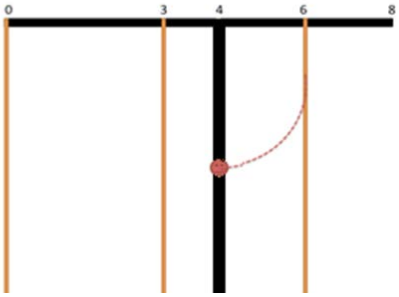
CIRCUS

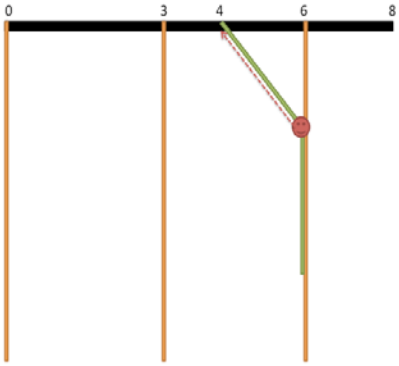
Novi cirkuski performans uključuje korišćenje jako dugačkih kanapa koji vise sa plafona cirkuske hale. Ulaz u cirkusku halu i pozicije tačaka vešanja svih kanapa nalaze se na jednoj pravoj liniji. Plafon je visok 10^{18} metara i svi kanapi su dovoljno dugački da dodiruju pod. Klovn-majmunče Marko mora skakati sa jednog kanapa na drugi i cilj mu je da se udalji bar M metara od ulaza.

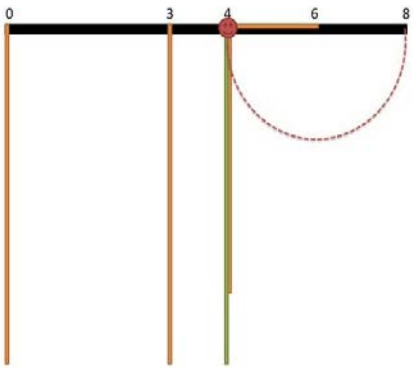
Ukupno ima N kanapa. i-ti kanap visi sa lokacije koja je udaljena P_i metara od ulaza, mereno duž plafona.

Klovn Marko je jako pažljiv i ne skače bezveze sa jednog kanapa na drugi. Zamislimo da se Marko drži za i-ti kanap na rastojanju (visini) S od plafona. Marko se može zaljuljati na kanapu koji drži. Ukoliko prilikom ljuljanja na kanapu Marko dosegne tačku koja je na rastojanju ne manjem od M metara od ulaza, smatramo da je njegov zadatak ispunjen. Prilikom ljuljanja, Marko može zgrabiti bilo koji drugi kanap koji visi sa lokacije koja je udaljena najviše S metara od trenutnog kanapa. **Formalnije, Marko može da zgrabi j-ti kanap ako je $|P_i - P_j| \leq S$**. U tom trenutku (kada zgrabi j-ti kanap), Marko drži i i-ti i j-ti kanap. Tada on počinje da se penje duž i-tog kanapa, držeći sve vreme j-ti kanap. Kada dostigne tačku u kojoj i-ti kanap dodiruje plafon, on privuče j-ti kanap tako da bude potpuno zategnut duž plafona. Tek onda, Marko pušta i-ti kanap i prelazi na j-ti kanap i dolazi u situaciju da se nalazi na rastojanju od plafona koje je jednako dužini dela j-tog kanapa koji je malopre privukao sebi. **Formalnije, sada se Marko nalazi na j-tom konopcu na udaljenosti $|P_i - P_j|$ od plafona i može nastaviti sa kretanjem**.

Direktor cirkusa želi da doda dodatni (privremeni) kanap i okači ga na plafon na rastojanju *D* metara od ulaza, mereno duž plafona. Na ovom kanapu će performans početi. Markov zadatak je i dalje da od ovog kanapa dođe do pozicije koja je udaljena bar *M* metara od ulaza. Prilikom prelaska sa kanapa na kanap, ovaj privremeni kanap se ne razlikuje od običnih (tj. važe ista pravila). **Vaš program mora da odgovora na nekoliko pitanja: koje je najmanje rastojanje od plafona (visina) na koju Marko mora da se drži za ovaj privremeni kanap na početku performansa tako da može ispuniti svoj zadatak.**







Razmotrimo primer sa 3 kanapa koja se nalaze na pozicijama 0, 3 i 6 metara od ulaza. Neka je cilj dostići *M* = 8.

