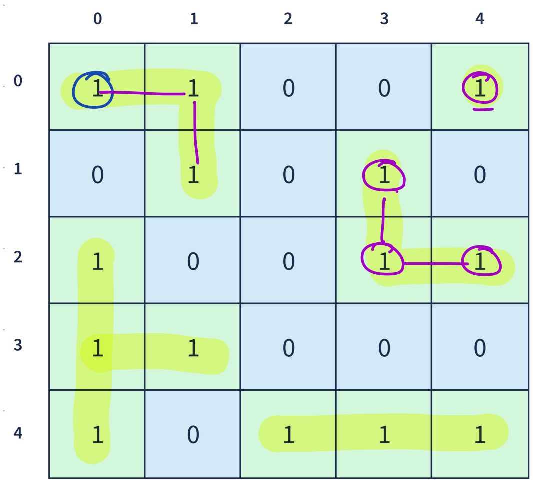
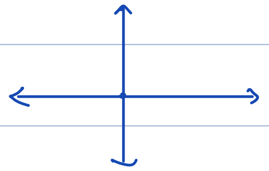
Number of Islands

You are given a 2D grid of '1's (land) and 'O's (water). Your task is to determine the number of islands in the grid. An island is formed by connecting adjacent (horizontally or vertically) land cells. Diagonal connections are not considered. Given here if the cell values has 1 then there is land and 0 if it is water, and you may assume all four edges of the grid are all surrounded by water.

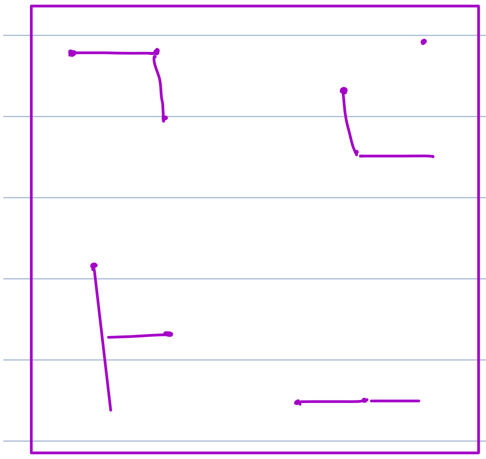
arr[N][N]



	0	1	2	3	4
0	1	1	0	0	1
1	0	1	0	1	0
2	1	0	0	1	1
3	1	1	0	0	0
4	1	0	1	1	1



Ans = 5

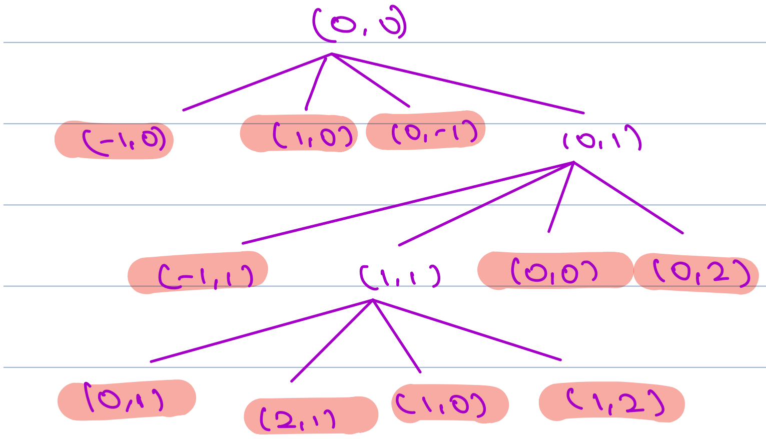


Note :- 1 is representing graph node.

Note :- Adj. list → Any node has max 4 nbrs.

arr[N][N]

	0	1	2	3	4
0	~~1~~ 2	~~1~~ 2	0	0	~~1~~ 2
1	0	~~1~~ 2	0	2 ~~1~~	0
2	2 ~~1~~	0	0	~~1~~ 2	~~1~~ 2
3	2 ~~1~~	~~1~~ 2	0	0	0
4	~~1~~ 2	0	2 ~~1~~	1	1



```
int island ( int mat [N][M]) {
        int c = 0;
        for (i=0; i<n; i++) {
            for (j=0; j<m; j++) {
                if (mat [i][j]==1){
                        c++;
                        dfs(mat, i,j);
                }
            }
        }
        return c;
}
```

T.C -> O(m*n)

S.C -> O(m*n)

```
void dfs ( int mat [N][M] , int i, int j) {
    if (i<0 || j<0 || i>=N || j>=M || mat[i][j]==0 ||
              mat[i][j]==2) {

              return;
    }

    mat [i][j] = 2;

    dfs (i
    dfs (i+1, j);
    dfs ( i-1, j);
    dfs ( i, j+1);
    dfs ( i, j-1);
}
```

Max Profit from stock Prices

Say you have an array, A, for which the ith element is the price of a given stock on day i.

