky.

Pokud bychom takto pokračovali dál, skončili bychom s 2^n světy a v podstatě bychom tedy udělali tutéž práci jako při řešení hrubou silou. Tady ale přijde klíčové pozorování: druhý a třetí svět jsou rovnocenné. V obou totiž Petra udělala právě jednu chybu, a tudíž je v přesně stejné situaci: pokud v obou světech udělá stejný zbytek jízdy, dostane v cíli stejný výsledek.

Můžeme si tedy naše pozorování přeformulovat do následující mnohem užitečnější podoby. Po dvou brankách je Petra v jednom z **tří** možných světů, a to:

- S pravděpodobností $p_1 p_2$ je ve světě, kde ještě neudělala chybu.
- S pravděpodobností $p_1(1 - p_2) + (1 - p_1)p_2$ je ve světě, kde udělala právě jednu chybu.
- S pravděpodobností $(1 - p_1)(1 - p_2)$ je ve světě, kde udělala právě dvě chyby.

Pokračováním této úvahy dostaneme mnohem efektivnější řešení než hrubou silou – časová složitost bude polynomiální v počtu branek.

Pořádnější formulace řešení pro z chyb

Označme si jako $q_{i,j}$ pravděpodobnost toho, že při průjezdu prvními i brankami Petra udělá přesně j chyb. Na začátku víme, že $q_{0,0} = 1$ a pro všechny ostatní j platí $q_{0,j} = 0$.

Pokud nyní známe všechny hodnoty $q_{i,\star}$ pro nějaké i, snadno z nich určíme všechny hodnoty $q_{i+1,\star}$. Stačí zopakovat výše popsanou úvahu pro každý z možných světů. Z té dostaneme, že pro každé j platí

$$q_{i+1,j} = q_{i,j}p_{i+1} + q_{i,j-1}(1 - p_{i+1}).$$

Do stavu „po $i+1$ brankách mám j chyb“ se totiž Petra umí dostat dvěma různými způsoby: buď měla po i brankách j chyb (což nastane s pravděpodobností $q_{i,j}$, kterou již známe) a následně projede branku $i+1$ čistě, nebo měla po i brankách $j-1$ chyb a v následující udělala další, tedy j-tou chybu.

Výše popsaným vzorcem každou z hodnot $q_{i,j}$ určíme v konstantním čase. Když už je všechny známe, celkovou pravděpodobnost toho, že Petra vyhraje, umíme vyjádřit jako $q_{n,0} + q_{n,1} + \cdots + q_{n,z}$. Celková časová složitost tohoto řešení je $\mathcal{O}(n^2)$, pokud určíme všechny nenulové prvky pole q. Stačí však počítat hodnoty pro počet chyb nepřesahující z, čímž časovou složitost mírně zlepšíme na $\mathcal{O}(nz)$.

Zcela obecné řešení

Namísto počtu chyb můžeme pracovat přímo s Petřiným náskokem v setinách. Budeme si tedy (velmi podobným způsobem jako výše) počítat pravděpodobnosti toho, že po i brankách má Petra ještě j setin náskoku.

```
N, T = [ int(_) for _ in input().split() ]
P, S = [], []
for n in range(N):
    tokens = input().split()
    P.append( float(tokens[0]) / 100 )
    S.append( int(tokens[1]) )

first_row = [ 0. for _ in range(T+1) ]
first_row[T] = 1.
Q = [ first_row ]
for i in range(N):
    next_row = [ 0. for _ in range(T+1) ]
    for t in range(T+1):
        next_row[t] += Q[i][t] * P[i]
        if t >= S[i]:
            next_row[t-S[i]] += Q[i][t] * (1-P[i])
    Q.append( next_row )

print( 100. * sum( Q[N] ) )
```

Tento program má zjevně časovou složitost $\mathcal{O}(nt)$, jelikož jeho časově nejnáročnější částí jsou dva vnořené cykly, vnější s n iteracemi a vnitřní s $t+1$ iteracemi. Paměťová složitost je také $\mathcal{O}(nt)$, ale dá se snadno zlepšit na $\mathcal{O}(n+t)$, když si všimneme, že po dopočítání hodnot $q_{i+1,\star}$ pro dané i můžeme hodnoty $q_{i,\star}$ zapomenout.

