(1) OA

1. Nearest Neighboring City

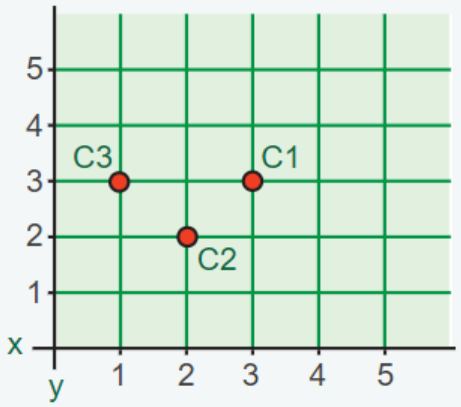
```
// brute force
```

18m left

ALL

1. Nearest Neighboring City

A number of cities are arranged on a graph that has been divided up like an ordinary Cartesian plane. Each city is located at an integral *(x, y)* coordinate intersection. City names and locations are given in the form of three arrays: *c*, *x*, and *y*, which are aligned by the index to provide the city name (*c[i]*), and its coordinates, *(x[i], y[i])*. Determine the name of the nearest city that shares either an *x* or a *y* coordinate with the queried city. If no other cities share an *x* or *y* coordinate, return *'NONE'*. If two cities have the same distance to the queried city, *q[i]*, consider the one with an alphabetically shorter name (i.e. *'ab' < 'aba' < 'abb'*) as the closest choice. The distance is the Manhattan distance, the absolute difference in *x* plus the absolute difference in *y*.



Example

n = 3
c = ['c1', 'c2', 'c3']
x = [3, 2, 1]
y = [3, 2, 3]
q = ['c1', 'c2', 'c3']

The three points at *(x[i], y[i])* are *(3, 3)*, *(2, 2)* and *(1, 3)* represent the coordinates of the cities on the graph. The nearest city to *c1* is *c3*, which shares a *y* value *(distance = (3-1) + (3-3) = 2)*. City *c2* does not have a nearest city as none share an *x* or *y* with *c2*, so this query returns *'NONE'*. A query of *c3* returns *c1* based on the first calculation. The return array after all queries are complete is *['c3', 'NONE', 'c1']*.

Function Description

Complete the function *closestStraightCity* in the editor below.

closestStraightCity has the following parameter(s):
string c[n]: an array of strings that represent the names of each *city[i]*
int x[n]: the x coordinates of each *city[i]*
int y[n]: the y coordinates of each *city[i]*
string q[m]: the names of each city to query

Returns:
string[m]: an array of m strings where the index of i element denotes the return value of the index of i query

Constraints

- $1 \le n, m \le 10^5$
- $1 \le x[i], y[i] \le 10^9$
- *1 ≤ length of q[i] and c[i] ≤ 10*
- Each character of all *c[i]* and *q[i]* is in the range ascii[a-z, 0-9, -]
- All city name values, *c[i],* are unique
- All cities have unique coordinates

▶ **Input Format For Custom Testing**

▼ **Sample Case 0**

Sample Input

```
STDIN       Function
-----       -----
3        → c[] size n = 3
fastcity → c[] = ['fastcity', 'bigbanana', 'xyz']
bigbanana
xyz
3        → x[] size n = 3
23       → x[] = [23, 23, 23]
23
23
3        → y[] size n = 3
1        → y[] = [1, 10, 20]
10
20
3        → q[] size m = 3
fastcity → q[] = ['fastcity', 'bigbanana', 'xyz']
bigbanana
xyz
```

2. Avoiding Landmines

dp

2. Avoiding Landmines

A robot is moving on a grid of size n*m. In one move, it moves one step right or one step down. Some cells of the grid have landmines, so the robot has to avoid them. The grid is given as an array of size n*m, *grid[n][m]*, where *grid[i][j] = 0* means that the cell has a landmine, and *grid[i][j] = 1* means the cell is safe to step upon. Find the number of ways in which the robot can reach the bottom-right most cell, *grid[n-1][m-1]*, starting from the top-left most cell, *grid[0][0]*. The number can be large, so return the value modulo (10^9+7).

Example

grid = [1, 1, 1], [1, 0, 1], [1, 1, 1]

The grid has a landmine at *grid[1][1] = 0*. There are two possible safe ways to go from *grid[0][0]* to *grid[2][2]*, as shown in the figure.

Therefore, return *2 modulo (10^9+7) = 2* as the answer.

Function Description

Complete the function *findSafeWays* in the editor below.

findSafeWays has the following parameter(s):

grid[n][m]: a two-dimensional array of integers of *n* rows and *m* columns

Returns:

int: the number of safe ways through the grid, modulo *(10^9 + 7)*.

17m left

ALL

findSafeWays has the following parameter(s):

grid[n][m]: a two-dimensional array of integers of *n* rows and *m* columns

Returns:

int: the number of safe ways through the grid, modulo $(10^9 + 7)$.

Constraints

- $1 \le n, m \le 1000$
- Each cell in the grid contains either a *0* or a *1*.