Design an algorithm to find the maximum profit.

You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

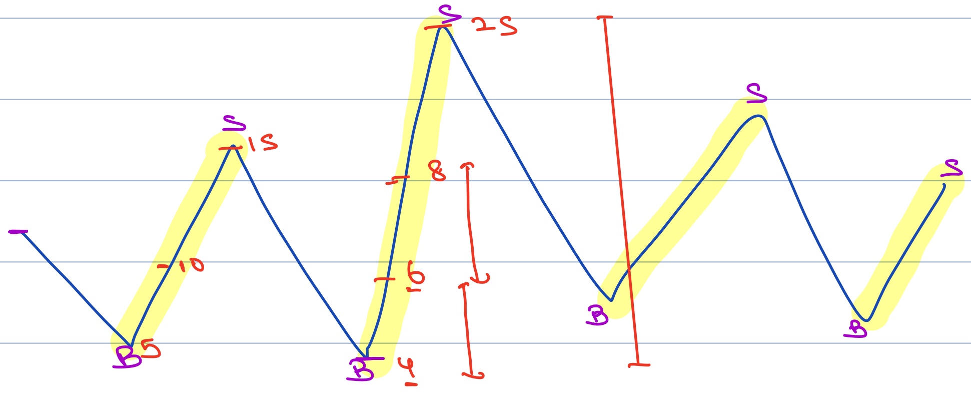
However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

$arr[\,] = [\overset{B}{\underset{0}{1}}, \overset{S}{\underset{\underset{B}{1}}{2}}, \underset{\underset{S}{2}}{3}]$

$2-1 = 1$
$3-2 = 1$
$\Rightarrow 2$

$arr[\,] = [5, \underset{B}{2}, \underset{S}{10}]$

$10-2 = \underline{8}$.



$25-4=21$

$6-4 = 2$
$8-6 = 2$
$25-8 = 17$
$\overline{21}$

10 - 5 = 5] → 10.
15 - 10 = 5

15 - 5 = 10.

```
public int maxProfit ( int[] prices) {
        int  ans = 0;
        for i → 1 to n-1
            if (prices[i] > prices[i-1]) {
                    ans += prices[i]-prices[i-1];
            }
        }
        return ans;
}
```

T.C → O(n)

S.C → O(1)

Shortest distance in a maze

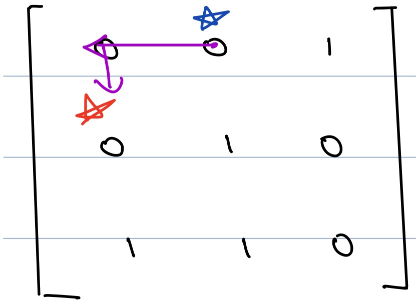
Given a matrix of integers A of size N x M describing a maze. The maze consists of empty locations and walls.

1 represents a wall in a matrix and 0 represents an empty location in a wall.

There is a ball trapped in a maze. The ball can go through empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall (maze boundary is also considered as a wall). When the ball stops, it could choose the next direction.

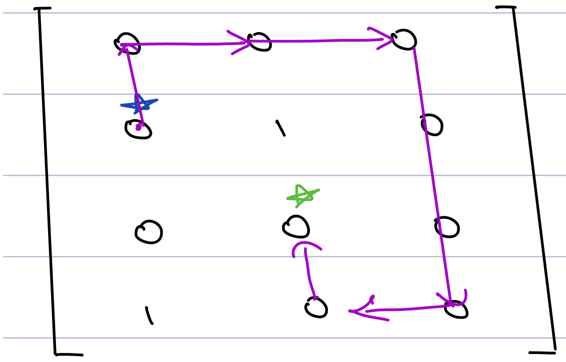
Given two array of integers of size B and C of size 2 denoting the starting and destination position of the ball.