Jiná formulace obdobného řešení je představit si rekurzivní funkci, která na vstupu dostane parametry i a j a na výstupu vrátí odpověď na otázku, s jakou pravděpodobností Petra vyhraje, pokud již projela i branek a má ještě j setin náskok. Každou takovou otázku umíme zodpovědět pomocí rekurze: pokud ještě nejsme na konci, vyzkoušíme obě možnosti pro následující branku a v každé z nich se následně rekurzivně zavoláme, abychom zjistili odpověď pro situaci, do které jsme se právě dostali.

Opět platí, že kdybychom takovou rekurzivní funkci implementovali a spustili, dostali bychom exponenciální časovou složitost – rekurzivní funkce postupně *backtrackingem* vyzkouší všechny možné průjezdy tratí.

Klíčové je ale pozorování, že tento program sice dělá exponenciálně mnoho volání funkce, ty ale nejsou všechny navzájem různé – naopak často se opakují a pak zbytečně znovu a znovu počítáme totéž. *Různých* relevantních volání naší funkce je jen $\mathcal{O}(nt)$: branek je n a Petřin aktuální náskok nikdy nebude větší než t. A jelikož výstup naší funkce je vždy jednoznačně určen jejími vstupy, můžeme použít *memoizaci:* jakmile nějakou hodnotu spočítáme, zapamatujeme si ji v tabulce, a vždy, když ji později znovu potřebujeme, místo nového pracného výpočtu jen rovnou použijeme zapamatovanou hodnotu.

```
from functools import cache

N, T = [ int(_) for _ in input().split() ]
P, S = [], []
for n in range(N):
    tokens = input().split()
    P.append( float(tokens[0]) / 100 )
    S.append( int(tokens[1]) )

@cache
def vypocitej(i, j):
    """
    Vrátí pravděpodobnost toho, že Petra vyhraje, jestliže po i brankách má
    j setin náskoku. Dekorátor @cache za nás dělá memoizaci.
    """
    if j < 0: return 0.    # Už nemůže vyhrát
    if i == N: return 1.   # Už vyhrála
    return P[i] * vypocitej(i+1, j) + (1-P[i]) * vypocitej(i+1, j-S[i])

print( 100. * vypocitej(0, T) )
```

Tento program má také celkovou časovou složitost $\mathcal{O}(nt)$, protože jediné, co dělá, je průběžně počítá a zapisuje si odpovědi na otázky výše popsaného typu. Otázek je $\mathcal{O}(nt)$ a odpověď na každou z nich (bez započítání času stráveného rekurzivními voláními) zjistíme v konstantním čase. Paměťová složitost je $\mathcal{O}(nt)$ a oproti prvnímu programu ji nejde snadno vylepšit.

P-II-2 Jedno obrácení

Existuje $\mathcal{O}(n^2)$ možností, jak zvolit úsek, který obrátíme. Vyzkoušíme-li všechny, můžeme si být jisti, že najdeme správné řešení. Pokud budeme zkoušet každý úsek zvlášť, budeme potřebovat čas $\mathcal{O}(n)$ na každou kontrolu, a tak dostaneme řešení s celkovou časovou složitostí $\mathcal{O}(n^3)$.

Lepší řešení dostaneme, když si uvědomíme, že můžeme dohromady zpracovávat všechny úseky se stejným středem. Například obrácení úseku od 8 do 20 je skoro totéž jako obrácení úseku od 9 do 19, jen ještě navíc vyměníme jeho konce, tedy prvky na indexech 8 a 20.

