A-II-1 Peťa lyžuje

Pre každú bránku sú dve možnosti: buď ju Peťa prejde čisto, alebo tam spraví chybu. Dokopy preto existuje 2^n možných prejazdov traťou. Úlohu teda vieme vyriešiť hrubou silou tak, že postupne vygenerujeme všetky možné prejazdy a pre každý si spočítame, koľko dokopy Peťa stratí (súčet s_i za bránky, kde spravila chybu) a s akou pravdepodobnosťou takýto prejazd nastane (súčin pravdepodobností zvoleného typu prejazdu bránkou). Celkovou pravdepodobnosťou výhry je potom súčet pravdepodobností tých prejazdov traťou, pri ktorých stratí dostatočne málo.

Riešenie za 5 bodov: nanajvýš dve ľubovoľné chyby

Ak vieme, že Peťa má náskok 25 stotín a každou chybou z neho stratí 10, je zjavné, že si môže na trati dovoliť nanajvýš dve chyby – a to na ľubovoľných bránkach. Potrebujeme teda spočítať celkovú pravdepodobnosť takýchto prejazdov traťou.
Pravdepodobnosť c čistého prejazdu vieme vypočítať vynásobením $p_1 \cdot p_2 \cdot \dots \cdot p_n$.
Keď už poznáme hodnotu c, vieme z nej vyjadriť pravdepodobnosti prejazdov s jednou aj dvoma chybami.
Napr. majme konkrétnu bránku j. Pravdepodobnosť toho, že Peťa celú trať prejde čisto, až na bránku j, kde spraví chybu, udáva vzorec $p_1 \cdot p_2 \cdot \ldots p_{j-1} \cdot (1-p_j) \cdot p_{j+1} \cdot \dots \cdot p_n$.
Tento vzorec však nemusíme celý počítať odznova: jeho hodnotu vieme vyjadriť z pravdepodobnosti c tak, že ju vydelíme p_j a následne vynásobíme $1-p_j$.
Analogicky môžeme v konštantnom čase pre každé (k, ℓ) vypočítať pravdepodobnosť toho, že Peťa pokazí práve bránky k a ℓ. Sčítaním všetkých $O(n^2)$ takto vygenerovaných pravdepodobností dostávame odpoveď.

Zlepšenie časovej zložitosti

Vyššie popísané riešenie malo z pochopiteľných dôvodov kvadratickú časovú zložitosť. Hľadanú odpoveď však vieme vypočítať aj v lineárnom čase pomocou trochu šikovnejšej matematiky.
Označme $x_j = (1-p_j)/p_j$. Potom zjavne platí, že $c \cdot x_j$ je pravdepodobnosť prejazdu s chybou práve na bránke x_j a $c \cdot x_k \cdot x_\ell$ pravdepodobnosť prejazdu s chybami práve na bránkach k a ℓ. Celková pravdepodobnosť prejazdu s jednou chybou je teda $cx_1 + cx_2 + \dots + cx_n = c(x_1 + \dots + x_n)$ a s dvoma chybami analogicky: $c(x_1x_2 + x_1x_3 + \dots + x_{n-1}x_n)$.
Označme si teraz $u = x_1 + \dots + x_n$ a $v = x_1x_2 + x_1x_3 + \dots + x_{n-1}x_n$. Ak by sme poznali hodnoty u a v, máme vyhrané: celková pravdepodobnosť, že Peťa vyhrá, je $c + cu + cv$.
Hodnotu u ľahko spočítame v lineárnom čase, ale na priamy výpočet v by sme potrebovali kvadratický čas. Ako to spraviť šikovnejšie? Keď si roznásobíme $u^2 = (x_1 + \dots + x_n)^2$, uvidíme, že výsledok sa dosť podobá na $2v$. Navyše sú tam len členy tvaru x_i^2. Hodnotu $w = x_1^2 + x_2^2 + \dots + x_n^2$ vieme ale tiež spočítať v lineárnom čase, a keď od u^2 odpočítame w, ostane nám $2v$ a vyhrali sme.
Skúste sa zamyslieť, ako by sa toto riešenie dalo zovšeobecniť pre tri, štyri, alebo dokonca všeobecne veľa chýb. Ako bude časová zložitosť závisieť od počtu chýb?

