Input Format

The first line contains an integer, x, denoting the number of segments.

The next line contains an integer, n, denoting the length of the pipe.

Each line i of the n subsequent lines (where 0 <= I <= n) contains an integer denoting the flow rate of the unit length of pipe from i to i+1.

Output Format

Output a single integer denoting the maximum sum of flow rates achieved by the new pipe segments if they are cut optimally.

Constraints

- 1 <= x <= n <= 500
- 1 <= flow rate of pipes <= 10^9

Peaks and Troughs

A function generates values that can be plotted as a line graph. Peaks are points at which the function output changes from increasing to decreasing and troughs are points at which the function output changes from decreasing to increasing.

The first or last value in the graph is either a peak or trough depending on whether it is greater than or lesser than its neighbouring value. If it is equal, it is neither a peak nor a trough.

Given the output of this function, find the maximum distance between any two consecutive peaks or two consecutive troughs.

[Note: distance is the difference in the index at which the value occurs]

Input Format

The first line contains an integer, n, where n denotes the number of values generated.

Each line i of the n subsequent lines (where 0 <= i <= n) contains the i^{th} value generated by the function.

Output Format

Print a single integer denoting the maximum distance between any two consecutive peaks or two consecutive troughs.

Constraints

- 1 <= n <= 10^5
- 1 <= values generated by the function <= 10^5

Rock Paper Scissor

In a Rock Paper Scissor tournament, the contestants stand in a straight line. Each pair of consecutive players compete in a round. If there is an odd number of players, the last one in the line qualifies for the next round without playing. For each game, each player indicates either a rock, paper or scissors denoted *'R'*, *'P'* or *'S'* respectively. Outcomes are as follows:

- *paper beats rock, P > R*
- *scissors beat paper, S > P*
- *rock beats scissors, R > S*

After each round, the winners remain and the losers are out of the competition. In case of a tie, both players lose. A player would like to win the competition and has one advantage: the knowledge that each other player uses only one type of hand formation in all the rounds of the tournament. Determine how many times the player will have to change the hand formation in order to win the competition.

For example, the nuber of players, *n=3*, and the player of interest, *POI*; is standing at position given by *a = 2* (0-based index). The hand formations used by other players are given by formations = *"PS"* with length *n-1 = 2*. In the first round, scissors (*S*) beats paper (*P*). *POI* must then beat the winner of round one who always chooses scissors (*S*). *POI* will choose rock (*R*) and win the tournament after having chosen only one hand formation, hence the answer is 0.

Function Description

Complete the fucntion *handFormationChange* in the editor below. The function must return an integer, the number of times the *POI* needs to change hand formation.

handFormationChange has the following parameter(s):

- *n:* an integer, the number of players in the tournament
- *a:* an integer, the POI's position in the line, 0-indexed
- *formations:* a string, the hand formations of players other than *POI*

Constraints

- *2 <= n <= 105*
- *0 <= a < n*
- *formations[i] {'R', 'P', 'S'} (0 <= i < n-1)*

Sample Input

3
2
RP

Explanation:

The *POI* is at position 2 to start. Of the first two players, paper beats rock. Next, *POI* chooses scissors to beat paper and win the tournament. Only one formation is required, so no change in hand formation is required.

Circular Trail

Do you remember *"Rock Lee"* from the Anime *"Naruto"*? The ordinary kid who works hard to reach to the top and his coach *"Mighty Guy"* who trains : *"Rock Lee"* to become stronger.

As part of his training, *Rock Lee* will run on a trail of length n labeled from *1 to n*. His coach gives him a starting and ending point. Usually, he runs m segments every day. According to Mighty Guy's instructions, in i^{th} segment, Rock Lee has to Start his run from *spot[i]* and end at *spot [i+1] (where*

0<=i<=m-1). The trail is circular, so if *spot[i+1] < spot[i]*, he has to run past the last marker and continue running from marker 1 until he reaches *spot[i-1]*. On each segment, *Rock Lee* visits each marker from beginning to end of the segment. Find the most frequesntly visited spot in the entire circular trail. If there are multiple markers with the same number of visits, choose the smallest one.

For example, given a track of length *n=3*, and *3* segments to run ending at *spot = [3,3,2]*, on the first segment, *Rock Lee* visits *spots 1, 2, and 3*. On the second segment, he visits *spots 3, 1, 2, 3*, and on the third segment he visits *3, 1, 2*. He hs visited *spot 3* the most times, with a visit count of *4*.

Function Description
Complete the function *circularTrail* in the editor given below. The function must return an integer that denotes the position number which appears in Rock Lee's running track the most number of times. If there are multiple answers, return the minimum one.

circularTrail has the following parameters:

- An integer *n*, that denotes the length of the trail.
- An array of *m* integers, *spot*, where the value of each element *spot[i]* means when *Rock Lee* starts running from position *spot[i-1]*, he will finish at position *spot[i]*. And when he starts running from position *spot[i]*, he will finish it at the position *spot[i+1] (where 0 <= i < m)*.

Constraints

- *1 <= n <= 105*
- *2 <= m <= 105*
- *1 <= spot[i] <= n (where 0 <= i < m)*
- *spot[i] != spot[i+1] (where 0 <= i < m-1)*

Sample Input For Custom Testing
10
4
1
5
10
5

Sample Output
5

Explanation
n = 10
m = 4
spot = {1,5,10,5}

Position 5 is the most visited position, visited a total of 3 times. It appears in the first *(1 -> 5)*, second *(5 -> 10)* and third *(10 -> 5)* segments.