Projdeme tedy postupně přes všechny možné středy úseku. Pro každý střed začneme od nejkratšího úseku a postupně zvětšujeme délku a průběžně si počítáme, kolik prvků je momentálně na svých místech. Možných středů úseků je $2n-1$: úseky liché délky mají svůj střed na políčku, zatímco pro úseky sudé délky se jejich střed nachází mezi dvěma sousedními políčky. Takové řešení má časovou složitost $\mathcal{O}(n^2)$, neboť na každý možný úsek pole se podíváme právě jednou a když se tak stane, zpracujeme ho v konstantním čase.

Myšlenky vedoucí k lepšímu řešení

Chceme-li lepší než kvadratickou časovou složitost, nemůžeme si už dovolit projít všechny možné úseky. Budeme tedy muset nějak chytřeji vybrat úseky, které má smysl zkoušet. Začněme tím, že si prvky v poli A rozdělíme na dva typy: ty, které momentálně na správném místě jsou („dobré“), a ty, které nejsou („špatné“).

Dobré prvky si můžeme jen pokazit: pokud obrátíme úsek, všechny dobré prvky v něm přestanou být dobré, a to s jednou výjimkou: pokud obrátíme úsek liché délky a prvek v jeho středu byl dobrý, zůstane dobrý.

Špatné prvky většinou zůstanou špatné, většina obrácení špatný prvek totiž přesune na jiné nesprávné místo. Podívejme se na konkrétní špatný prvek a zamysleme se nad tím, jak z něj udělat dobrý. Mějme například hodnotu 4 na indexu 17. Která obrácení zafungují pro tuto hodnotu? Zjevně právě ty, které vymění políčko 4 s políčkem 17. A co mají všechna tato obrácení společného? Zjevně mají tentýž střed: je uprostřed mezi políčky 4 a 17.

Pro každý špatný prvek tedy zjevně existuje právě jeden střed obrácení, při kterém (pokud uvažujeme úsek dostatečné délky) se z tohoto špatného prvku stane dobrý. U všech ostatních obrácení tento prvek zaručeně zůstane špatný.

Zpracování dobrých prvků

Uvažujme pole B, ve kterém $B[i] = 1$ pokud $A[i] = i$ a $B[i] = 0$ jinak. Chceme-li zjistit, kolik dobrých prvků obsahuje konkrétní úsek pole A, stačí zjistit součet odpovídajícího úseku pole B.

Abychom toto uměli dělat rychle, předpočítáme si prefixové součty pole B (viz např. *https://ksp.mff.cuni.cz/kucharky/zakladni-algoritmy/*). Díky nim pak umíme pro libovolný úsek pole A v konstantním čase říci, kolik z původně dobrých prvků zůstane dobrých i po jeho obrácení.

Nalezení zajímavých obrácení

Pro každý špatný prvek jsme si už našli jeho jediný správný střed obraceného úseku. Nyní se na tatáž data podívejme z opačné strany. Vezměme si konkrétní střed obrácení. Pro něj jsme právě zjistili, které špatné prvky umí změnit na dobré. Tyto prvky od teď nazývejme „nadějné“. Každému nadějnému prvku odpovídá nějaká minimální délka obraceného úseku, od které bude přesunut na správné místo.

Představme si nyní, že máme pevný střed a postupně zvětšujeme délku obraceného úseku. Jak se mění uklizenost výsledného pole? Občas klesne a občas stoupne. Klesnout může jen tehdy, když do obraceného úseku přibude nějaký původně dobrý prvek. A podobně stoupnout může jen tehdy, když do něj přibude nějaký nadějný prvek. Pokud tedy chceme najít nejlepší obrácení s tímto středem, zjevně se stačí dívat právě na ty délky, pro které do obraceného úseku právě přibyl nový nadějný prvek. Žádná jiná délka nám nemůže dát lepší řešení.

Toto nám dává téměř úplný návod, jak zpracovat konkrétní střed: Uspořádáme si jeho nadějné prvky vzestupně podle jejich minimální délky obraceného úseku. V tomto pořadí pak projdeme všechny tyto zajímavé délky. Pro každou délku umíme v konstantním čase říci, kolik nadějných prvků dá obrácení úseku této délky na správné místo (podle indexu, na kterém jsme v uspořádaném poli délek) a také v konstantním čase umíme říct, kolik dobrých prvků dá toto obrácení ze správného místa pryč (pomocí dříve předpočtených prefixových součtů).

