Question - 1
Maximize the Power

Given an array *arr* of *n* non-negative integers and an array *power* of *k* integers where *k* is an even number, perform the following operation in steps.

- Select two integers *i* and *j* such that $0 \le i < j < |power|$. In each operation, *i* and *j* are selected independently.
- if $power[i] \le power[j]$ add the summation of the subarray *arr[power[i]...power[j]]* to the power.
- if *power*[i] > *power*[j] add the summation of the subarray *arr[power[j]...power[i]]* to the power.
- Delete *i-th* and *j-th* elements from *power*, and its length is reduced by 2.

Starting at 0 initial power, maximize the final power after exactly *k/2* operations. Return that maximum power modulo (10^9+7).

Note: Subarray *arr[l...r]* denotes the subarray formed by the elements *arr[l], arr[l+1], ... arr[r-1], arr[r].*

Example

arr = [3, 5, 6, 0, 7] and *power* = [3, 1, 0, 2].

k = 4, the size of *power,* so perform *k/2 = 2* operations.

One of the optimal approach is shown.

- Select *i*=0, *j*=2. Here, *power[0]* = 3 and *power[2]* = 0. Add the sum of subarray *arr[0...3],* (3 + 5 + 6 + 0) = 14 to the power, 0 + 14 = 14. Remove the two elements from *power*, and *power* is [1, 2].
- Select i=0, j=1. Here, *power[0]* = 1 and *power[1]* = 2. Add the sum of subarray *arr[1..2]*, 5 + 6 = 11, so power is 14 + 11 = 25.

Return the maximum power possible, 25. (25 modulo (10^9+7) = 25)

Function Description

Complete the function *getMaximumPower* in the editor below.

getMaximumPower has the following parameter(s):

int arr[n]: An array of integers

int power[k]: An array of integers

Returns

int: the maximum power modulo 10^9+7

Constraints

- $1 \le n \le 10^5$
- $2 \le k \le 10^5$, *k* is even.
- $0 \le arr[i] \le 10^7$, $0 \le i < n$
- $0 \le power[i] < n$

▼ Input Format for Custom Testing

The first line contains an integer, n, number of elements in *arr*.

Each of the next *n* lines contains an integer *arr[i]*.

The next line contains an integer, k, number of elements in *power*.

Each of the next *k* lines contains an integer *power[i]*.

▼ Sample Case 0

Sample Input 0

```
STDIN       FUNCTION
-----       --------
5      →    n = 5
2      →    arr = [2, 4, 2, 1, 6]
4
2
1
6
4      →    k = 4
4      →    power = [4, 1, 1, 3]
1
1
3
```

Sample Output 0

```
20
```

Explanation

Optimal operations

```
                                                 subarray    resultant
i   j   power[i]    power[j]    subarray         sum         power
--  --  --------    --------    --------         ---------   ---------
0   1   4           1           [4, 2, 1, 6]         13      [1, 3]
0   1   1           3           [1, 6]                7      []

Total power = 13 + 7 = 20
```

Sub-optimal operations for comparison

```
                                                 subarray    resultant
i   j   power[i]    power[j]    subarray         sum         power
--  --  --------    --------    --------         ---------   ---------
1   2   1           1           [4]                  4       [4, 3]
0   1   4           3           [1, 6]               7           []

Total power = 4 + 7 = 11
```

▼ Sample Case 1

Sample Input 1

```
STDIN       FUNCTION
-----       --------
4      →    n = 4
1      →    arr = [1, 2, 3, 4]
2
3
4
2      →    k = 2
0      →    power = [0, 0]
0
```

Sample Output 1

```
1
```

Explanation

Here, *arr* = [1, 2, 3, 4] and *power* = [0, 0]. There is only one way to perform the operation.

Select *i* = 0, *j* = 1. Here, *power[0]* = 0 and *power[1]* = 0. We add the sum of subarray *arr[0...0]* which is 1 to the power making it 1.

Question - 2
Encircular

Build a computer simulation of a mobile robot. The robot moves on an infinite plane, starting from position *(0, 0)*. Its movements are described by a command string consisting of one or more of the following three letters:

- *G* instructs the robot to move forward one step.
- *L* instructs the robot to turn left in place.
- *R* instructs the robot to turn right in place.

The robot performs the instructions in a command sequence in an infinite loop. Determine whether there exists some circle such that the robot always moves within the circle.

Consider the commands R and G executed infinitely. A diagram of the robot's movement looks like:

RG → RG
↑ ↓
RG ← RG

The robot will never leave the circle.

Function Description
Complete the function *doesCircleExist* in the editor below. The function must return an array of *n* strings either *YES* or *NO* based on whether the robot is bound within a circle or not, in order of test results.

doesCircleExist has the following parameter(s):
commands[commands[0],...commands[n-1]]: An array of *n commands[i]* where each represents a list of commands to test.

Constraints

- $1 \leq |commands[i]| \leq 2500$
- $1 \leq n \leq 10$
- Each command consists of *G*, *L*, and *R* only.

▼ Input Format for Custom Testing

Input from stdin will be processed as follows and passed to the function.

The first line contains an integer *n*, the number of elements in *commands*.
The next *n* lines each contain a string describing *commands[i]* where $0 \leq i < n$.

