guruárth **Challenge #4**

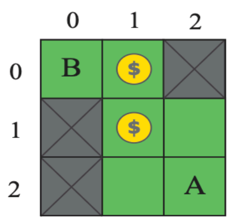
Problem #1: Bob Navigates a Maze
Score points: 100

Bob and Alice have teamed up on a game show. After winning the first round, they now have access to a maze with hidden gold. If Bob can collect all the gold coins and deliver them to Alice's position, they can split the gold. Bob can move horizontally or vertically as long as he stays in the maze, and the cell is not blocked.

The maze is represented by an $n \times m$ array. Each cell has a value, where *0* is *open, 1 is blocked*, and *2* is *open with a gold coin*. Bob starts at the top left in cell in (*row, column) = (0, 0).* Alice's position is given by *(x,y).*
Determine the shortest path Bob can follow to collect all gold coins and deliver them to Alice. If Bob can't collect and give all the gold coins, return -*1*.

Example: *maze = [[0,2,1],[1,2,0],[1,0,0]]* with Alice at *(2,2)* is represented as follows:

$$\begin{matrix} 0 & 2 & 1 \\ 1 & 2 & 0 \\ 1 & 0 & 0 \end{matrix} \longrightarrow$$



B represents Bob's starting position at *(0,0),* and A represents Alice's position (which will also be Bob's final position). As Bob starts at *(0,0),* he has two possible paths to collect the gold and deliver to Alice*: (0, 0) → (0, 1) → (1, 1) → (2, 1) → (2, 2)* and *(0, 0) → (0, 1) → (1, 1) → (1, 2) → (2,2). Both paths have a length of 4 and could represent the shortest path.*

Function Description
Complete the function *minMoves*. The function must return the integer length of Bob's shortest path, or -*1* if it's not possible.

minMoves has the following parameter(s):

maze[maze[0][0],...maze[n-1][m-1]]: a 2D array of integers

x: an integer denoting Alice's row coordinate

y: an integer denoting Alice's column coordinate

Constraints

- *1 ≤ n, m ≤ 100*
- *0 ≤ the number of coins ≤ 10*
- *1 ≤ x < n*
- *1 ≤ y < m*

Input Format For Custom Testing

The first line contains an integer *n*, the numbers of rows in *maze*.

The second line contains an integer *m*, the number of columns in *maze*.

Each of the next *n* lines contains *m* space-separated integers that describe the cells of each row in *maze*.

The next line contains an integer *x*.

The next line contains an integer, *y*.

Sample Case 0

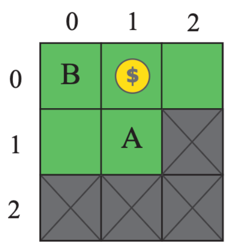
Sample Input 0

```
STDIN          Function Parameters
-----          -------------------
3      →      maze[][] number of rows n = 3
3      →      maze[][] number of columns m = 3
0 2 0  →      maze[][] = [ [0 2 0], [0 0 1], [1 1 1] ]
0 0 1
1 1 1
1      →      x = 1
1      →      y = 1
```

Sample Output 0

```
2
```

Explanation 0



The shortest path Bob can take is (0, 0) → (0, 1) → (1, 1).

Sample Case 1

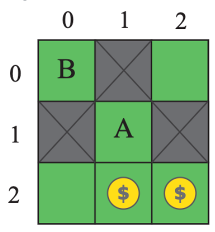
Sample Input 1

```
STDIN          Function Parameters
-----          -------------------
3       →      maze[][] number of rows n = 3
3       →      maze[][] number of columns m = 3
0 1 0   →      maze[][] = [ [0 1 0], [1 0 1], [0 2 2] ]
1 0 1
0 2 2
1       →       x = 1
1       →       y = 1
```

Sample Output 1

```
-1
```

Expanation 1



It is not possible for Bob to reach Alice, so return −1.