Jelikož každý špatný prvek je nadějný jen pro jeden střed, dohromady toto řešení zpracuje každý špatný prvek jen jednou. Celková časová složitost je tedy lineární, až na jeden detail: logaritmus navíc kvůli potřebě pořádat seznamy nadějných prvků. Máme tedy řešení v čase $\mathcal{O}(n \log n)$.

Rychlejší třídění

Logaritmu se zbavíme tak, že si uvědomíme, že prvky, které řadíme dle velikosti (délky úseků) jsou celá čísla z rozsahu od 1 do n. Můžeme tedy ke třídění použít přihrádkové třídění.

Je ale nutné ještě dořešit technické detaily. Nemůžeme si dovolit dělat pro každý střed zvlášť jedno přihrádkové třídění, neboť u každého z nich bychom strávili $\mathcal{O}(n)$ času vyprázdněním přihrádek. Namísto toho to uděláme následovně:

Na začátku projdeme celé pole A a pro každý špatný prvek si najdeme minimální délku úseku, jehož obrácením se dostane na správné místo. Nyní jedním přihrádkovým tříděním uspořádáme všechny špatné prvky najednou podle této délky. Když už je máme takto uspořádané, tak je ještě jednou projdeme a přerozdělíme k jednotlivým středům. U každého středu tak dostaneme správně uspořádaný seznam jeho nadějných prvků. Na toto přerozdělení se případně můžeme dívat jako na druhé kolo přihrádkového třídění, při kterém nyní všechny špatné prvky přeuspořádáme podle souřadnice jejich středu obrácení, přičemž v případě rovnosti zachováme aktuální pořadí.

Máme-li pro nějaký střed dva nadějné prvky se stejnou délkou (při obrácení si vymění místa a oba budou dobré), není třeba to nijak speciálně ošetřovat. V níže

uvedeném programu příslušnou délku obrácení prostě zpracujeme postupně dvakrát, jednou pro každý prvek. Při druhém zpracování dostaneme správnou hodnotu uklízenosti pro toto obrácení; při prvním o jedna menší, což vůbec nevadí.

Časová i paměťová složitost výsledného řešení je $\mathcal{O}(n)$.

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

int main() {
    int N; cin >> N;
    vector<int> A(N); for (int &a : A) cin >> a;

    // Předpočítáme B a jeho prefixové součty.
    vector<int> B(N);
    for (int n=0; n<N; ++n) B[n] = int( A[n] == n );
    vector<int> SB(1, 0);
    for (int b : B) SB.push_back( SB.back()+b );

    // Uspořádáme špatné prvky dle středu a v rámci středu dle minimální délky
    // úseku, jehož obrácením se dostanou na správné místo.
    vector<vector<int>> podle_delky(N);
    for (int n=0; n<N; ++n)
        if (A[n] != n)
            podle_delky[ abs(A[n]-n) ].push_back(n);
    vector<vector<int>> podle_stredu(2*N-1);
    for (auto row : podle_delky)
        for (int x : row)
            podle_stredu[ x+A[x] ].push_back(x);