Find the shortest distance for the ball to stop at the destination. The distance is defined by the number of empty spaces traveled by the ball from the starting position (excluded) to the destination (included). If the ball cannot stop at the destination, return -1.

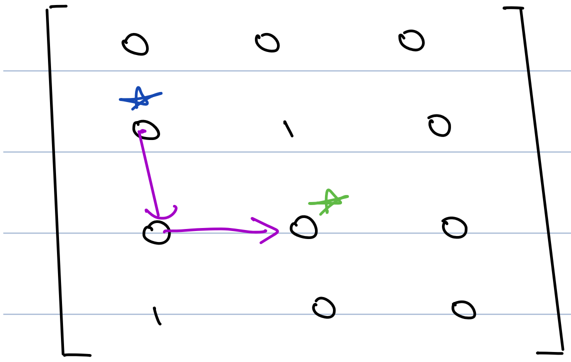


$$\begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

Ans => 2

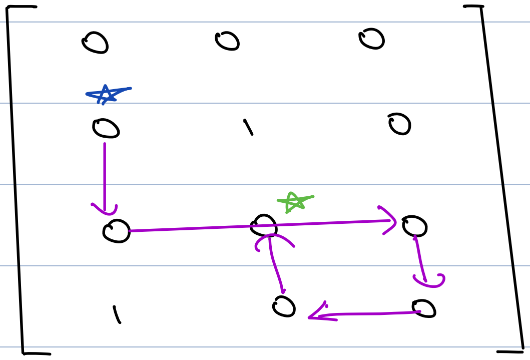


$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$



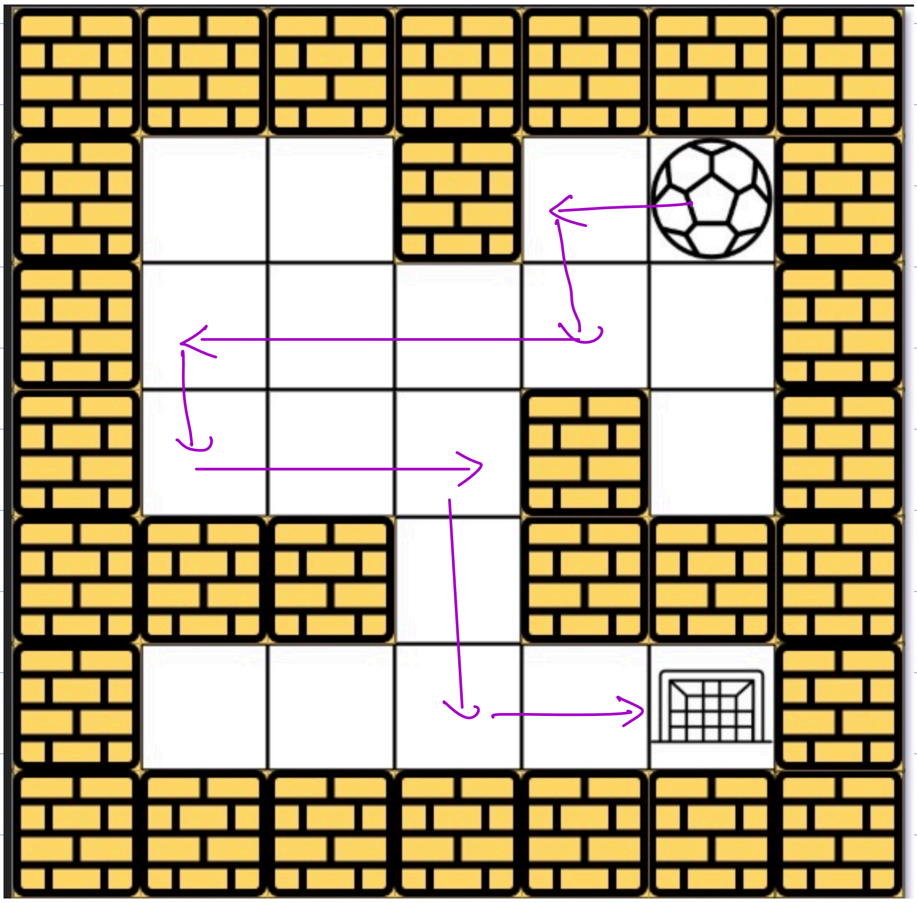
$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