Riešenie pre z ľubovoľných chýb

K osembodovému riešeniu vedie aj ľahšia cesta ako tá z predchádzajúcej časti.
Ak vieme, že všetky s_i sú rovnaké, vieme, že Peťa vyhrá práve vtedy, ak na trati spraví nanajvýš $z = \lfloor c/s_1 \rfloor$ chýb. Potrebujeme teda spočítať celkovú pravdepodobnosť toho, že sa toto stane. Namiesto toho, aby sme si ako v predchádzajúcom riešení museli my ručne odvádzať vzorce, necháme počítač spraviť väčšinu práce za nás.
Keď ešte Peťa stojí na štarte, vieme, že s pravdepodobnosťou 1 nespravila žiadnu chybu.
Po prvej bránke sú dve možnosti: s pravdepodobnosťou p_1 má stále nula chýb, s pravdepodobnosťou $1-p_1$ má prvú chybu. Toto si môžeme predstaviť ako dva nezávislé svety: s pravdepodobnosťou p_1 sa nachádza v prvom z nich, s pravdepodobnosťou $1-p_1$ v druhom.
V každom z týchto svetov Peťa následne prejde druhou bránou. S pravdepodobnosťou p_2 ju prejde čisto, s pravdepodobnosťou $1-p_2$ tam spraví chybu. Každý možný svet sa nám tým opäť rozdelil na dva nové.
Po dvoch bránkach sú teda štyri možné svety: s pravdepodobnosťou p_1p_2 je Peťa vo svete, v ktorom obe bránky prešla čisto, s pravdepodobnosťou $p_1(1-p_2)$ vo svete, kde mala chybu na druhej bránke, s pravdepodobnosťou

$(1-p_1)p_2$ je vo svete, kde mala chybu na prvej bránke, a s pravdepodobnosťou $(1-p_1)(1-p_2)$ je vo svete, kde pokazila obe bránky.
Ak by sme takto pokračovali ďalej, skončili by sme s 2^n svetmi a v podstate by sme spravili tú istú prácu ako pri riešení hrubou silou. Tu ale príde kľúčové pozorovanie: druhý a tretí svet sú rovnocenné. V oboch totiž Peťa spravila práve jednu chybu, a teda je v presne rovnakej situácii: ak v oboch svetoch spraví rovnaký zvyšok jazdy, dostane v cieli rovnaký výsledok.
Môžeme si teda naše pozorovanie preformulovať do nasledovnej omnoho užitočnejšej podoby. Po dvoch bránkach je Peťa v jednom z **troch** možných svetov, a to:

- S pravdepodobnosťou p_1p_2 je vo svete, kde ešte nespravila chybu.
- S pravdepodobnosťou $p_1(1-p_2)+(1-p_1)p_2$ je vo svete, kde spravila práve jednu chybu.
- S pravdepodobnosťou $(1-p_1)(1-p_2)$ je vo svete, kde spravila práve dve chybu.

Pokračovaním v tejto úvahe dostaneme omnoho efektívnejšie riešenie ako hrubou silou – časová zložitosť tohto riešenia bude dokonca polynomiálna od počtu bránok.

Poriadnejšia formulácia riešenia pre z ľubovoľných chýb

Označme si $q_{i,j}$ pravdepodobnosť toho, že pri prejazde prvými i bránkami Peťa spraví presne j chýb.
Na začiatku vieme, že $q_{0,0}=1$ a pre všetky ostatné j platí $q_{0,j}=0$.
Ak teraz poznáme všetky hodnoty $q_{i,\star}$ pre nejaké i, ľahko z nich určíme všetky hodnoty $q_{i+1,\star}$. Stačí zopakovať vyššie popísanú úvahu pre každý z možných svetov. Z tej dostaneme, že pre každé j platí $q_{i+1,j}=q_{i,j}p_{i+1}+q_{i,j-1}(1-p_{i+1})$. Totiž do stavu „po $i+1$ bránkach mám j chýb“ sa Peťa vie dostať dvoma rôznymi spôsobmi: buď mala po i bránkach j chýb (čo nastane s pravdepodobnosťou $q_{i,j}$, ktorú už poznáme) a následne prejde bránku $i+1$ čisto, alebo mala po i bránkach $j-1$ chýb a v nasledujúcej spravila ďalšiu, teda j-tu chybu.
Vyššie popísaným vzorcom každú z hodnôt $q_{i,j}$ spočítame v konštantnom čase. Keď už ich všetky poznáme, celkovú pravdepodobnosť toho, že Peťa vyhrá, vieme vyjadriť ako $q_{n,0}+q_{n,1}+\cdots+q_{n,z}$.
Celková časová zložitosť tohto riešenia je $O(n^2)$, ak spočítame všetky nenulové prvky poľa q. Stačí však počítať hodnoty pre počet chýb nepresahujúci z, čím časovú zložitosť mierne zlepšíme na $O(nz)$.

Úplne všeobecné riešenie

Namiesto počtu chýb môžeme pracovať priamo s Petiným náskokom v stotinách. Budeme si teda (veľmi podobným spôsobom ako vyššie) počítať pravdepobnosti toho, že po i bránkach má Peťa ešte j stotín náskoku.