▼ Sample Case 0

Sample Input 0
3
G
L
RGRG

Sample Output 0

```
NO
YES
YES
```

Explanation 0

There are *n = 2* commands:

1. For *commands[0] = "G"*, the robot will move forward forever (G → G → G →...) without ever turning or being restricted to a circle. Set index *0* of the return array to *NO*.
2. For *commands[1] = "L"*, the robot will just turn *90* degrees left forever without ever moving forward (because there is no *"G"* instruction). The robot is effectively trapped at one spot, thus bound within a circle. Set index *1* of the return array to *YES* .
3. For *commands[2] = "GRGR"*, concatenate the string to *"GRGRGRGR"* and it will follow the circular path in the statement example. Set index *2* of the return array to *YES*.

▼ Sample Case 1

Sample Input 1

```
1
GRGL
```

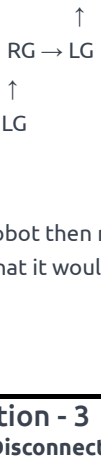
Sample Output 1

```
NO
```

Explanation 1

Consider the robot's initial orientation to be facing north toward the positive *y* direction. The robot performs the following four steps in a loop:

1. Go forward one step. The robot moves from *(0, 0)* to *(0, 1)*.
2. Turn right. The robot turns eastward, facing the positive *x* direction.
3. Go forward one step. The robot moves from *(0, 1)* to *(1, 1)*.
4. Turn left. The robot turns northward, facing the positive *y* direction again.



The robot then repeats these steps infinitely, following an endless zig-zag path in the northeasterly direction. Because the robot will never turn in such a way that it would be restricted to a circle, set index *0* of the return array to *NO*.

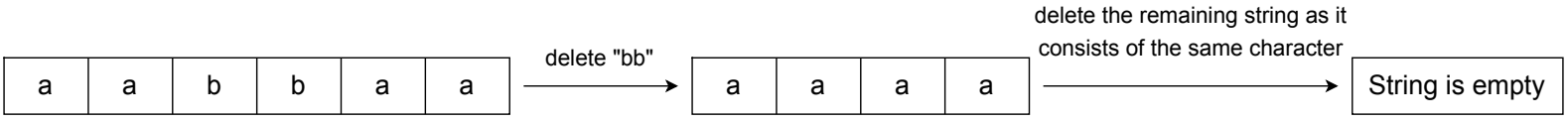
Question - 3

Node Disconnection

A secure communication network has been compromised and the cyber security team must restore its security. The network, *series,* is represented as a series of nodes identified using lowercase English letters. The nodes must be disconnected in order to remove the threat. In a single operation, any number of adjacent nodes identified by the same character can be disconnected. Find the minimum number of operations required to disconnect all the nodes and secure the network.

Example

Suppose *n* = 5 and *series* = "aabbaa".



The minimum number of operations required to delete the entire series is 2.

Function Description
Complete the function *getMinOperations* in the editor below.

getMinOperations has the following parameter:
string series: a series of nodes

Returns
int: the minimum number of operations required to delete the entire series

Constraints

- 1 ≤ *series* ≤ 500
- It is guaranteed that *series* contains lowercase English letters only.

▼ Input Format For Custom Testing

The only line contains a string, *series.*

▼ Sample Case 0

Sample Input For Custom Testing

```
STDIN         FUNCTION
-----         --------
abaca    →    series = "abaca"
```

Sample Output

```
3
```

Explanation
It is optimal to delete the substrings "b" and "c" first in two operations to get the string "aaa" which can be deleted in the next operation.

▼ Sample Case 1

Sample Input For Custom Testing

```
STDIN         FUNCTION
-----         --------
abcddcba   →  series = "abcddcba"
```

Sample Output

```
4
```

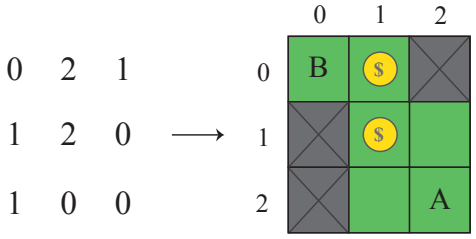
Explanation
It is optimal to delete "dd" first to get "abccba". Then delete "cc" to leave "abba", "bb" to get "aa" and finally delete "aa".

Question - 4
Bob Navigates a Maze

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 × 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:



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* in the editor below. 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

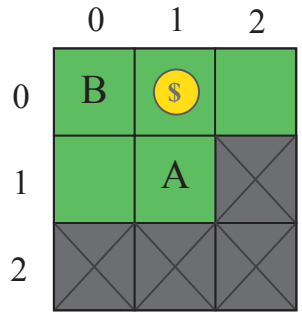
Sample Input 0

```
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

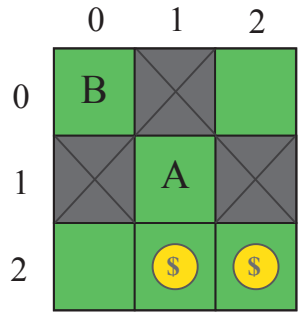
Sample Input 1

```
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
```

Explanation 1



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

▼ Sample Case 2

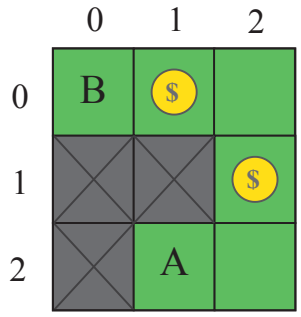
Sample Input 2

```
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).*