X

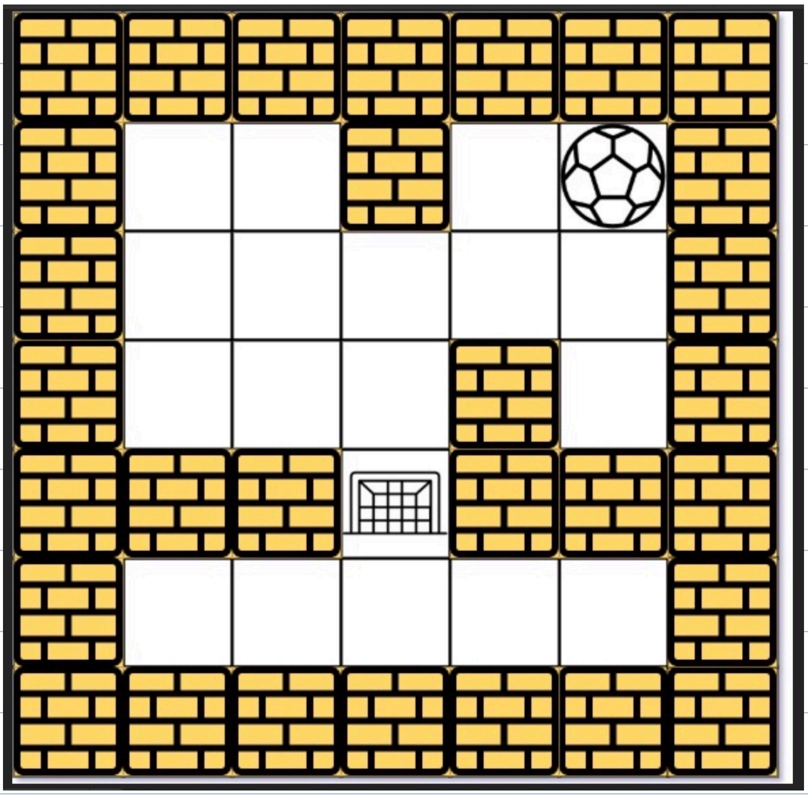


$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

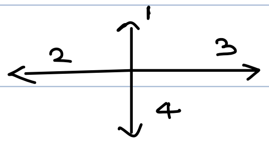
Ans = 6

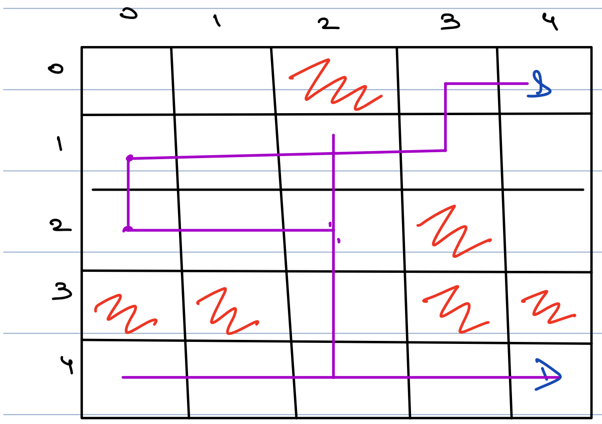


Ans = 12



Ans = -1 .