Listing programu (Python)

```
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] ) )
```

Iná, v podstate symetrická formulácia takéhoto riešenia je predstaviť si rekurzívnu funkciu, ktorá dostane na vstupe parametre i a j a na výstupe vráti odpoveď na otázku, s akou pravdepobnosťou Peťa vyhrá, ak už prešla i bránok a má ešte j stotín náskok. Každú takúto otázku vieme zodpovedať pomocou rekurzie: ak ešte nie sme na konci, vyskúšame obe možnosti pre nasledujúcu bránku a v každej z nich sa následne rekurzívne zavoláme, aby sme zistili odpoveď pre situáciu, do ktorej sme sa práve dostali.

Opäť platí, že keby sme takúto rekurzívnu funkciu implementovali a spustili, dostali by sme exponenciálnu časovú zložitosť – rekurzívna funkcia postupne *backtrackingom* prezrie všetky možné prechody traťou.
Kľúčové je ale pozorovanie, že tento program síce robí exponenciálne veľa volaní funkcie, tie ale nie sú všetky navzájom rôzne – naopak, často sa opakujú a potom zbytočne znova a znova počítame to isté. *Rôznych* relevantných volaní našej funkcie je len $O(nt)$: bránok je n a Petin aktuálny náskok nikdy nenarastie nad t. A keďže výstup našej funkcie je vždy jednoznačne určený jej vstupmi, môžeme použiť *memoizáciu*: akonáhle nejakú hodnotu spočítame, zapamätáme si ju v tabuľke, a vždy, keď ju neskôr znova potrebujeme, namiesto nového pracného výpočtu len rovno použijeme zapamätanú hodnotu.
Aj tento program má celkovú časovú zložitosť $O(nt)$, keďže jediné, čo robí, je priebežne počíta a zapisuje si odpovede na otázky vyššie popísaného typu. Otázok je $O(nt)$ a odpoveď na každú z nich (väčšinou využijúc skôr vypočítané odpovede na iné otázky) zistíme v konštantnom čase.

