Scrambled

Donald is a very successful entrepreneur and likes to keep all his things in tremendous order, which is why he keeps all his credit cards in ascending order based on their balance. His best friend, Hillary, doesn’t share his passion for order, so she scrambled all of his credit cards. Being the calculated businessman that he is, Donald would like to know how many of his credit cards are in a wrong order before rearranging them, by which he means how many groups of two cards c*i* and c*j* (*i* *< j*) have *balance[i] > balance[j]*. He would like you to help him in this endeavour.

Data format

The input file, *scrambled.in*, contains, **on its first line**, an integer *N* representing the number of credit cards Donald has. **On the second line** it will contain *N* integer numbers *balance[i]* representing the balance for each card, in the scrambled order.

The output file, *scrambled.out*, should contain **on a single line**, the number of credit cards found in wrong order.

Data limits

1 <= N <= 105

1 <= balance[i] <= 106, for every 0 <= i < N

Time limits: **C++** **-** **0.2s, Java - 0.7s**

Example

|  |  |
| --- | --- |
| scrambled.in | scrambled.out |
| 5  3 5 1 4 6 | 3  *Explanation*: The initial order would have been 1, 3, 4, 5, 6. Therefore, the scrambled pairs are (3, 1), (5, 1), (5, 4) |
| 10  40 20 10 50 60 30 90 80 100 70 | 10  *Explanation*: The initial order would have been 10, 20, 30, 40, 50, 60, 70, 80, 90  Therefore, the scrambled pairs are (40, 20), (40, 10), (40, 30), (20, 10), (50, 30), (60, 30), (90, 80), (90, 70), (80, 70), (100, 70) |

Ultimate Team

Bob is a big FIFA fan, to the point at which it’s become an obsession for him to always have the best players, but he needs the funds to do so. To achieve this, he needs to trade wisely and take advantage of the transfer market’s ever-fluctuating prices.

Since Christmas is coming, Bob knows that he can make a killing on the market by buying players cheap and selling them when their market value is greater. He has somehow managed to get an accurate prediction of Messi’s prices for a period of N days. The prediction is given as a list in which *pi* represents the player’s *price* in day *i*. Bob plans to make multiple, successive transactions during this period, but CAN’T have more than one Messi at a time, and therefore needs to sell him before buying him again.

Bob starts out with a limited budget *B* and can’t buy a Messi which costs more than he can afford. Of course, Bob can add to his budget any profit he gets from buying and selling his Messis. Luckily for him, some of the times he starts with a Messi from opening a random gift pack beforehand.

At the end Bob just wants to have as much profit as possible, and sell hist last Messi, as he is a Cristiano Ronaldo fan anyway.

Data format

On the first line of the input file, *ultimate\_team.in*, you will find three integers *N B and M*.

*N* represents the number of days for which Bob has a prediction of Messi’s price.

*B* represents Bob’s initial budget.

*M* can be either 0 or 1; 0 if Bob starts without an initial Messi to sell and 1 if he does start with an initial Messi to sell.

On the next line you will find *N* integers: p1, p2, … , pN, separated by a whitespace, in which pi, represents Messi’s price on day *i*.

In the output file, *ultimate\_team.out*, you have to print out, on a single line, the maximum profit you can obtain by buying and selling Messi.

Data limits

1 <= N <= 105

0 <= B <= 106

1 <= pi <= 105 for each *i*, where 1 <= *i* <= N

Time limits: **C++** **-** **0.1s, Java - 0.7s**

Example

|  |  |
| --- | --- |
| ultimate\_team.in | ultimate\_team.out |
| 6 20 0  20 50 40 10 50 80 | 100  Explanation:  Bob starts with an initial budget of 20 and no initial Messi to sell. Therefore he can buy a Messi for 20, sell him for 50, buy another for 10 and sell him for 80, totaling a profit of (50-20) + (80-10) = 100 |
| 7 5 0  20 10 30 5 10 10 20 | 15  Explanation:  Bob starts with an initial budget of 5 and no initial Messi to sell. He cannot buy any Messi until his price drops to 5, so his profit is only (20-5) = 15 |
| 7 0 1  20 10 50 80 60 20 10 | 90  Explanation:  Bob starts with an initial budget of 0 and one Messi to sell. Therefore he sells his initial Messi for 20, buys him back for 10 and sells him for 80, so his profit is 20 + (80-10) = 90 |