Neka je dat privremeni kanap na rastojanju 4 metra od ulaza. Pretpostavimo da Marko drži privremeni kanap (**mastan** na slici) na rastojanju 3 metra od plafona. Marko se može zaljuljati i uhvatiti za kanap koji je udaljen 6 metara od ulaza.

Čim Marko zgrabi kanap koji je udaljen 6 metara od ulaza, kreće da se penje uz trenutni kanap (u ovom slučaju – onaj privremeni)

Kada se popne do plafona, on drži drugi kanap na dužini 6 - 4 = 2 metara od njegove tačke vešanja. Sada Marko može da pređe na drugi kanap, zaljulja se i dosegne traženu daljinu od 8 metara od ulaza.

Zadatak

Potrebno je implementirati dve funkcije: `init`, koja se poziva samo jednom na početku vašeg programa i koja obrađuje početne ulazne parametre, i `minLength`, koja odgovara na gore opisani upit za datu lokaciju privremenog kanapa. Vaša funkcija `minLength` će biti pozivana nekoliko puta u toku izvršenja programa.

- `init(N, M, P[])`
  - N: broj kanapa
  - M: traženo rastojanje od ulaza (u metrima)
  - P[]: niz dužine *N* u kome se nalaze lokacije svih kanapa, merene u metrima duž plafona (udaljenosti od ulaza). **Niz je indeksiran od 0**.
- `minLength(D)`
  - D: rastojanje (u metrima) od ulaza, mereno duž plafona, na kome će se nalaziti privremeni kanap u trenutnom upitu.

Detalji implementacije

Potrebno je da pošaljete tačno jedan fajl koji se mora zvati `circus.cpp`. Ovaj fajl mora implementirati `init` i `minLength` funkcije kao što je gore opisano, koristeći sledeće potpise:

```
void init(int N, int M, int P[])
int minLength(int D)
```

Fajl **circus.cpp** NE SME sadržati funkciju *main()*, ali može sadržati dodatne promenljive i funkcije ukoliko su potrebne za funkcije *init* i *minLength.* Vaš program mora sadržati liniju `#include "circus.h"` na početku.

Ograničenja

$1 \le M \le 10^9$

$0 \le P_i, D < M$

Primer

$N = 3$, $M = 8$, $P = \{0, 3, 6\}$

Poziv funkcije `minLength(4)` treba da vrati 2. Moguće je preći sa privremenog kanapa na kanap na poziciji 6, držeći se za njega na rastojanju 2 metra od plafona a zatim ljuljanjem dostići rastojanje $M = 8$. Još jedan mogući način prelaska sa kanapa na kanap je preći sa privremenog kanapa na kanap na poziciji 0, zatim na kanap na poziciji 3, zatim na 6 a zatim dostići $M = 8$. Međutim, za ovaj niz poteza je neophodno da se na početku klovn-majmunče Marko drži za privremeni kanap na visini 4 metra od plafona.

Poziv funkcije `minLength(5)` treba da vrati 3. Primetimo da Marko može dostići traženo rastojanje M odmah sa trenutnog kanapa.

Subtasks

subtask	Poeni	N	Broj poziva funkcije minLength	komentar
1	11	$0 \le N \le 100\,000$	**1**	Samo jedan poziv funkciji minLength sa parametrom D = 0
2	17	$0 \le N \le 100$	**100**	$M \le 2000$
3	23	$0 \le N \le 2\,000$	**1 000**	
4	9	$0 \le N \le 2\,000$	**1 000 000**	
5	31	$0 \le N \le 100\,000$	**1 000**	
6	9	$0 \le N \le 100\,000$	**1 000 000**	

Lokalno testiranje

Da biste mogli da testirate vaše funkcije *init* i *minLength* na vašem lokalnom računaru, dobićete fajlove *Lgrader.cpp* i *circus.h*. Kompajlirajte ih zajedno sa vašim fajlom **circus.cpp** i dobićete program koji možete koristiti za testiranje vaših funkcija.

Lgrader čita ulaz u sledećem formatu

- linija 1: dva cela broja N i M
- linije $\mathbf{2 + i}$ ($0 \le i \le N$-1): P_i
- linija $N + 2$: Q – broj poziva funkciji `minLength`
- linije $\mathbf{N + 3 + i}$ ($0 \le i \le Q$-1): brojeve D – parametre za svaki poziv funkciji *`minLength`*

Lgrader će odštampati Q brojeva, po jedan u svakoj liniji, koji predstavljaju povratnu vrednost funkcije *`minLength`* za odgovarajuće pozive.