Listing programu (Python)

```
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 vypocitaj(i, j):
    '''
    vrati pravdepodobnost toho ze Peta vyhra, ak po i brankach ma j stotin naskoku
    dekorator @cache za nas robi memoizaciu
    '''
    if j < 0: return 0.  # uz nemoze vyhrat
    if i == N: return 1. # uz vyhrala
    return P[i] * vypocitaj(i+1, j) + (1-P[i]) * vypocitaj(i+1, j-S[i])

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

Výhodou prvého prístupu (iteratívne dynamické programovanie) je ľahšie šetrenie pamäťou. Pri rekurzívnom riešení si v najhoršom prípade potrebujeme uložiť všetkých $O(nt)$ odpovedí na otázky, zatiaľ čo pri iteratívnom si môžeme všimnúť, že pri výpočte hodnôt $q_{i+1,\star}$ používame vždy len hodnoty $q_{i,\star}$ a teda všetky skôr vypočítané hodnoty q už môžeme zabudnúť. Takto vieme prvému vzorovému riešeniu zlepšiť pamäťovú zložitosť na $O(t)$.
Druhý prístup vie byť v praxi pre niektoré vstupy efektívnejší vďaka tomu, že zatiaľ čo pri iteratívnom postupe musíme vždy vypočítať všetky hodnoty q, lebo nevieme, ktoré budeme neskôr potrebovať, pri rekurzívnom postupe potrebujeme zodpovedať len tie otázky, ktoré naozaj pri niektorom prejazde môžu nastať. Rozmyslite si napr., aké volania funkcie `vypocitaj` môžu nastať, keď rekurzívne vzorové riešenie spustíme na vstupe, v ktorom sú všetky s_i rovné 10.

A-II-2 Jeden reverz

Existuje $O(n^2)$ možností, ktorý úsek reverznúť. Ak prezrieme všetky, môžeme si byť istí, že máme správne riešenie.
Ak budeme prezerať každý úsek zvlášť, budeme potrebovať čas $O(n)$ na každú kontrolu, a tak dostaneme riešenie s celkovou časovou zložitosťou $O(n^3)$, teda kubickou od dĺžky vstupu.

Lepšie riešenie dostaneme, keď si uvedomíme, že je šikovnejšie spracúvať postupne všetky reverzy s rovnakým stredom. Napr. reverz úseku od 8 po 20 je skoro to isté ako reverz úseku od 9 po 19, len ešte navyše vymeníme jeho konce – teda prvky na indexoch 8 a 20.
Prejdeme teda postupne cez všetky možné stredy úseku. Pre každý z nich začneme od najkratšieho úseku a postupne zväčšujeme dĺžku a priebežne si pamätáme, koľko prvkov momentálne sedí na svojich miestach.
(Možných stredov úsekov je $2n-1$: úseky nepárnej dĺžky majú svoj stred na políčku, zatiaľ čo pre úseky párnej dĺžky sa ich stred nachádza medzi dvoma susednými políčkami.)

Takéto riešenie má časovú zložitosť $O(n^2)$, lebo na každý možný úsek poľa sa raz pozrieme a keď sa tak stane, spracujeme ho v konštantnom čase.

Myšlienky vedúce k lepšiemu riešeniu

Ak chceme lepšiu ako kvadratickú časovú zložitosť, nemôžeme si už dovoliť pozerať sa na všetky možné úseky. Budeme teda musieť nejak šikovnejšie vybrať úseky, ktoré má zmysel skúšať.
Začnime tým, že si prvky v poli A rozdelíme na dva typy: tie, ktoré momentálne na správnom mieste sú („dobré“), a tie, ktoré nie sú („zlé“).
Dobré prvky si môžeme len pokaziť: ak reverzneme úsek, všetky dobré prvky v ňom prestanú byť dobré. (S jednou výnimkou: ak reverzneme úsek nepárnej dĺžky a prvok v jeho strede bol dobrý, ostane dobrý.)
Zlé prvky väčšinou zostanú zlé, väčšina reverzov zlý prvok totiž presunie na iné nesprávne miesto.
Pozrime si konkrétny zlý prvok a zamyslime sa nad tým, ako z neho spraviť dobrý. Majme napr. hodnotu 4 na indexe 17. Ktoré reverzy zafungujú pre túto hodnotu? Zjavne práve tie, ktoré vymenia políčko 4 s políčkom 17. A čo majú všetky tieto reverzy spoločné? Zjavne majú ten istý stred: je v strede medzi políčkami 4 a 17.
Pre každý zlý prvok teda zjavne existuje práve jeden stred reverzu, pri ktorom (ak máme reverz dostatočnej dĺžky) sa z tohto zlého prvku stane dobrý. Pri všetkých ostatných reverzoch tento prvok zaručene ostane zlý.

Spracovanie dobrých prvkov

Uvažujme pole B, v ktorom $B[i] = 1$ ak $A[i] = i$ a $B[i] = 0$ inak. Ak chceme zistiť, koľko dobrých prvkov obsahuje konkrétny úsek poľa A, stačí zistiť súčet zodpovedajúceho úseku poľa B.
Aby sme toto vedeli robiť rýchlo, predpočítame si prefixové súčty poľa B (viď napr. https://www.ksp.sk/kucharka/prefixove_sumy/). Vďaka ním potom vieme pre ľubovoľný reverz poľa A v konštantnom čase povedať, koľko z pôvodne dobrých prvkov ostane dobrých aj po tomto reverze.

Nájdenie zaujímavých reverzov

Pre každý zlý prvok sme si už našli jeho jediný správny stred reverzu. Teraz sa na tie isté dáta pozrime z opačnej strany. Zoberme si konkrétny stred reverzu. Preň sme práve zistili, ktoré zlé prvky vie zmeniť na dobré. Tieto prvky odteraz volajme nádejné. Každému nádejnému prvku zodpovedá nejaká minimálna dĺžka reverzu, od ktorej bude upratený na správne miesto.
Predstavme si teraz, že máme pevný stred reverzu a postupne zväčšujeme jeho dĺžku. Ako sa mení upratanosť výsledného poľa? Občas klesne a občas stúpne. Klesnúť môže len vtedy, keď do reverzovaného úseku pribudne nejaký pôvodne dobrý prvok. A podobne stúpnuť môže len vtedy, keď doň pribudne nejaký nádejný prvok.
Ak teda chceme nájsť najlepší reverz s týmto stredom, zjavne sa stačí pozerať práve na tie dĺžky, pre ktoré do reverzovaného úseku práve pribudol nový nádejný prvok. Žiadna iná dĺžka nám nemôže dať lepšie riešenie.
Toto nám dáva takmer kompletný návod, ako spracovať konkrétny stred: Usporiadame si jeho nádejné prvky podľa ich minimálnej dĺžky reverzu, od najkratšej. V tomto poradí potom prejdeme cez všetky tieto zaujímavé dĺžky. Pre každú dĺžku vieme v konštantnom čase povedať, koľko nádejných prvkov dá tento reverz na správne miesto (podľa indexu, na ktorom sme v usporiadanom poli z predchádzajúceho kroku) a taktiež v konštantnom čase vieme povedať, koľko dobrých prvkov dá tento reverz zo správneho miesta preč (pomocou skôr predpočítaných prefixových súčtov).
Keďže každý zlý prvok je nádejný len pre jeden stred, dokopy toto riešenie spracuje každý zlý prvok len raz. Celková časová zložitosť je teda lineárna, až na jeden detail: logaritmus navyše kvôli potrebe usporadúvať zoznamy nádejných prvkov. Máme teda riešenie v čase $O(n \log n)$.

Usporiadame šikovnejšie

Logaritmu sa zbavíme tak, že si uvedomíme, že kľúče, podľa ktorých prvky usporadúvame (t.j. minimálna dĺžka reverzu, ktorý ho dostane na správne miesto) sú všetko celé čísla z rozsahu od 1 po n. Môžeme teda na triedenie použiť countsort.
Chce to ale ešte trochu dohliadnuť na technické detaily. Nemôžeme si dovoliť robiť pre každý stred zvlášť jeden countsort, lebo pri každom z nich by sme strávili $O(n)$ času inicializáciou počítadiel na nuly. Namiesto toho to

spravíme nasledovne:
Na začiatku prejdeme celé pole A a pre úplne každý zlý prvok si nájdeme jeho minimálnu dĺžku reverzu. Teraz jedným countsortom usporiadame všetky zlé prvky naraz podľa tejto dĺžky. No a keď už ich máme takto usporiadané, tak ich raz prejdeme a prerozdelíme k jednotlivým stredom. Pri každom strede tak dostaneme správne usporiadaný zoznam jeho nádejných prvkov.
(Na toto prerozdelenie sa tiež môžeme dívať ako na druhé kolo countsortu, pri ktorom teraz všetky zlé prvky preusporiadame podľa súradnice ich správneho stredu reverzu, pričom v prípade rovnosti zachovávame aktuálne poradie.)
Ak máme pre nejaký stred dva nádejné prvky s rovnakou dĺžkou (pri reverze si vymenia miesta a oba budú dobré), netreba to nijak špeciálne ošetrovať. V nižšie uvedenom programe príslušnú dĺžku reverzu proste spracujeme postupne dvakrát – raz pre každý prvok. Pri druhom spracovaní dostaneme správnu hodnotu upratanosti pre tento reverz. (Pri prvom o jedno menšiu, čo vôbec nevadí.)

Listing programu (C++)

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


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

    // predpocitame B a jeho prefixove sucty
    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 );

    // usporiadame vsetky zle prvky podla stredu a v ramci stredu podla dlzky reverzu
    vector< vector<int> > podla_dlzky(N);
    for (int n=0; n<N; ++n) if (A[n] != n) podla_dlzky[ abs(A[n]-n) ].push_back(n);
    vector< vector<int> > podla_stredu(2*N-1);
    for (auto row : podla_dlzky) for (int x : row) podla_stredu[ x+A[x] ].push_back(x);

    // prejdeme vsetkych kandidatov na najlepsie riesenie
    int bestupr = SB[N], bestzac = 0, bestkon = 0;
    for (int stred=0; stred<2*N-1; ++stred) {
        for (int i=0; i<int(podla_stredu[stred].size()); ++i) {
            int x = podla_stredu[stred][i];
            int dlzka = abs( A[x] - x );
            int zac = (stred-dlzka)/2, kon = (stred+dlzka)/2;
            // prezerame usek od zac po kon
            int upr = SB[N]; // tolkoto bola upratanost pred reverzom
            upr += (i+1); // tolkoto nadejnych prvkov sme reverzom zmenili na dobre
            upr -= SB[kon+1] - SB[zac]; // tolkoto dobrych prvkov sme reverzli
            if (stred%2 == 0 && A[stred/2] == stred/2) ++upr; // dobry prvok v strede reverzu zostal dobry
            if (upr > bestupr) {
                // nasli sme nove optimum
                bestupr = upr; bestzac = zac; bestkon = kon;
            }
        }
    }
    cout << bestzac << " " << bestkon << endl;
}
```