Sample Case 2

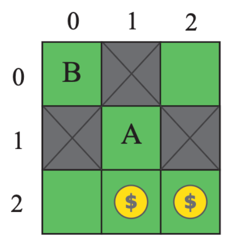
Sample Input 2

```
STDIN        Function Parameters
-----        -------------------
3      →     maze[][] number of rows n = 3
3      →     maze[][] number of columns m = 3
0 2 0  →     maze[][] = [ [0 2 0] , [1 1 2] , [1 0 0] ]
1 1 2
1 0 0
2      →     x = 2
1      →     y = 1
```

Sample Output 2

```
5
```

Explanation 2



The shortest path Bob can take is (0, 0) → (0, 1) → (0, 2) → (1, 2) → (2, 2) → (2, 1).

Problem #2: Paint the Ceiling
Score points: 100

You want to build yourself a house. The building company you hired can only build houses with sides from their specific set *s*. That means they can build you a square house or a rectangular one but if and only if its length and width belong to the set *s*.

This month, they have a special promotion: they will paint the ceiling of a new house for free...but only if its area is not more than *a*. You want them to do it for free but you also want to be sure that the house will be comfortable and not too small. How many possible house configurations can you create to have the ceiling painted for free given the side lengths offered?

There is a method to how the company decides what lengths of sides to produce. To determine *n* lengths of wall segments to offer, they start with a seed value *s0*, some variables *k, b* and *m*, and use the following equation to determine all other side lengths *s[i]*:

*s[i]=((k*s[i-1]+b) mod m)+1+s[i-1]* for $1 \le i < n$

For example, you are given *s[0] = s0 = 2* and they will produce *n = 3* total wall lengths. If *k = 3, b = 3* and *m = 2* we have:

```
s       i       calculation     result
[2]     1       ((3*2+3)%2)+1+2 4
[2,4]   2       ((3*4+3)%2)+1+4 6
[2,4,6]
```

Now that we have our set of lengths, we can brute force the solution using the following tests assuming *a = 15*:

```
s = [2, 4, 6]

s1      s2      s1*s2   s1*s2 <= a
2       2       4       True
2       4       8       True
2       6       12      True
4       2       8       True
4       4       16      False
4       6       24      False
6       2       12      True
6       4       24      False
6       6       36      False
```

There are *5* combinations that will result in a free paint job. Brute force will not meet the time constraints on large sets.

Function Description

Complete the function *variantsCount*. The function must return an integer that denotes the number of variants that allow you to use the promotion.

variantsCount has the following parameter(s):
   n: an integer, the number of wall lengths offered
   s0: an integer, the length of the shortest wall
   k, b, m: three arbitrary integers
   a: a long integer, the largest area that will be painted for free

Constraints

- $1 \le n \le 6 * 10^7$
- $1 \le s[i] \le 10^9$, $0 \le i < n$
- $1 \le k, b, m \le 10^9$
- $1 \le a \le 10^{18}$

Sample Case 0
Sample Input 0

```
3
1
1
1
2
4
```

Sample Output 0

```
6
```

Explanation

n = 3, *s[0] = s0 = 1, k* = 1, *b = 1*, *m* = 2 and *a* = 4. That means that *s[1]=((1*1+1) mod 2)+1+1 = 2, s[2]=((1*2+1) mod 2) +1+2 = 4.*
That yields the following variants: 1*1 (area=1<=4, good); 1*2 (area=2<=4, good); 1*4 (area=4<=, good); 2*1 (area=2<=4, good); 2*2 (area=4<=4, good); 2*4 (area=8>; bad); 4*1 (area=4<=4, good); 4*2 (area=8>4, bad) and 4*4 (area=16>4, bad). 6 of the variants are good and 3 are bad.