Maze Escape

Dr Strange is stuck in a circular maze, in Dormammu's world, and this time he can’t bargain his way out of it. He has managed to create a map of his the maze, in which ‘X’ are obstacles, ‘\*’ are free cells and ‘G’ are exit gates. A circular maze has the property that all top cells also neighbor the bottom cells, and all left cells neighbor the right cells. For example, if Stephen is on cell (0, 4) in the example below, he can travel left, right, bottom, but also to cell (3, 4). Obviously, he cannot go into any cell that has an obstacle.

Since he’s very thorough (as most former surgeons turned into master wizards are), before he even starts searching for one of the exit gates, he wants to update the map, for each free cell, with the distance between it and the nearest gate. For this task he needs your help.

Data format

In maze\_escape.in you will find, on the first line, two integers, N and M, which are the height and width of the circular maze. On the following N lines you will find M characters per line, representing the type of cells in the maze, as explained above.

In maze\_escape.out you will need to print an N by M matrix, where each cell (i, j) will contain either the distance between it and the nearest gate, -1 if no gate can be reached, or 0 if it is already a gate. Each cell is separated by a single space.

Data limits

1 <= N, M <= 1000

Time limits: **C++** **-** **0.7s, Java - 1.1s**

Example

|  |  |
| --- | --- |
| maze\_escape.in | maze\_escape.out |
| 4 7  X\*G\*\*\*\*  GX\*X\*\*\*  X\*\*\*\*G\*  \*\*\*\*G\*\* | -1   1  0  1   1   2   2  0  -1  1 -1   2   1   1  -1   3  2  2   1   0   1  3   2  1  1   0   1   2  (in your output, **only a single space is needed**, here we added several spaces to make the matrix easier to read) |

# Stores

You are given the map of a city which contains the locations of all its apartments and stores. The map is given as a graph with N intersections and M streets, and the apartments and stores can only be found in one of the N intersections (an intersection can hold at most one store or apartment, but can also hold none).

To make the life of its citizens easier, we want to analyze this map and determine for each apartment what the nearest store is. If there are two stores placed at an equal distance from an apartment, we will choose the one with the lower index.

### Input Format

The first line of the input file, *stores.in*, will contain two integers, N and M, representing the number of intersections and streets.

On the next M lines you will find 3 integers: x, y, and c, which means that there is a street from x to y with cost c (you can go from x to y and from y to x with the same cost).

After the M lines describing the streets you will find another line with two integers Na and Ns, representing the number of apartments and stores in the city.

Finally, you will find two more lines containing Na and Ns integers, representing the indexes of the intersections that contain apartments and stores, respectively.

### Output Format

In the output file, *stores.out*, you will print out, on one line, Na integers representing the index of the closest intersection containing a store for each of the Na apartments (keeping the same order as in the input file). If an apartment cannot reach any store you will print out 0 for it.

### Constraints

1 ≤ N, M ≤ 50 000

1 ≤ Na ≤ 50 000

1 ≤ Ns ≤ 1 000

The street cost c is an integer between 1 and 1000

Time limits: C++ 0.2s, Java 1.2s

### Example

|  |  |
| --- | --- |
| stores.in | stores.out |
| 8 9  1 3 6  1 5 3  1 6 1  2 3 9  5 6 5  6 8 7  3 6 2  4 7 1000  2 8 5  3 2  1 4 8  2 5 | 5 0 2  Explanation:  There are 8 intersections (of which 1, 4 and 8 have apartments and 2 and 5 have stores).  We print out 3 numbers, representing the indices of the nearest intersection containing a store for each of the 3 apartments:   * from the first intersection’s apartment we can reach the store from intersection 5 with a cost of 3 * from the apartment from intersection 4 we cannot reach any store, so we print 0 * from the apartment from intersection 8 we can reach the store from intersection 2 with a cost of 5 |