A-II-3 Ťažký robot skáče

Kvôli lepšej čitateľnosti budeme pri odhadoch časovej zložitosti predpokladať, že bludisko je štvorcového tvaru rozmerov $n \times n$. Všetky algoritmy však samozrejme fungujú aj pre obdĺžnikové vstupy a čitateľ si isto ľahko doplní presnejšie odhady, ak ich bude potrebovať.

Cesta po akciách

Ak vieme, akú akciu robot spravil ako poslednú, vieme jednoznačne povedať, aké akcie môže robiť teraz. Ak bol poslednou akciou krok, môžeme kráčať alebo skákať ľubovoľným smerom, ak ňou ale bol skok, môžeme kráčať aj skákať len tým istým smerom.
Každú akciu môžeme popísať napr. políčkom, kde začína a druhým políčkom (v tom istom riadku alebo stĺpci), kde končí. Iný, ekvivalentný popis akcie pozostáva z políčka kde začína, smeru ktorým vedie a dĺžky pohybu. Dokopy zjavne existuje $O(n^3)$ možných akcií.

Predstavme si teraz graf, ktorého vrcholmi sú všetky možné akcie a ktorého (orientované) hrany hovoria, že bezprostredne po akcii x môžeme spraviť akciu y. Pre každú konkrétnu akciu existuje len $O(n)$ možností ako vyzerá nasledujúca: políčko, kde začína, je pevné, zvoliť si môžeme len dĺžku a možno smer. Dokopy má preto tento graf $O(n^4)$ hrán.
Pomocou prehľadávania zo šírky môžeme v tomto grafe nájsť najkratšiu cestu začínajúcu ľubovoľnou akciou vedúcou z ľavého horného rohu a končiacu krokom do pravého dolného rohu. Táto najkratšia cesta grafom zjavne priamo zodpovedá najkratšej možnej postupnosti akcií riešiacej našu úlohu.
Máme teda prvé korektné riešenie. Jeho časová zložitosť je $O(n^4)$.