    // Projdeme všechny kandidáty na nejlepší řešení.
    int bestupr = SB[N], bestzac = 0, bestkon = 0;
    for (int stred=0; stred < 2*N-1; ++stred) {
        for (int i=0; i < int(podle_stredu[stred].size()); ++i) {
            int x = podle_stredu[stred][i];
            int delka = abs( A[x] - x );
            int zac = (stred-delka)/2, kon = (stred+delka)/2;
            // Zpracováváme úsek od zac do kon.
            int upr = SB[N]; // Tolik byla uklizenost před obrácením.
            upr += (i+1);    // Tolik nadějných prvků se změní na dobré.
            upr -= SB[kon+1] - SB[zac]; // Tolik dobrých prvků jsme zkazili,
            // ale dobrý prvek uprostřed obráceného úseku zůstává dobrý.
            if (stred%2 == 0 && A[stred/2] == stred/2)
                ++upr;
            if (upr > bestupr) {
                // Našli jsme nové optimum.
                bestupr = upr; bestzac = zac; bestkon = kon;
            }
        }
    }
    cout << bestzac << " " << bestkon << endl;
}
```

P-II-3 Těžkotonážní skokobot

Kvůli lepší čitelnosti budeme při odhadech časové složitosti předpokládat, že bludiště je čtvercového tvaru o rozměrech $n \times n$. Všechny algoritmy ovšem samozřejmě fungují i pro obdélníkové vstupy a čtenář si jistě snadno doplní přesnější odhady, bude-li je potřebovat.

Cesta po akcích

Pokud víme, jakou akci robot udělal jako poslední, umíme jednoznačně říct, jaké akce může dělat nyní. Pokud byl poslední akcí krok, můžeme kráčet nebo skákat libovolným směrem, pokud jí ale byl skok, můžeme kráčet i skákat jen týmž směrem.

Každou akci můžeme popsat například políčkem, kde začíná a druhým políčkem (ve stejném řádku nebo sloupci), kde končí. Jiný, ekvivalentní popis akce sestává z políčka kde začíná, směru kterým vede a délky pohybu. Dohromady zjevně existuje $\mathcal{O}(n^3)$ možných akcí.

Představme si nyní graf, jehož vrcholy jsou všechny možné akce a jehož (orientované) hrany říkají, že bezprostředně po akci x můžeme udělat akci y. Pro každou konkrétní akci existuje jen $\mathcal{O}(n)$ možností jak vypadá následující: začáteční políčko je pevné, zvolit si můžeme jen délku a možná směr. Dohromady má proto tento graf $\mathcal{O}(n^4)$ hran.

Pomocí prohledávání do šířky můžeme v tomto grafu najít nejkratší cestu začínající libovolnou akcí vedoucí z levého horního rohu a končící krokem do pravého dolního rohu. Tato nejkratší cesta grafem zjevně přímo odpovídá nejkratší možné posloupnosti akcí řešící naši úlohu.

Máme tedy první korektní řešení. Jeho časová složitost je $\mathcal{O}(n^4)$.

Šikovněji skáčeme

Lepší řešení začneme tím, že si uvědomíme, že se nám nikdy nevyplatí udělat dva skoky po sobě. Každá posloupnost po sobě jdoucích skoků musí celá vést jedním směrem a končit krokem stejným směrem. V optimálním řešení se zjevně nikdy nevyplatí dělat po sobě více skoků, neboť všechny po sobě jdoucí skoky stejným směrem můžeme nahradit jedním velkým skokem. Například místo dvou po sobě jdoucích skoků délek 7 a 3 stačí udělat jeden skok délky 10 a ušetřit tak akci.

Naše úloha je tedy ekvivalentní s následující: robotovi umíme zadat libovolnou posloupnost příkazů „udělej krok“ (1 akce) a „skoč a pak udělej krok stejným směrem“ (2 akce). Uvažujme nyní graf, jehož vrcholy jsou prázdná políčka původní mapy. Z každého vrcholu vedou ≤ 4 hrany délky 1 odpovídající platným krokům a $\mathcal{O}(n)$ hran délky 2 odpovídající platným kombinacím skok+krok. I v tomto grafu platí, že nejkratší cesta (tentokrát přímo z políčka vlevo nahoře na políčko vpravo dole) odpovídá nejkratší posloupnosti akcí, kterou hledáme.