Primer lokalnog testiranja

Input	Output
3 8 0 3 6 2 4 5	2 3

HAPPINESS

Novčani sistem u *X*-zemlji je malo čudan – postoje novčanice sa vrednošću svakog celog broja od 1 do *M,* uključivo. Postoji još jedno čudno pravilo u prodavnicama ove zemlje – kupac nikada ne sme dobiti kusur niti ostaviti bakšiš – drugim rečima, kupac uvek mora platiti tačan iznos onoga što kupuje. Ako to ne može uraditi, nije mu dozvoljeno da kupi.

Didi je dečak iz X-zemlje. Kao i ostali, on se stalno bori protiv gore pomenutih pravila. On uvek zna skup novčanica koje poseduje – pretpostavimo da su njihove vrednosti $a_1, a_2,\ldots,a_N$. Sve te vrednosti su između 1 and *M* i on može posedovati više novčanica iste vrednosti. Takođe, niz $a_1, a_2,\ldots\ldots,a_N$, nije uređen ni na koji način. Didi se oseća srećno kada može da kupi bilo koju robu čija je cena jednaka bilo kom broju između 1 i sume svih njegovih novčanica $a_1 + a_2 + ... + a_N$, uključivo. Naravno, on želi da bude u mogućnosti da odmah odredi da li datu robu može kupiti ili ne.

Napomena: Sortirajmo niz $a_1, a_2,\ldots\ldots,a_N$ u neopadajući poredak. Neka je $S_i = 1+a_1+ a_2+\ldots\ldots+a_i$. Potreban i dovoljan uslov za predstavljanje svakog broja između 1 i $a_1+ a_2+\ldots\ldots+a_N$ kao zbir nekoliko novčanica iz multiskupa $a_1, a_2,\ldots\ldots,a_N$, je da za svako i>1 važi $S_i \geq a_{i+1}$ *i da važi* $a_1 = 1$.

Naravno, Didijev skup novčanica se menja posle svake kupovine, ali takođe i posle svake plate koju dobije – zato je njegova sreća promenljiva. Pomozite Didiju da odredi kada je zapravo srećan. Vaš program će na početku dobiti početni skup Didijevih novčanica i sve događaje koji će se desiti – kupovine i primanje plata. Potrebno je da odredite da li je Didi srećan na početku kao i posle svakog događaja.

Primetimo da je Didi srećan i kada nema uopšte para – tada ne kupije ništa i trči kilometar ispod tri minuta jer tako je u mogućnosti.

Zadatak

Napišite funkcije *init()* i *is_happy()*, koje će biti kompajlirane pomoću žirijevog grader-a. Ove funkcije treba da odrede Didijevu sreću na početku i posle svakog događaja. Funkcije će prihvatati kao parametre početni skup novčanica i skupove novčanica koje se uklanjaju iz trenutnog skupa (prilikom kupovina) ili dodaju u trenutni skup (prilikom primanja plata).

Detalji implementacije

Potrebno je poslati na sistem fajl **happiness.cpp,** koji sadrži funkcije:

bool init(int coinsCount, long long maxCoinSize, long long coins[]).

bool is_happy(int event, int coinsCount, long long coins[]).

Opis parametara:

coinsCount – broj novčanica koje se dodaju (početni skup ili primanje plate) ili uklanjaju (kupovina).

maxCoinSize – maksimalna moguća vrednost novčanice (*M*).

coins[] – niz koji koji sadrži vrednosti novčanica, u slučajnom poretku (**indeksiranje počinje od 0**).

event – tip događaja :

−1 – Kupovina;
1 – Primanje plate.

Funkcija *init* se poziva samo jednom na početku i služi za dobijanje početnog skupa Didijevih novčanica. Zatim se *Q* puta poziva funkcija *is_happy* sa *event* = −1 (kupovina) ili *event* = 1 (primanje plate). Posle svakog poziva, funkcija *is_happy* treba da vrati *true*, ako se Didi oseća srećno sa svojim trenutnim skupom novčanica odnosno *false* ako se ne oseća srećno.

