

### Task A11. Tunnels

 1.25 s  1024 MB

Iepuraşul a săpat un sistem de tuneluri, care are  $N$  niveluri (numerotate de sus în jos: de la 0 la  $N - 1$ ). Fiecare nivel are  $M$  camere (numerotate de la stânga la dreapta: de la 0 la  $M - 1$ ). Putem considera camerele ca fiind celulele unei matrice de  $N$  linii şi  $M$  coloane. Există tuneluri între toate perechile de camere adiacente (care au o latură comună). Sub toate acestea, se află o cameră mare a comorii (care acoperă efectiv întreg nivelul  $N$ ). Cu toate acestea, unele dintre cele  $N \times M$  camere s-au prăbuşit şi sunt astfel blocate (la fel şi toate tunelurile lor).

Iepuraşul vrea să-şi protejeze comoara aşa că a instalat capcane în aproape toate tunelurile verticale. Mai precis, fiecare nivel (inclusiv nivelul camerei comorii) are exact un tunel vertical de intrare sigur (fără capcana). În plus, este garantat că există un drum sigur (un drum care nu trece prin tuneluri cu capcane) de la suprafaţă până la camera comorii.

Alice doreşte să ajungă în camera comorii Iepuraşului, dar să intre în sistemul de tuneluri fără să ştie nimic ar fi extrem de periculos. Mai întâi, ea a studiat camerele folosind sonarul şi a aflat care dintre ele sunt blocate. Cu toate acestea, ea încă nu ştie care tuneluri sunt cu capcane şi care sunt sigure. Apoi, a început să-l observe pe Iepuraş. Ea ştie că el este mereu într-o grabă şi că alege întotdeauna cel mai scurt drum sigur (fără tuneluri cu capcane) de la suprafaţă la comoară (şi invers), adică drumul unic sigur care nu trece de două ori prin aceeaşi cameră. De asemenea, ea l-a văzut pe Iepuraş intrând în sistemul de tuneluri (nivelul 0) de la suprafaţă deasupra camerei  $K$  ( $0 \leq K < M$ ). Acest lucru îi permite ei să intre în siguranţă în sistemul de tuneluri. În cele din urmă, Alice a adus un câine care poate mirosi urmele lăsate de Iepuraşului. Cu ajutorul câinelui, ea poate merge într-o cameră accesibilă (pe nivelul ei curent) şi să o investigheze – să verifice dacă Iepuraşul a trecut prin ea sau nu. După o serie de astfel de investigaţii, când este sigură, ea poate merge într-o cameră accesibilă şi să coboare mai adânc (la nivelul următor) de acolo. Ea nu vrea să-şi rişte viaţa, aşa că trebuie să fie absolut sigură că tunelul vertical ales este sigur. Ea vrea să ajungă la comoară în siguranţă, folosind cât mai puţine investigaţii posibile în cel mai rău caz (şi ea este grăbită deoarece trebuie să participe la o petrecere cu ceai).

Formal, definim cel mai rău caz legat de numărul de investigaţii al unei strategii de explorare pentru un test anume ( $N$ ,  $M$ ,  $K$  şi un set de camere blocate) astfel: Enumeră toate seturile posibile de tuneluri sigure, determină drumul iepuraşului pentru fiecare, şi aplica strategia pe fiecare caz (independent). În cadrul strategiei, cazul cel mai rău este numărul maxim de investigaţii făcute pe oricare dintre cazuri (asumând ca este valid şi reuşeşte să obţină în siguranţa comoara în toate cazurile; în caz contrar, spunem că cel mai rău caz legat de numărul de investigaţii este infinit). Apoi enumerăm toate strategiile posibile şi găsim cel mai rău caz legat de numărul de investigaţii minim. Acesta este numărul maxim de investigaţii alocate pentru acel test.

Alice este o fată deşteaptă, dar sistemul de tuneluri al Iepuraşului este vast, aşa că nici măcar ea nu-şi poate da seama de o strategie optimă de explorare. Ajutaţi-o prin scrierea unui program care explorează sistemul de tuneluri şi găseşte camera comorii fără să folosească mai multe investigaţii decât numărul alocat testului, fără să se plimbe printre camerele blocate sau să treacă prin tuneluri cu capcane. (Programul tău trebuie

să îndeplinească toate aceste condiții pentru a trece corect un test anume.)