Také v tomto grafu umíme nejkratší cestu najít prohledáváním do šířky. Buď ho upravíme tak, aby umělo pracovat i s hranami délky 2, nebo si ještě před spuštěním prohledávání na každou hranu délky 2 přidáme do středu nový vrchol, který ji rozdělí na dvě hrany délky 1. Časová složitost tohoto řešení je $\mathcal{O}(n^3)$.

Šikovněji zpracováváme skoky

Předchozí řešení ještě dělá spoustu práce zbytečně dvakrát. Pokud jsme již vyzkoušeli všechny možnosti, jak skočit doprava z políčka $(4, 7)$, a pak později zpracováváme jeho pravého souseda, tj. políčko $(4, 8)$, ničeho nového už skákáním doprava nedosáhneme. Jak ale zabránit prohlížení téhož dvakrát?

Postavme si ještě jeden nový graf. Tentokrát budeme mít vrcholy dvou typů:

- Jeden vrchol pro každý stav „stojím na políčku (x, y)“.
- Jeden vrchol pro každý stav „jsem ve vzduchu nad políčkem (x, y) a skáču směrem d“.

Stavů prvního typu je nanejvýš n^2, stavů druhého typu je nanejvýš $4n^2$, dohromady tedy má náš graf $\mathcal{O}(n^2)$ vrcholů. Hrany budou opět orientované a budou vést do stavů, které mohou bezprostředně následovat. Hrany tedy budou vypadat následovně:

- Z každého stavu, ve kterém stojíme, budeme mít nanejvýš čtyři hrany odpovídající krokům a dále nanejvýš čtyři hrany odpovídající začátku skoku každým možným směrem.
- Z každého stavu, ve kterém letíme vzduchem, budeme mít hranu odpovídající letu o políčko dál.
- Pokud jsme ve vzduchu a naše políčko i to bezprostředně následující naším směrem jsou volné, budeme mít hranu do stavu, ve kterém stojíme na tom následujícím políčku.

Hrany odpovídající tomu, že dále letíme vzduchem, budou mít délku 0. Ostatní hrany budou mít délku 1. Snadno ověříme, že libovolný krok odpovídá přechodu po hraně délky 1 a pro libovolný skok+krok přejedeme hranami o délkách 1, 0, 0, ..., 0, 1, a tedy dohromady nějakou cestu délky 2. Vzdálenosti mezi stavy, ve kterých někde stojíme, tedy vždy odpovídají počtu akcí, které robot na příslušnou změnu stavu potřebuje.

Z každého vrcholu vede jen konstantně mnoho hran (nanejvýš 8), dohromady má tedy náš graf nejen $\mathcal{O}(n^2)$ vrcholů, ale i $\mathcal{O}(n^2)$ hran. Nejkratší cestu opět umíme najít algoritmem prohledávání do šířky, tentokrát ho ale potřebujeme upravit tak, aby uměl pracovat is hranami délky 0.

Prohledávání do šířky s hranami délek 0 a 1

V klasickém prohledávání do šířky graf postupně zpracováváme po vrstvách: nejdříve začátek, pak vrcholy ve vzdálenosti 1, pak ty ve vzdálenosti 2, a tak dále. K tomu používáme frontu, ve které čekají vrcholy na zpracování. V každém okamžiku platí, že frontu umíme rozdělit na dvě (možná prázdné) části. V první jsou ještě nezpracované vrcholy z aktuálně zpracovávané vrstvy. Nechť d je vzdálenost, kterou mají všechny tyto vrcholy od startu. Ve druhé části fronty pak na zpracování čekají již objevené vrcholy z následující vrstvy. Tyto mají všechny od startu vzdálenost $d + 1$.

Máme-li v grafu i hrany délky 0 a nějakou takovou hranou objevíme nový vrchol v, víme, že má stejnou vzdálenost d jako ten vrchol, který právě zpracováváme, a tedy patří ještě do aktuální vrstvy. Jak zajistit, abychom v zpracovali ještě s ostatními vrcholy v aktuální vrstvě? Snadno: stačí, když ho zařadíme **na začátek** fronty, ne na její konec.