Fajl **happiness.cpp** NE SME sadržati funkciju *main()*, ali sme sadržati druge promenljive i funkcije ukoliko su potrebne za implementaciju funkcija *init* and *is_happy.* Vaš program mora sadržati liniju `#include "happiness.h"` na početku

Ograničenja

Označimo sa *Nc* broj Didijevih novčanica u bilo kom datom trenutku a sa *K* – broj novčanica korišćen prilikom bilo koje kupovine ili primanja plate. Tada važi:

$0 \le Nc \le 200\,000$
$0 \le Q \le 100\,000$
$1 \le M \le 10^{12}$
$1 \le K \le 5$

Garantuje se da prilikom svakog poziva funkcije *is_happy* sa *event* = −1 (kupovina) skup novčanica u nizu *coins*[] je podskup trenutnog skupa Didijevih novčanica.

Primer

pozvana funkcija	događaj	coinsCount	maxCoinSize	coins[]	funkcija vraća
init		5	100	4 8 1 2 16	true
is_happy	−1(kupovina)	2		2 8	false
is_happy	1(primanje plate)	3		7 9 2	true

Subtasks

Subtask	Poeni	Nc	M	Q
1	10	≤ 300	≤ 100	≤ 100
2	20	≤ 20000	$\le 10^{12}$	≤ 1000
3	30	≤200000	≤ 1000000	≤ 100000
4	40	≤ 200000	$\le 10^{12}$	≤ 100000

Lokalno testiranje

Da biste mogli da testirate vaše funkcije *init* i *is_happy* na vašem lokalnom računaru, dobićete fajlove *Lgrader.cpp* i *happiness.h*. Kompajlirajte ih zajedno sa vašim fajlom **circus.cpp** i dobićete program koji možete koristiti za testiranje vaših funkcija.

Program očekuje sledeći ulazni format:

Dva pozitivna cela broja *N* and *M* u prvom redu – početni broj Didijevih novčanica i maksimalna vrednost novčanice.

N pozitivnih celih brojeva u drugom redu, razdvojeni razmacima – vrednosti novčanica u početnom skupu.

Nenegativan ceo broj *Q* u trećem redu – broj događaja.

U svakom od sledećih *Q* redova opisan je jedan događaj – prvo, vrednost promenljive *event*: −1 (kupivina) ili 1 (primanje plate). Nakon toga broj *K* – broj novčanica koje se uklanjaju ili dodaju trenutnom Didijevom skupu novčanica. Na kraju, dato je *K* brojeva, razdvojeni razmacima – vrednosti dodatih/uklonjenih novčanica.

Na standardni izlaz program će odštampati *Q*+1 redova koji sadrže 0 ili 1 – status sreće Didija na početku i posle svakog događaja.

Primer lokalnog testiranja

Input	Output
5 100 4 8 1 2 16 2 -1 2 2 8 1 3 7 9 2	1 0 1

Ultimate TTT

Tic-Tac-Toe (Iks-Oks) je igra za dva igrača sa vrlo jednostavnim pravilima. Igra se odvija na (inicijalno praznoj) tabli sa tri reda i tri kolone. Prvi igrač bira jedno od polja i popunjava ga svojim znakom ('X'). Nakon toga drugi igrač bira jedno prazno polje i popunjava ga svojim znakom ('O'). Treći potez igra prvi igrač ('X') tako što izabere jedno prazno polje i popunjava ga svojim znakom ('X') i tako dalje. Igrači igraju naizmenično i ne mogu preskočiti svoj potez jer to jednostavno nema smisla. Igra se nastavlja sve dok jedan od igrača ne postavi tri svoja znaka u istom redu, koloni, ili jednoj od dve dijagonale. Ako je tabla popunjena, a nijedan od igrača nije postigao pobedničku poziciju, igra je završena sa nerešenim ishodom.

Midža i Zdule igraju modifikovanu verziju ove igre. Umesto standardne table dimenzija 3x3, oni počinju igru na tabli beskonačnih dimenzija. Nakon što prvi igrač odigra potez (apsolutno je nebitno gde), sledeći potez može biti odigran samo na polju koje se potencijalno može nalaziti na tabli 3x3 zajedno sa prvim poljem. Drugim rečima, drugi potez mora biti najviše 2 kolone i/ili reda udaljen od prvog. Sledeći potez sledi sličnu logiku: igrač uvek mora izabrati takvo prazno polje tako da svi već odigrani potezi mogu stati na neku tablu dimenzija 3x3. Obratite pažnju da što više igra odmiče, smanjuje se broj mogućih polja za igru, sve dok se ne formira standardna tabla 3x3 (pod pretpostavkom da nijedan igrač ne pobedi pre toga). Pogledajte primer koji sledi kako biste bolje razumeli igru.