### Detalii de implementare

Ai de implementat funcția `solve`:

```
void solve(int n, int m, int k, const std::vector<std::vector<bool>>& blocked)
```

Va fi apelată o singură dată per test cu  $N$ ,  $M$  și  $K$ , dar și o descriere a camerelor (indexate sub forma `blocked[level][position]`). Trebuie să faci explorarea tunelelor (începând cu nivelul 0, camera  $K$ ) întrucât să ajungă nivelul camerei cu comoara (nivelul  $N$ ) fără a folosi mai multe investigații față de cel mai rău caz posibil legat de numărul de investigații pentru acel test. De la această funcție (și alte funcții pe care le scrii), poți apela funcțiile investigate și goDeeper.

```
bool investigate(int s)
```

Apelarea funcției `investigate` reprezintă deplasarea către camera  $S$  de pe nivelul curent (trebuie să se poată ajunge în ea, de exemplu, nu trebuie să fie nicio cameră blocată în drum spre) și verifică dacă lepurășul trece prin această cameră în drumul lui spre comoară. Funcția returnează `true` dacă da, și `false` în caz contrar.

```
bool goDeeper(int s)
```

Apelarea funcției `goDeeper` reprezintă deplasarea către camera  $S$  de pe nivelul curent (trebuie să se poată ajunge în ea) și apoi coborârea spre următorul nivel folosind tunelul vertical. Tunelul trebuie să fie sigur (fără capcane) iar camera de dedesubt nu trebuie să fie blocată. După aceea, poți continua explorarea din camera  $S$  de pe nivelul următor (cu excepția cazului în care atingi nivelul  $N$ , caz în care funcția ta `solve` ar trebui să se oprească).

Codul tău va fi compilat împreună cu un grader, deci nu ar trebui să conțină o funcție `main`, respectiv să citească din `stdin` sau scrie în `stdout`. Va trebui de asemenea să includeți fișierul header `tunnels.h`.

Graderul poate fi în unele cazuri adaptiv (dar în continuare deterministic), de exemplu, poate decide ce va returna funcția `investigate` sau dacă `goDeeper` are succes, în funcție de acțiunile precedente ale programului tău, fără să aibă un set fixat de tuneluri sigure. Însă, este garantat că orice va face grader-ul este valid în cadrul unui set anume posibil de tunele sigure. Graderul poate opri execuția, dacă apelezi `goDeeper` și încerci să treci printr-un tunel cu capcane, sau dacă graderul decide dacă nu mai este posibil pentru tine să rezolvi testul încadrându-te în numărul alocat de investigații. În plus, timpul de execuție al graderului nu este considerat în cadrul limitei de timp.

### Testare locală

Pentru a testa programul local, un evaluator local și un fișier header sunt prevăzute. Evaluatorul local nu este adaptiv și nu efectuează calcule sau verificări. Citește  $N$ ,  $M$ ,  $K$  și o matrice de 0 și 1 (fără spații), apelează `solve` și afișează oricând programul apelează `investigate` sau `goDeeper` (așteaptă input pentru fiecare `investigate` - un 0 sau un 1 pe care îl va returna). Sunteți liberi să modificați evaluatorul local.

### Restricții

- $1 \leq N, M \leq 5000$

### Subtaskuri

Subtaskuri	Punctaj	Restricții
1	5	$M = 2$
2	16	Nicio cameră nu este blocată.
3	19	$N, M \leq 100$
4	19	$N, M \leq 400$
5	20	$N, M \leq 1000$
6	21	Nicio restricție.

Se acordă punctele pentru un subtask, doar dacă soluția trece toate testele incluse în el și în toate subtaskurile incluse în el.

### Exemplu

Intrare	Interacțiune
2 3 1 001 100	solve(2, 3, 1, {{0, 0, 1}}, {1, 0, 0}) goDeeper(1) investigate(2): return true goDeeper(2)

Numărul minim de investigații în cel mai defavorabil caz pentru acest test este 1, deci soluția ar trece testul.