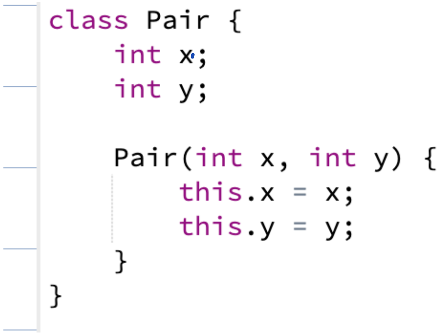




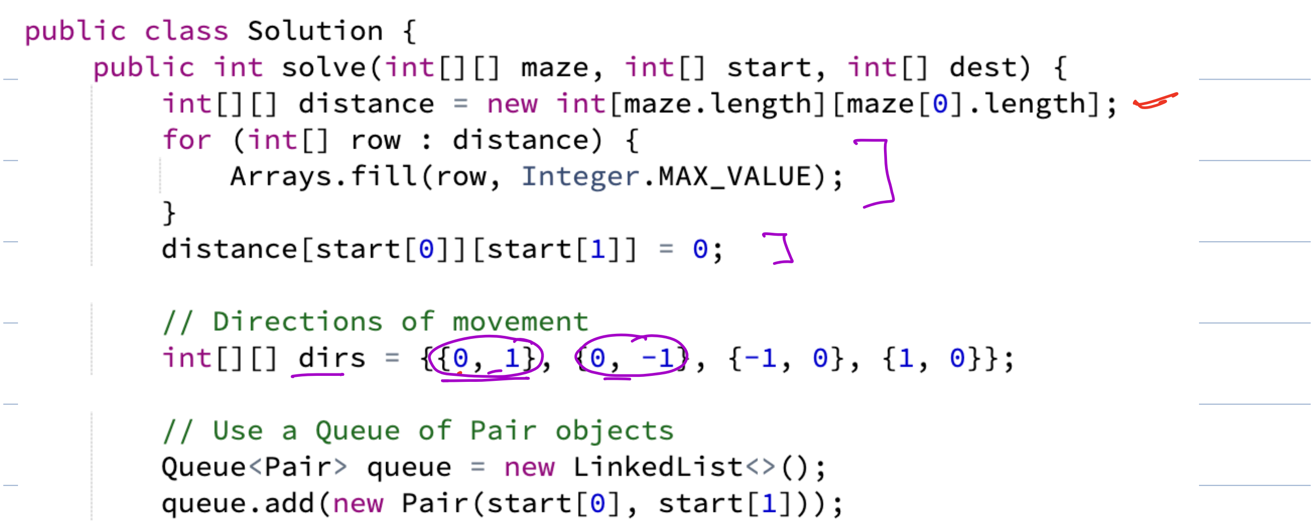
	0	1	2	3	4
0	6	7	∞	1	0
1	5	∞	9	2	3
2	6	9	8	∞	2
3	∞	∞	∞	∞	∞
4	12	∞	10	∞	12

~~(0,4)~~ ~~(0,3)~~ ~~(2,4)~~ ~~(1,3)~~ ~~(1,0)~~ ~~(1,4)~~ ~~(0,0)~~ ~~(2,0)~~

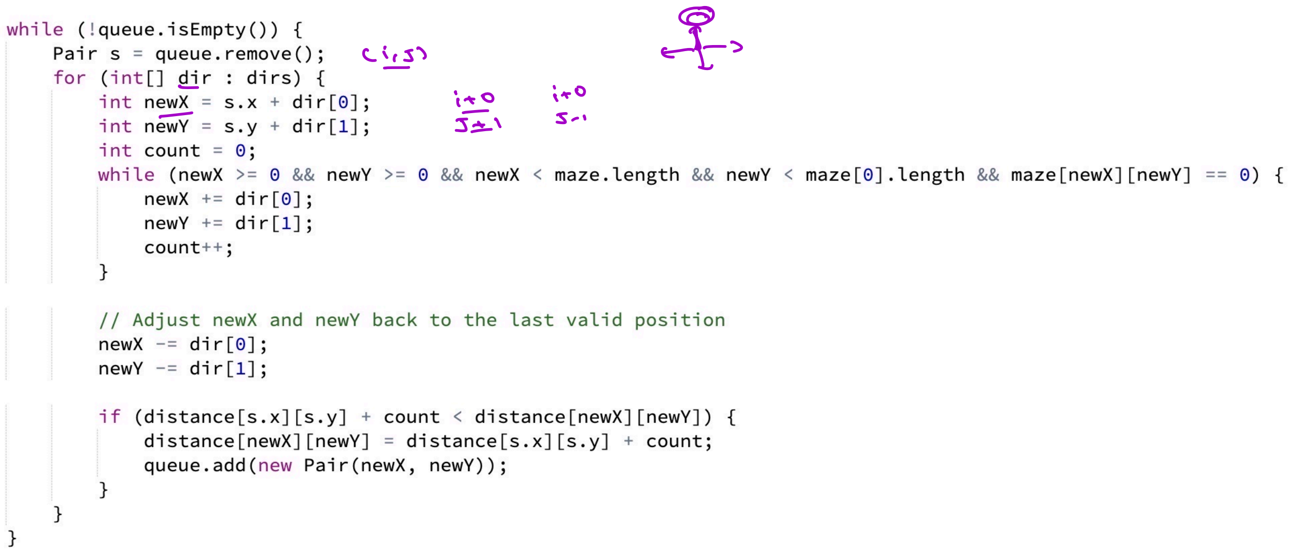
~~(0,1)~~ ~~(2,2)~~ ~~(2,1)~~ ~~(1,2)~~ ~~(4,2)~~ ~~(4,0)~~ ~~(4,4)~~



```java
class Pair {
    int x;
    int y;

    Pair(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```



