PSG COLLEGE OF TECHNOLOGY
DEPARTMENT OF APPLIED MATHEMATICS AND COMPUTATIONAL SCIENCES
15XW26 MScSoftware Systems DATA STRUCTURES
Worksheet 3 Stacks

1. Implement Stack ADT using C++ class

   Create a class called stack with data memebters as S(stack), n and top. Initialize top=-1 and n=10. Create member functions for push(S,elt), pop(S), isstackempty(S), isstackfull(S), displaytop(), displaytopelt().

2. Given a paragraph of text with brackets, check parenthesis matching using stacks.

   Example : A[]="`([]{()[]})[{}]`" . Display the stack after each operation

3. Find the minimum element using stack, where stack is used to store elements and in that process it determines the minimum.

   Example :

Input	Stack	Aux stack	min
Push(10)	10	10	10
Push(7)	10 , 7	10, 7	7
Push(8)	10,7,8	10,7	7
Push(3)	10,7,8,3	10,7,3	3
Pop()	10,7,8	10,7	7
Pop()	10, 7	10,7	7

   Use two stacks, one to store elemnt and other to determine the minimum from stack.

4.

You are a waiter at a party. There are N stacked plates on pile A_0. Each plate has a number written on it. Then there will be Q iterations. In i-th iteration, you start picking up the plates in A_{i-1} from the top one by one and check whether the number written on the plate is divisible by the i-th prime. If the number is divisible, you stack that plate on pile B_i. Otherwise, you stack that plate on pile A_i. After Q iterations, plates can only be on pile $B_1, B_2, \ldots, B_Q, A_Q$. Output numbers on these plates from top to bottom of each piles in order of $B_1, B_2, \ldots, B_Q, A_Q$.

Input Format

The first line contains two space separated integers, N and Q.
The next line contains N space separated integers representing the initial pile of plates, i.e., A_0. The leftmost value represents the bottom plate of the pile.

Sample Input

```
5 1
3 4 7 6 5
```

Sample Output

```
4
6
3
7
5
```

Initially:

= [3, 4, 7, 6, 5]<-TOP

After 1 iteration:

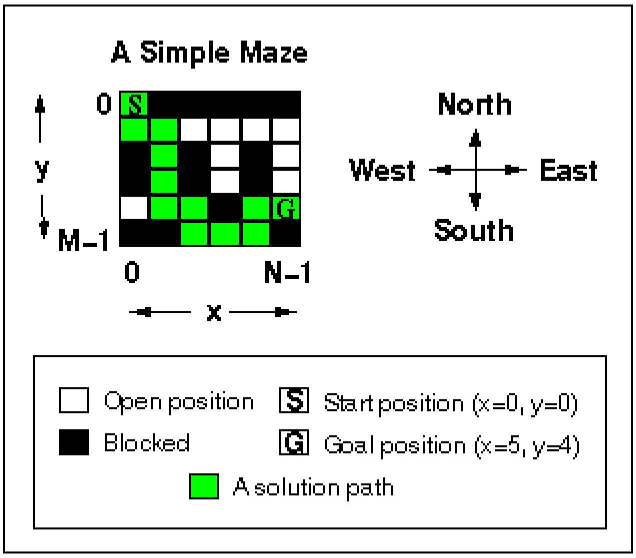
= []<-TOP

= [6, 4]<-TOP

= [5, 7, 3]<-TOP

We should output numbers in first from top to bottom, and then output numbers in from top to bottom.

5. A robot is asked to navigate a maze. It is placed at a certain position (the *starting* position) in the maze and is asked to try to reach another position (the *goal* position). Positions in the maze will either be open or blocked with an obstacle. Positions are identified by (x,y) coordinates.



At any given moment, the robot can only move 1 step in one of 4 directions. Valid moves are:

- Go North: (x,y) -> (x,y-1)
- Go East: (x,y) -> (x+1,y)
- Go South: (x,y) -> (x,y+1)
- Go West: (x,y) -> (x-1,y)

Note that positions are specified in zero-based coordinates (i.e., 0...size-1, where *size* is the size of the maze in the corresponding dimension).

The robot can only move to positions without obstacles and must stay within the maze.

The robot should search for a path from the starting position to the goal position (a *solution path*) until it finds one or until it exhausts all possibilities. In addition, it should mark the path it finds (if any) in the maze.

Representation

To make this problem more concrete, let's consider a maze represented by a matrix of characters. An example 6x6 input maze is:

```
S#####
.....#
#.####
#.####
...#.G
##...#
```

'`.`' - where the robot can move (open positions)
'`#`' - obstacles (blocked positions)
'`S`' - start position (here, x=0, y=0)
'`G`' - goal (here, x=5, y=4)

Aside: Remember that we are using *x* and *y* coordinates (that start at 0) for maze positions. A *y* coordinate therefore corresponds to a row in the matrix and an *x* coordinate corresponds to a column.

A path in the maze can be marked by the '`+`' symbol...

A *path* refers to either a *partial* path, marked while the robot is still searching:

```
+#####
++++.#
#.####
#.####
...#.G
##...#
```

(i.e., one that may or may not lead to a solution). Or, a *solution* path:

```
S#####
++...#
#+####
#+####
.++#+G
##+++#
```

which leads from start to goal.