Šikovnejšie skáčeme

Lepšie riešenie začneme tým, že si uvedomíme, že sa nám nikdy neoplatí spraviť dva skoky po sebe.
Každá postupnosť po sebe idúcich skokov musí celá viesť jedným smerom a končiť krokom tým istým smerom. V optimálnom riešení sa zjavne nikdy neoplatí robiť po sebe viac skokov, lebo všetky po sebe idúce skoky tým istým smerom vieme jednoducho nahradiť jedným veľkým skokom. Napríklad namiesto dvoch po sebe idúcich skokov dĺžky 7 a 3 stačí spraviť jeden skok dĺžky 10 a ušetriť tak akciu.
Naša úloha je teda ekvivalentná s nasledovnou: robotovi vieme zadať ľubovoľnú postupnosť príkazov „sprav krok“ (1 akcia) a „sprav skok a za ním krok tým istým smerom“ (2 akcie).
Uvažujme teraz graf, ktorého vrcholmi sú prázdne políčka pôvodnej mapy. Z každého vrcholu vedú ≤ 4 hrany dĺžky 1 zodpovedajúce platným krokom a $O(n)$ hrán dĺžky 2 zodpovedajúcich platným kombináciám skok+krok. Aj v tomto grafe platí, že najkratšia cesta (presnejšie, tentokrát priamo najkratšia cesta z políčka vľavo hore na políčko vpravo dole) zodpovedá najkratšej postupnosti akcií, ktorú hľadáme.
No a aj v tomto grafe vieme najkratšiu cestu nájsť prehľadávaním do šírky. (Buď ho upravíme tak, aby vedelo pracovať aj s hranami dĺžky 2, alebo si ešte pred spustením prehľadávania na každú hranu dĺžky 2 pridáme do stredu nový vrchol, ktorý ju rozdelí na dve hrany dĺžky 1.)
Časová zložitosť tohto riešenia je $O(n^3)$.

Šikovnejšie spracúvame skoky

Predchádzajúce riešenie ešte robí veľa práce zbytočne dvakrát. Ak sme už vyskúšali všetky možnosti, ako skočiť doprava z políčka $(4, 7)$, a potom neskôr spracúvame jeho pravého suseda – políčko $(4, 8)$ – príliš nového toho skákaním doprava už nedosiahneme. Ako ale zabrániť prezeraniu toho istého dvakrát?
Postavme si pre našu úlohu ešte jeden nový graf. Tentokrát budeme mať vrcholy dvoch typov:

- Jeden vrchol pre každý stav „stojím na políčku (x, y)“.
- Jeden vrchol pre každý stav „som vo vzduchu nad políčkom (x, y) a skáčem smerom d“.

Stavov prvého typu je nanajvýš n^2, stavov druhého typu je nanajvýš $4n^2$, dokopy teda má náš graf $O(n^2)$ vrcholov.
Hrany budú opäť orientované a budú viesť do stavov, ktoré môžu bezprostredne nasledovať. Hrany teda budú vyzerať nasledovne:

- Z každého stavu, v ktorom stojíme, budeme mať nanajvýš štyri hrany zodpovedajúce krokom a tiež štyri hrany zodpovedajúce začiatku skoku každým možným smerom.
- Z každého stavu, v ktorom letíme vzduchom, budeme mať hranu zodpovedajúcu letu o políčko ďalej.
- Ak sme vo vzduchu a aj naše políčko aj to bezprostredne nasledujúce naším smerom sú voľné, budeme mať hranu do stavu, v ktorom stojíme na tom nasledujúcom políčku.

Hrany zodpovedajúce tomu, že ďalej letíme vzduchom, budú mať dĺžku 0. Ostatné hrany budú mať dĺžku 1.
Ľahko overíme, že ľubovoľný krok zodpovedá prechodu po hrane dĺžky 1 a pre ľubovoľný skok+krok prejdeme hranami s dĺžkou 1, 0, 0, ..., 0, 1, a teda dokopy nejakú cestu dĺžky 2. Vzdialenosti medzi stavmi, v ktorých niekde stojíme, teda vždy zodpovedajú počtu akcií, ktoré robot na príslušnú zmenu stavu potrebuje.
Z každého vrcholu vedie len konštantne veľa hrán (nanajvýš 8), dokopy má teda náš graf nielen $O(n^2)$ vrcholov ale aj $O(n^2)$ hrán. No a najkratšiu cestu ním opäť vieme nájsť prehľadávaním do šírky. Tentokrát ho len potrebujeme upraviť tak, aby vedelo pracovať aj s hranami dĺžky 0.