Některé implementace fronty sice nepodporují vkládání na začátku (pouze výběr na začátku a vkládání na konci), ale existuje několik obecnějších datových struktur, které tuto možnost mají. Dá se např. vhodně použít spojový seznam, ale i v poli umíme šikovně implementovat tzv. obousměrnou frontu (deque), u které umíme na obou koncích i vkládat i vybírat prvky. Detailní popis deque naleznete např. ve vzorových řešeních úlohy OI39-B-I-2 (viz *https://oi.sk/archiv/2023/*).

Obecně se nám při této variantě prohledávání do šířky může stát, že konkrétní vrchol v zařadíme do fronty ke zpracování až dvakrát. Toto nastane tehdy, pokud jej během zpracování vrstvy ve vzdálenosti d nejprve objevíme hranou délky 1, takže mu dáme vzdálenost $d + 1$ a zařadíme jej na konec fronty, ale následně do něj později přijedeme z téže vrstvy i hranou délky 0. Tehdy mu jednoduše snížíme vzdálenost na d a zařadíme jej i na začátek fronty. Nic špatného se nestane, když ho později postupně zpracujeme dvakrát – druhé zpracování už nic nového neobjeví a asymptotickou časovou složitost nám to nezkazí. Jinou možností je pamatovat si explicitně o každém vrcholu, zda již byl zpracován, a každý vrchol zpracovat jen jednou.

Toto řešení má optimální časovou i paměťovou složitost $\mathcal{O}(n^2)$, resp. $\mathcal{O}(rs)$ pro obdélníkové vstupy.

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

typedef vector<int>    pole1d;
typedef vector<pole1d> pole2d;
typedef vector<pole2d> pole3d;
struct stav { int r, s, d; };

// směr 0 označuje, že stojime, směry 1-4 jsou světové strany
const int DR[] = {0, -1, 0, 1, 0};
const int DS[] = {0, 0, 1, 0, -1};

pole3d vzdalenost;
deque<stav> fronta;

void zarad_do_fronty(int stara_vzd, int hrana, int nr, int ns, int nd) {
    int nova_vzd = stara_vzd + hrana;
    if (vzdalenost[nr][ns][nd] <= nova_vzd)
        return; // nic nového jsme nenašli
    vzdalenost[nr][ns][nd] = nova_vzd;
    if (hrana == 0)
        fronta.push_front( {nr,ns,nd} );
    else
        fronta.push_back( {nr,ns,nd} );
}

int main() {
    int R, S;
```

```
    cin >> R >> S;
    vector<string> plan(R);
    for (int r=0; r<R; ++r) cin >> plan[r];

    vzdalenost.resize(R, pole2d(S, pole1d(5, 987654321)));
    vzdalenost[0][0][0] = 0;
    fronta.push_back( {0,0,0} );

    while (!fronta.empty()) {
        stav akt = fronta.front();
        fronta.pop_front();
        int akt_vzd = vzdalenost[ akt.r ][ akt.s ][ akt.d ];
        if (akt.d == 0) {
            // jsme na zemi, můžeme udělat krok nebo začít skok libovolným směrem
            for (int nd=1; nd<=4; ++nd) {
                int nr = akt.r + DR[nd], ns = akt.s + DS[nd];
                // dostali bychom se mimo mapu?
                if (!(0 <= nr && nr < R && 0 <= ns && ns < S))
                    continue;
                // pokud je políčko volné, můžeme udělat krok
                if (plan[nr][ns] == '.')
                    zarad_do_fronty(akt_vzd, 1, nr, ns, 0);
                zarad_do_fronty(akt_vzd, 1, nr, ns, nd); // vždy můžeme začít skok
            }
        } else {
            // jsme ve vzduchu, můžeme letět dál nebo přistát
            int nr = akt.r + DR[akt.d], ns = akt.s + DS[akt.d];
            // vyletěli bychom mimo mapu?
            if (0 <= nr && nr < R && 0 <= ns && ns < S) {
                zarad_do_fronty(akt_vzd, 0, nr, ns, akt.d); // vždy můžeme letět dál
                // je-li pod i před námi volno, můžeme přistát
                if (plan[akt.r][akt.s] == '.' && plan[nr][ns] == '.')
                  zarad_do_fronty(akt_vzd, 1, nr, ns, 0);
            }
        }
    }
    int odpoved = vzdalenost[R-1][S-1][0];
    cout << (odpoved == 987654321 ? -1 : odpoved) << endl;
}
```