Mountain Climbing

The land where a mountain climber decided to go is represented as a 2D grid of n rows and m columns, let's call this grid mountain, where $mountain_{ij}$ represents the height of the spot at the ith and the jth column. The mountain climber knows that he can start his journey from any spot, however he doesn't know how to get down, he knows that while standing on any spot on the

mountain he can go to any adjacent spot such that the height of the destination spot is not greater than that of the spot he is standing on at that time.

Note that two spots are adjacent if they share an edge in the grid mountain.

So the mountain climber decided to ask you for help, he wanted to know for each spot in the grid if the considers that spot his starting point, what is the smallet height he can reach from there through any sequence of moves?

E.g. Let's consider a grid mountain = {{7 2 5} {2 3 0}} the answer for the cells will be {{2 2 0} {2 0 0}} considering all spots:

- He can go from mountain 1,1 with a height equal to 7 to mountain 2,1 directly.
- He can't go from mountain1,2 to any spot since all options surrounding him have strictly bigger height.
- He can go from mountain1,3 with a height equal to 5 to mountain2,3 directly.
- He can;t go from mpountain2,1 to any spot since all options surrounding him have strictly bigger height.
- He can go from mountain2,2 to mountain2,1 or mountain2,3 so he chooses mountain2,3 since it has____
- He can't go from mountain 2,3 to any spot since all options surrounding him have strictly bigger height.

Function Description
Complete the function solveMountain in the editor below. The function must return a 2D array of integers representing the minimum height *mountain[i][j]* can reach.

SolveMountain has the following parameter(s):
mountain[mountain[0],.....mountain[n-1]]: a 2D array of integers

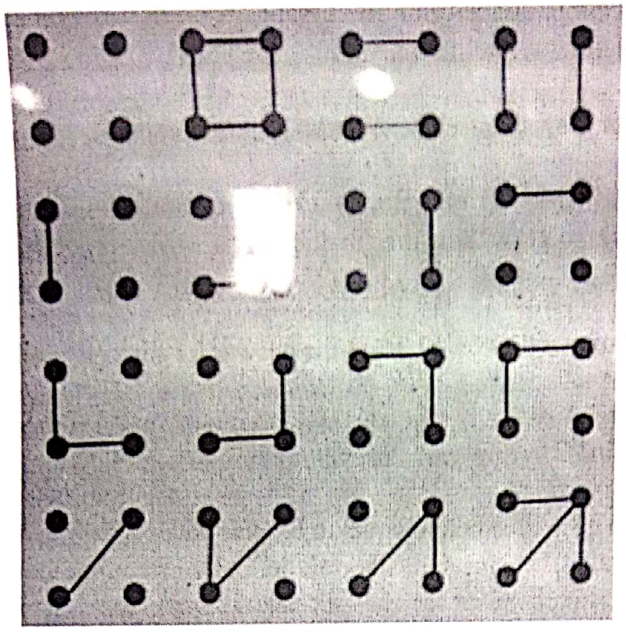
Constraints
$1 <= n, m <= 1000$
$0 <= mountain_{ij} <= 10^9$

Drawing Edge

Given a number of vertices, find the number of ways edges can be drawn between them. It is not necessary to connect all edges or form connected components per se. Since the number of ways edges can be drawn might be very large, print the answer as a modulo value of 10^9+7.

Note: Two ways of drawing edges are considered different if the number of edges is different or at least one pair of vertices has a different connection.

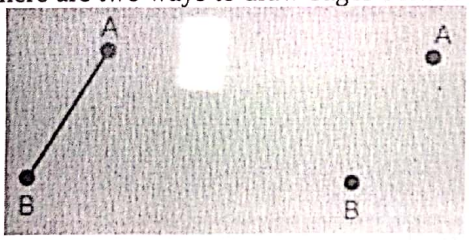
For example, if there are four vertices, some of the 64 possible edge sets are as follows:



Input Format for Custom Testing
The first line contains an integer, n, the number of vertices in the graph.

Sample Input
2
Sample Output
2

Explanation
Since there are two vertices, there are two ways to draw edges between the vertices.



Game Events

Two teams were playing a football (soccer) match and the record of events for each team is available. There are 4 possible events, Goal (G), Yellow Card (Y), Red Card (R) and Substitution (S). The first three events (G, Y, R) are represented as player-name time event-name whereas the last event (S) is represented as player-name time event-name second-player-name. The time is represented in minutes from the start of the game. A football game is divided into two halves of 45 minutes each, and sometimes a little extra time is given at the end of each half, which is represented in the time as time+extra-time. So, there can be two types of times, for example, 32 and 45+2. The extra time is always considered to have occured before the events of the next half, so, 45+2 happened earlier than 46.

Merge the events for each team into a single game event with teams name in front and sorted by time and event in order of G, Y, R, S. In the case of the same event happening at the same time, output should be sorted lexicographically based on teams name and players name.

Take for example, the first team team1 = “EDC” with events recorded as events1 = {‘Name1 12 G’, ‘FirstName LastName 43 Y’} and second team team2 = “CDE” with events recorded as events2 = {‘Name3 45+1 S Subname’, ‘Name4 46 G’}, then the chronological order of events is given by:

EDC Name1 12 G
EDC FirstName LastName 43 Y
CDE Name3 45+1 S SubName
CDE Name4 46 G

Function Description

Complete the functon *getEventsOrder* in the editor below. The function must return an array of strings.

getEventsOrder has the following parameter(s):

- team1 a string, the name of the first team
- team2 a string, the name of the second team