0-1 prehľadávanie do šírky

V klasickom prehľadávaní do šírky graf postupne spracúvame akoby po vrstvách: najskôr začiatok, potom vrcholy vo vzdialenosti 1, potom tie vo vzdialenosti 2, a tak ďalej. Na to používame frontu, v ktorej čakajú vrcholy na spracovanie. V každom okamihu platí, že frontu vieme rozdeliť na dve (možno prázdne) časti. V prvej sú ešte nespracované vrcholy z aktuálne spracúvanej vrstvy. Nech d je vzdialenosť, ktorú majú všetky tieto vrcholy od štartu. V druhej časti fronty potom na spracovanie čakajú už objavené vrcholy z nasledujúcej vrstvy. Tieto majú všetky od štartu vzdialenosť $d + 1$.
Ak máme v grafe aj hrany dĺžky 0 a nejakou takouto hranou objavíme nový vrchol v, vieme, že má rovnakú vzdialenosť d ako ten vrchol, ktorý práve spracúvame, a teda patrí ešte do aktuálnej vrstvy. Ako zabezpečiť, aby sme v spracovali ešte s ostatnými vrcholmi v aktuálnej vrstve? Ľahko: stačí, keď ho zaradíme **na začiatok** fronty, nie na jej koniec.
Detaily implementácie: Niektoré implementácie fronty síce nepodporujú vkladanie na začiatku (len výber na začiatku a vkladanie na konci), ale existuje viacero všeobecnejších dátových štruktúr, ktoré túto možnosť majú. Dá sa napr. vhodne použiť spájaný zoznam, ale aj v poli vieme šikovne implementovať tzv. obojsmernú frontu (deque), pri ktorej vieme na oboch koncoch aj vkladať aj vyberať prvky. Detailný popis deque nájdete napr. vo vzorových riešeniach úlohy OI39-B-I-2.
Vo všeobecnosti sa nám pri „0-1 prehľadávaní do šírky“ môže stať, že konkrétny vrchol v zaradíme do fronty na spracovanie až dvakrát. Toto nastane vtedy, ak ho počas spracúvania vrstvy vo vzdialenosti d najskôr objavíme hranou dĺžky 1, takže mu dáme vzdialenosť $d + 1$ a zaradíme ho na koniec fronty, ale následne doň neskôr prídeme z tej istej vrstvy aj hranou dĺžky 0. Vtedy mu jednoducho znížime vzdialenosť na d a zaradíme ho aj na začiatok fronty. Nič zlé sa nestane, keď ho neskôr postupne spracujeme dvakrát – druhé spracovanie už nič nové neobjaví a asymptotickú časovú zložitosť nám to nepokazí. (Druhou možnosťou je pamätať si explicitne o každom vrchole, či už bol spracovaný, a každý vrchol explicitne spracovať len raz.)
Toto riešenie má optimálnu časovú zložitosť $O(n^2)$, resp. $O(rs)$ pre obdĺžnikové vstupy.

Listing programu (C++)

```
#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; };

// smer 0 oznacuje ze stojime, smery 1-4 su svetove strany
const int DR[] = {0, -1, 0, 1, 0};
const int DS[] = {0, 0, 1, 0, -1};

pole3d vzdialenost;
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 (vzdialenost[nr][ns][nd] <= nova_vzd) return; // nic nove sme nenasli
    vzdialenost[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];

    vzdialenost.resize(R, pole2d(S, pole1d(5, 987654321)));
    vzdialenost[0][0][0] = 0;
    fronta.push_back( {0,0,0} );
    while (!fronta.empty()) {
        stav akt = fronta.front();
        fronta.pop_front();
        int akt_vzd = vzdialenost[ akt.r ][ akt.s ][ akt.d ];
        if (akt.d == 0) {
            // sme na zemi, mozeme spravit krok alebo zacat skok do lubovolnej svetovej strany
            for (int nd=1; nd<=4; ++nd) {
                int nr = akt.r + DR[nd], ns = akt.s + DS[nd];
                if (!(0 <= nr && nr < R && 0 <= ns && ns < S)) continue; // vysli sme mimo mapy
                if (plan[nr][ns] == '.') zarad_do_fronty(akt_vzd, 1, nr, ns, 0); // ak je volno, mozeme krok
                zarad_do_fronty(akt_vzd, 1, nr, ns, nd); // vzdy mozeme zacat skok
            }
```

```
        } else {
            // sme vo vzduchu, mozeme letiet dalej alebo pristavat
            int nr = akt.r + DR[akt.d], ns = akt.s + DS[akt.d];
            if (0 <= nr && nr < R && 0 <= ns && ns < S) { // ak nejdeme vyletiet mimo mapy
                zarad_do_fronty(akt_vzd, 0, nr, ns, akt.d); // vzdy mozeme letiet dalej
                // a ak je pod aj pred nami volno, mozeme aj pristat
                if (plan[akt.r][akt.s] == '.' && plan[nr][ns] == '.') zarad_do_fronty(akt_vzd, 1, nr, ns, 0);
            }
        }
    }
    int odpoved = vzdialenost[R-1][S-1][0];
    cout << (odpoved == 987654321 ? -1 : odpoved) << endl;
}
```