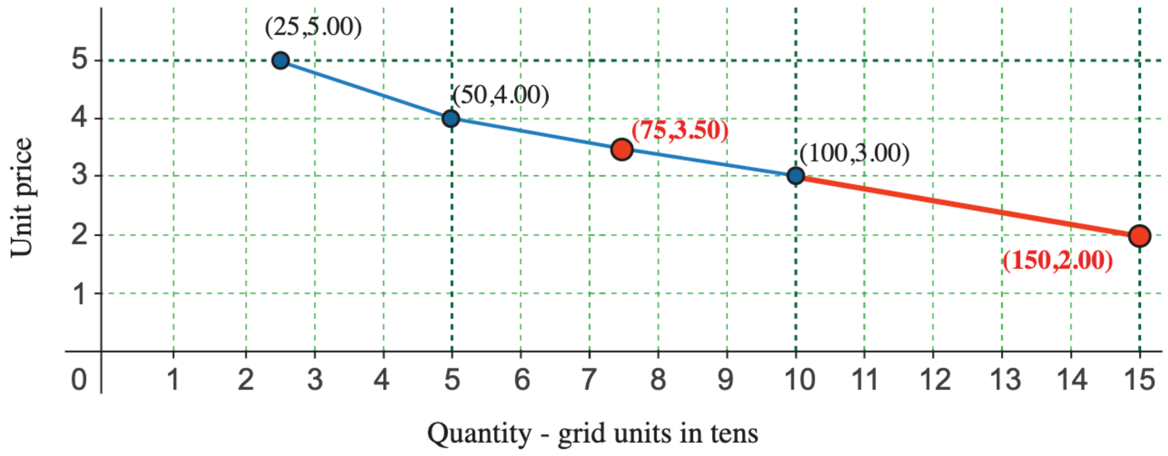
Problem #3: VM Pricing
Score points: 100

A cloud service provider offers quantity discounts based on the number of virtual machines a customer needs. Their offerings vary from *2* to *2000* instances. When pricing is requested, the customer's representative refers to a list of past pricing. Given a list of past price quotes and the number of instances a customer needs, determine the per-instance price for the customer.

The method for determining price is as follows:

- If the number of instances needed is *exactly the same* as the quantity for a prior customer, the unit price is that price.
- If there is a price for a larger number and a price for a smaller number of instances, linearly interpolate the price of the quantity needed from the unit prices for the closest smaller and larger quantities.
- If the quantities for which there is past data are *all smaller* or *all larger* than the amount needed, then linearly extrapolate the unit price from the *2* points *closest* to the quantity needed.
- If the database only has *1* quantity, that is the price per unit.
- Sometimes, price quotes lapse. When that happens, the old pricing is overwritten with either a *0* or a negative number. The quantities associated with zero or negative unit prices must be disregarded.

For example, assume the price breaks occur for *instances = [25, 50, 100]* units at *price = [5.0, 4.0, 3.0]*. A diagram follows with pricing for *75* and *150* units. In the graph, price versus quantity for given values are in blue. The target numbers of instances and the linear extrapolation are plotted in red.



Function Description
Complete the function *interpolate*. The function must return the expected price per unit rounded to two places after the decimals and cast as a string.

interpolate has the following parameter(s):
n: an integer that denotes the number of instances required
instances[instances[0],...instances[m-1]]: an array of integers in increasing order, each a number of instances ordered in the past
price[price[0],...price[m-1]]: an array of floating point numbers where *price[i]* is the unit price when *instances[i]* units were purchased.

Note: The *interpolate* function's array parameters may be vectors, where necessitated by the language.

Constraints

- $2 \le n \le 2000$
- $1 \le m \le 100$
- $|instances| = |price| = m$
- $instances[i] < instances[j]$, where $0 \le i < j < m$

Input Format For Custom Testing
The first line contains an integer, *n*, the number of instances needed.
The second line contains an integer, *m*, the size of the array *instances*.
Each line *i* of the *m* subsequent lines (where $0 \le i < m$) contains an integer that describes *instances[i]*.
The next line again contains the integer, *m*, the size of the array price.
Each line *i* of the *m* subsequent lines (where $0 \le i < m$) contains a floating point number that describes *price[i]*.

Sample Case 0
Sample Input 0

```
25
5
10
25
50
100
500
5
2.46
2.58
2.0
2.25
3.0
```

Sample Output 0

```
2.58
```

Explanation 0
The following arguments are passed to the *interpolate* function:
n = 25
instances = { 10, 25, 50, 100, 500 }
price = { 2.46, 2.58, 2.0, 2.25, 3.0 }

The quantity *25* is in the database, so *vmPricing* returns the unit price associated with that quantity, *2.58*, cast as a string.