▶ **Input Format for Custom Testing**

▼ **Sample Case 0**

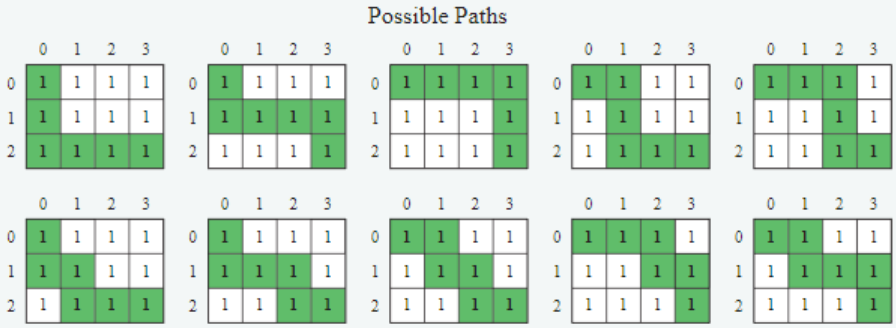
Sample Input 0

```
STDIN       Function
-----       --------
3       →   grid[][] size n=3 m=4
4
1 1 1 1 →   grid = [[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]
1 1 1 1
1 1 1 1
```

Sample Output 0

```
10
```

Explanation 0



There are 10 possible safe paths from *grid[0][0]* to *grid[2][3]* and *10 modulo* $(10^9 + 7) = 10$.

▶ **Sample Case 1**

3. Metro Land Festival

```
find x that minimizes x-cost
find y that minimizes y-cost

then x + y will give you total cost
```

3. Metro Land Festival

Metro Land is a country located on a *2D* plane. They are planning a summer festival for everyone in the country and would like to minimize the overall cost of travel for their citizens. Costs of travel are calculated as follows:

- A city is located at coordinates (x, y).
- The festival is located at coordinates (a, b).
- Cost of travel from a city to the festival = |x-a| + |y-b| per person.

The festival can be held at any integral location in Metro Land. Find the optimal location for the festival, defined as the location with the *minimal total travel cost* assuming all people attend. Return the total travel cost for all citizens to attend the festival.

Example
numPeople = [1, 2]
x = [1, 3]
y = [1, 3]

There is *1* person in City 0 located at *(x[0], y[0]) = (1, 1)* and there are *2 people* in City 1 at *(x[1], y[1]) = (3, 3).*

If the location for the festival is at *(2, 2):*

- The c*ost* of travel from city 0 at (1, 1) to the festival 2 = |1-2| + |1-2| .
- The c*ost* of travel from city 1 at (3, 3) to the festival 4 = 2 *(|3-2| + |3-2|).
- The total cost of travel for all citizens to go to the festival is 2+ 4 = 6.

If the location for the festival is at *(3, 3).*

- The cost of travel from city 0 at (1, 1) to the festival is |1-3| + |1-3| = 4 .
- The cost of travel from city 1 at (3, 3) to the festival |3-3| + |3-3| = 0.
- The total cost of travel for all citizens to go to the festival is 0+ 4 = 4.

The optimal location for the festival is at *(3, 3)* with a cost of *4.* Return 4.

There is another way to analyze the problem. These two matrices show the costs to move one person from a city to any other position (a, b) on the map with an origin at *(x, y) = (1, 1)* at the top left.

There is another way to analyze the problem. These two matrices show the costs to move one person from a city to any other position (a, b) on the map with an origin at *(x, y) = (1, 1)* at the top left.

$$\text{City } 0 = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix} \text{City } 1 = \begin{bmatrix} 4 & 3 & 2 \\ 3 & 2 & 1 \\ 2 & 1 & 0 \end{bmatrix}$$

Since there are two people in City 1, multiply those values by *2*, then sum the two matrices, (City 0 * 1) + (City 1 * 2):

$$\text{City } 0 + \text{City } 1 * 2 = \begin{bmatrix} 0 & 1 & 2 \\ 1 & 2 & 3 \\ 2 & 3 & 4 \end{bmatrix} + \begin{bmatrix} 8 & 6 & 4 \\ 6 & 4 & 2 \\ 4 & 2 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 7 & 6 \\ 7 & 6 & 5 \\ 6 & 5 & 4 \end{bmatrix}$$

The lowest total cost to travel is at grid position *(3,3)* at a cost of *4*.

Function Description

Complete the function *minimizeCost* in the editor below.

minimizeCost has the following parameter(s):

int numPeople[n]: each *numPeople[i]* denotes the number of people in city *i*

int x[n]: each *x[i]* denotes the x coordinate of city *i*

int y[n]: each *y[i]* denotes the y coordinate of city *i*

Returns

int: the minimum cost of getting all the people to an optimal festival location

Constraints

- $1 \le n \le 10^3$
- $1 \le x[i], y[i] \le 10^4$
- $1 \le p[i] \le 50$

▶ **Input Format for Custom Testing**

▼ **Sample Case 0**

Sample Input 0