A-II-4 O Vekslákbotovi a Pokladničke

Podúloha A (2 body): prefarbi na maximum

Prvá inštrukcia zoberie tri žetóny navzájom rôznych farieb (červený, zelený a modrý) a nahradí ich troma bielymi. Keď sa už táto inštrukcia nedá použiť, vieme, že nám ostali práve dve farby (lebo počty žetónov na začiatku boli rôzne).
Ďalej si teda pridáme inštrukcie „prefarbujúce“ ľubovoľné dva rôzne žetóny na biele. V každej situácii sa bude dať opakovane použiť práve jedna z týchto inštrukcií.
Keď sa už to nevieme spraviť, vieme, že nám ostala práve jedna farba: tá, ktorej bolo na začiatku najviac. Na tú prefarbíme všetky biele žetóny.

Listing programu

```
cervena, zelena, modra -> 3 biela

cervena, zelena -> 2 biela
cervena, modra -> 2 biela
zelena, modra -> 2 biela

cervena, biela -> 2 cervena
zelena, biela -> 2 zelena
modra, biela -> 2 modra
```

Dodáme ešte, že je zjavné, že v situácii, kde sú všetky žetóny tej istej farby, už žiadnu inštrukciu tohto programu nevieme použiť, a teda program zastane.

Podúloha B (3 body): prefarbi na minimum

Rovnako ako v podúlohe A začneme tým, že kým máme všetky tri farby, zoberieme po jednom žetóne z každej a nahradíme ich troma bielymi.
Akonáhle nám ostanú len dve farby, vieme, ktorej bolo najmenej – tej, ktorú už nemáme. Ak nám napr. ešte ostali červené a zelené žetóny, vieme, že na konci chceme mať všetko modré. Pridáme si preto pre modrú inštrukciu „červená, zelená → 2 tmavomodré“ a analogické dve pre ostatné farby.
Akonáhle máme nejaké tmavé žetóny, môžeme všetko ostatné prefarbiť na príslušnú tmavú farbu. A akonáhle máme všetky žetóny jednej tmavej farby, môžeme ju prefarbiť na pôvodnú svetlú a skončiť.

Listing programu

```
cervena, zelena, modra -> 3 biela

zelena, modra -> 2 tmavocervena
cervena, modra -> 2 tmavozelena
cervena, zelena -> 2 tmavomodra

tmavocervena, zelena -> 2 tmavocervena
tmavocervena, modra -> 2 tmavocervena
tmavocervena, biela -> 2 tmavocervena

tmavozelena, modra -> 2 tmavozelena
tmavozelena, cervena -> 2 tmavozelena
tmavozelena, biela -> 2 tmavozelena
```

```
tmavomodra, zelena -> 2 tmavomodra
tmavomodra, cervena -> 2 tmavomodra
tmavomodra, biela -> 2 tmavomodra

tmavocervena -> cervena
tmavozelena -> zelena
tmavomodra -> modra
```

Podúloha C (5 bodov): mocnina troch

Vyrobíme si jeden modrý žetón, čiže 3^0 modrých žetónov. Potom postupne c-krát strojnásobíme počet modrých žetónov. Presnejšie, v každej iterácii tohto cyklu odstránime jeden červený žetón, potom zmeníme každý modrý na tri fialové a potom každý fialový naspäť na modrý.
Keď sa nám červené žetóny minú, vieme, že máme 3^c modrých žetónov. Na záver už len tie premeníme na červené a skončíme.
Ako vieme ale zabezpečiť, aby sa nám nepomiešalo, ktoré inštrukcie kedy použiť? Obzvlášť nie vtedy, keď na konci opäť vzniknú červené žetóny?
Na to použijeme žetóny rôznych ďalších farieb, ktoré budú predstavovať rôzne fázy, v ktorých sa práve náš program nachádza. Do programu pridáme obmedzenie hovoriace, že zo všetkých týchto farieb dokopy nikdy nesmie naraz existovať viac ako jeden žetón.

Listing programu

```
OBMEDZENIE: kontroluj + nasob + upratuj + skonci <= 1

-> kontroluj, modra

kontroluj, cervena -> nasob
kontroluj -> skonci

nasob, modra -> nasob, 3 fialova
nasob -> upratuj

upratuj, fialova -> upratuj, modra
upratuj -> kontroluj

skonci, modra -> skonci, cervena
```

TRIDSIATY DEVIATY ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2024