```java
public class Solution {
    public int solve(int[][] maze, int[] start, int[] dest) {
        int[][] distance = new int[maze.length][maze[0].length];
        for (int[] row : distance) {
            Arrays.fill(row, Integer.MAX_VALUE);
        }
        distance[start[0]][start[1]] = 0;

        // Directions of movement
        int[][] dirs = {{0, 1}, {0, -1}, {-1, 0}, {1, 0}};

        // Use a Queue of Pair objects
        Queue<Pair> queue = new LinkedList<>();
        queue.add(new Pair(start[0], start[1]));
```

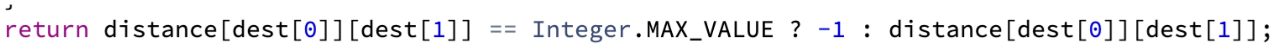


```java
while (!queue.isEmpty()) {
    Pair s = queue.remove();
    for (int[] dir : dirs) {
        int newX = s.x + dir[0];
        int newY = s.y + dir[1];
        int count = 0;
        while (newX >= 0 && newY >= 0 && newX < maze.length && newY < maze[0].length && maze[newX][newY] == 0) {
            newX += dir[0];
            newY += dir[1];
            count++;
        }

        // Adjust newX and newY back to the last valid position
        newX -= dir[0];
        newY -= dir[1];

        if (distance[s.x][s.y] + count < distance[newX][newY]) {
            distance[newX][newY] = distance[s.x][s.y] + count;
            queue.add(new Pair(newX, newY));
        }
    }
}
```

(i,j)

i+0
j+1

i+0
j-1



```java
return distance[dest[0]][dest[1]] == Integer.MAX_VALUE ? -1 : distance[dest[0]][dest[1]];
```

T.C → $O(m \times n)$

S.C → $O(m \times \underline{n})$,

Jump Game - 2

You are given an array of non-negative integers A, where each element in the array represents the maximum number of steps you can jump forward from that position.

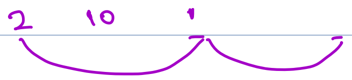
A jump means moving from the current index to another index further in the array within the allowed range of steps defined by the value at the current index. For instance, if A[i] = 3, you can jump to any of the next 3 positions from index i (i.e., i+1, i+2, or i+3), provided these positions are within the bounds of the array.

Your task is to determine the minimum number of jumps required to reach the last index of the array, starting from the first index.

If it's not possible to reach the last index, return -1.

e.g 1) $arr[\,] = \{\overset{\downarrow}{2}, 3, 1, 1, \overset{\downarrow}{4}\}$ Ans = 2

e.g 2) $arr[\,] = \{\underline{\overset{\downarrow}{0}}, 1, 1, \overset{\downarrow}{4}\}$ Ans = -1



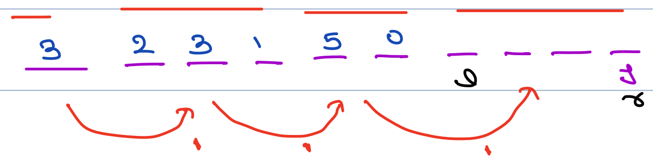
arr[] = {2, 3, 1, 1, 4}

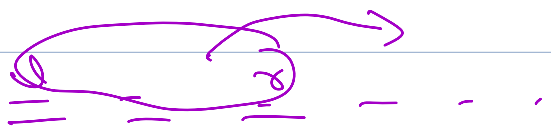
dp = 0 1 1 2 2

dp[i] = min jumps reqd to reach this idx, from start

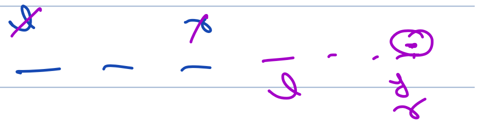
idx . → O(n²).

arr[] = {2, 3, 1, 1, 4}





```
func solve (int[] arr) {
    int l=0;
    int r=0;
    int count=0;
    while(r< n-1) {
        int max =0;
        for i -> l to r
            max = Math.max(max, i+arr[i])
        if( max <= r) { return -1}
        l=r+1
        r= max
        count++;
    }
    return count;
}
```



T.C -> O(n)
S.C -> O(1)