P-II-4 O Vekslákbotovi a Pokladničce

Podúloha A: přebarvení na maximum

První instrukce vezme tři žetony navzájem různých barev (červený, modrý a zelený) a nahradí je třemi bílými. Když už se tato instrukce nedá použít, víme, že nám zůstaly právě dvě barvy (protože počty žetonů na začátku byly různé).

Dále si tedy přidáme instrukce „přebarvující“ libovolné dva různé žetony na bílé. V každé situaci se bude dát opakovaně použít právě jedna z těchto instrukcí.

Když už to neumíme udělat, víme, že nám zůstala právě jedna barva: ta, které bylo na začátku nejvíc. Na tu přebarvíme všechny bílé žetony.

1. červená, zelená, modrá → 3bílá
2. červená, zelená → 2bílá
3. červená, modrá → 2bílá
4. zelená, modrá → 2bílá
5. červená, bílá → 2červená
6. zelená, bílá → 2zelená
7. modrá, bílá → 2modrá

Dodáme ještě, že je zjevné, že v situaci, kdy jsou všechny žetony téže barvy, již žádnou instrukci tohoto programu neumíme použít, a tedy se program zastaví.

Podúloha B: přebarvení na minimum

Stejně jako v podúloze A začneme tím, že dokud máme všechny tři barvy, vezmeme po jednom žetonu z každé a nahradíme je třemi bílými.

Jakmile nám zůstanou jen dvě barvy, víme, které bylo nejméně – té, kterou už nemáme. Pokud nám např. ještě zůstaly červené a zelené žetony, víme, že na konci chceme mít všechno modré. Přidáme si proto pro modrou barvu instrukci „červená, zelená → 2 tmavěmodrá“ a dvě obdobné pro ostatní barvy.

Jakmile máme nějaké tmavé žetony, můžeme vše ostatní přebarvit na příslušnou tmavou barvu. A jakmile máme všechny žetony jedné tmavé barvy, můžeme ji přebarvit na původní světlou a skončit.

1. červená, zelená, modrá → 3bílá
2. zelená, modrá → 2tmavočervená
3. červená, modrá → 2tmavozelená
4. červená, zelená → 2tmavomodrá
5. tmavočervená, zelená → 2tmavočervená
6. tmavočervená, modrá → 2tmavočervená
7. tmavočervená, bílá → 2tmavočervená
8. tmavozelená, modrá → 2tmavozelená
9. tmavozelená, červená → 2tmavozelená

10. tmavozelená, bílá → 2tmavozelená
11. tmavomodrá, zelená → 2tmavomodrá
12. tmavomodrá, červená → 2tmavomodrá
13. tmavomodrá, bílá → 2tmavomodrá
14. tmavočervená → červená
15. tmavozelená → zelená
16. tmavomodrá → modrá

Podúloha C: mocnina tří

Vyrobíme si jeden modrý žeton, tedy 3^0 modrých žetonů. Potom postupně c-krát ztrojnásobíme počet modrých žetonů. Přesněji, v každé iteraci tohoto cyklu odstraníme jeden červený žeton, poté změníme každý modrý na tři fialové a poté každý fialový zpět na modrý. Když nám červené žetony dojdou, víme, že máme 3^c modrých žetonů. Na závěr už jen ty přeměníme na červené a skončíme.

Jak ale zajistíme, aby se nám nepomíchalo, které instrukce kdy použít? Obzvlášť ne tehdy, když na konci opět vzniknou červené žetony? K tomu použijeme žetony různých dalších barev, které budou představovat různé fáze, ve kterých se právě náš program nachází. Do programu přidáme omezení říkající, že ze všech těchto barev dohromady nikdy nesmí najednou existovat více než jeden žeton.

Omezení: kontroluj + násob + uklízej + skonči ≤ 1

1. → kontroluj, modrá
2. kontroluj, červená → násob
3. kontroluj → skonči
4. násob, modrá → násob, 3fialová
5. násob → uklízej
6. uklízej, fialová → uklízej, modrá
7. uklízej → kontroluj
8. skonči, modrá → skonči, červená