Kao u originalnoj igri, pobednik je igrač koji prvi ima tri susedna polja, u istom redu, istoj koloni, ili dijagonali. Slično, ako se sva validna polja popune pre nego što iko uspe pobediti, smatra se da se igra završila nerešeno.

Pogledajte pažljivo sledeći primer:

<table>
<tr>
<td>. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .</td>
<td>. . . . . .
. . . . . .
. . . . . .
. . . X . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .</td>
<td>. . . . .
. . . . .
. . X . .
. . . . .
. . . . .</td>
<td>. . . . .
. . . . .
. . X . .
. . . . .
. . 0 . .</td>
</tr>
<tr>
<td>. . X . .
. . . . .
. . 0 . .</td>
<td>. . X X .
. . . . .
. . 0 . .</td>
<td>. X X .
. . . .
. 0 . .</td>
<td>. X X 0
. . . .
. 0 . .</td>
</tr>
<tr>
<td>X X 0
. X .
0 . .</td>
<td>X X 0
. X .
0 . 0</td>
<td>X X 0
. X .
0 X 0</td>
<td>X | 0
. | .
0 | 0</td>
</tr>
</table>

U ovom primeru prvi igrač (‘X’) pobeđuje, zato što drugi igrač (’O’) nije koristio optimalnu strategiju.

Smatraćemo *optimalnom igrom* igranje takvih poteza, tako da igrač sigurno pobeđuje ako može pobediti, igra nerešeno ukoliko ne može da pobedi, a može da igra nerešeno, ili gubi, ukoliko ne može ni da remizira, ni da pobedi. Pretpostavljamo da oba igrača igraju optimalno.

Zadatak

Napišite program koji za dato trenutno stanje igre na tabli pronalazi koji potez treba da odigra igrač koji je na potezu kako bi igrao optimalno.

Ulaz

Prva linija standardnog ulaza sadrži dva prirodna broja N i M – broj preostalih validnih redova i broj preostalih validnih kolona, tim redom. Svaka od sledećih N linija sadrži string dužine M, koji opisuje taj red table. Sve linije formiraju ispravno trenutno stanje na tabli. Garantuje se da je najmanje jedan potez odigran (tako da je preostali broj validnih polja za igru konačan), i da igra još nije završena.

Izlaz

Program ispisuje u jednoj liniji standardnog izlaza dva prirodna broja R i C (razdvojena jednim razmakom): gde je R predstavlja broj reda, a C broj kolone (**indeksirani počevši od broja 1**). Uređen par (R, C) predstavlja potez igrača koji je sledeći na potezu, a koji ga vodi u optimalnu poziciju. Ukoliko postoji više optimalnih poteza, možete ispisati bilo koji.

Ograničenja

- $3 \leq \mathbf{N}, \mathbf{M} \leq 5$
- Svi simboli koji opisuju tablu su iz alfabeta {'.' (**tačka**), 'X' (**veliko slovo**), 'O' (**veliko slovo**)}.
- U test primerima vrednim 50% ukupnog broja poena važi **N** = **M** = 3 (dakle tabla je standardna Iks-Oks tabla dimenzija 3x3).

Ocenjivanje

Test primeri su podeljeni u 4 grupe. Svaki grupa test primera nosi 25 poena, ukoliko su svi test primeri unutar grupe korektno rešeni.

Primer

Ulaz	Izlaz
3 4 .XX. .... .O..	1 1

Pojašnjenje

Na potezu je drugi igrač (’O’). Ako odabere (1,4), onda prvi igrač ’X’ može pobediti tako što će odigrati potez (2,3) i staviti drugog igrača ‘O’ u gubitničku poziciju tako da bez obzira koje polje drugi igrač ‘O’ izabere, prvi igrač ‘X’ će imati pobednički potez.

Igrajući na (1,1) sledeći potez na tabli će biti limitiran na kolone 1..3, tako da će biti nemoguće za prvog igrača ’X’ da pobedi u jednom potezu igrajući potez (1,4). Sada drugi igrač ‘O’ igrajući potez (1, 1) ima strategiju koja mu garantuje da će se igra završiti nerešeno.