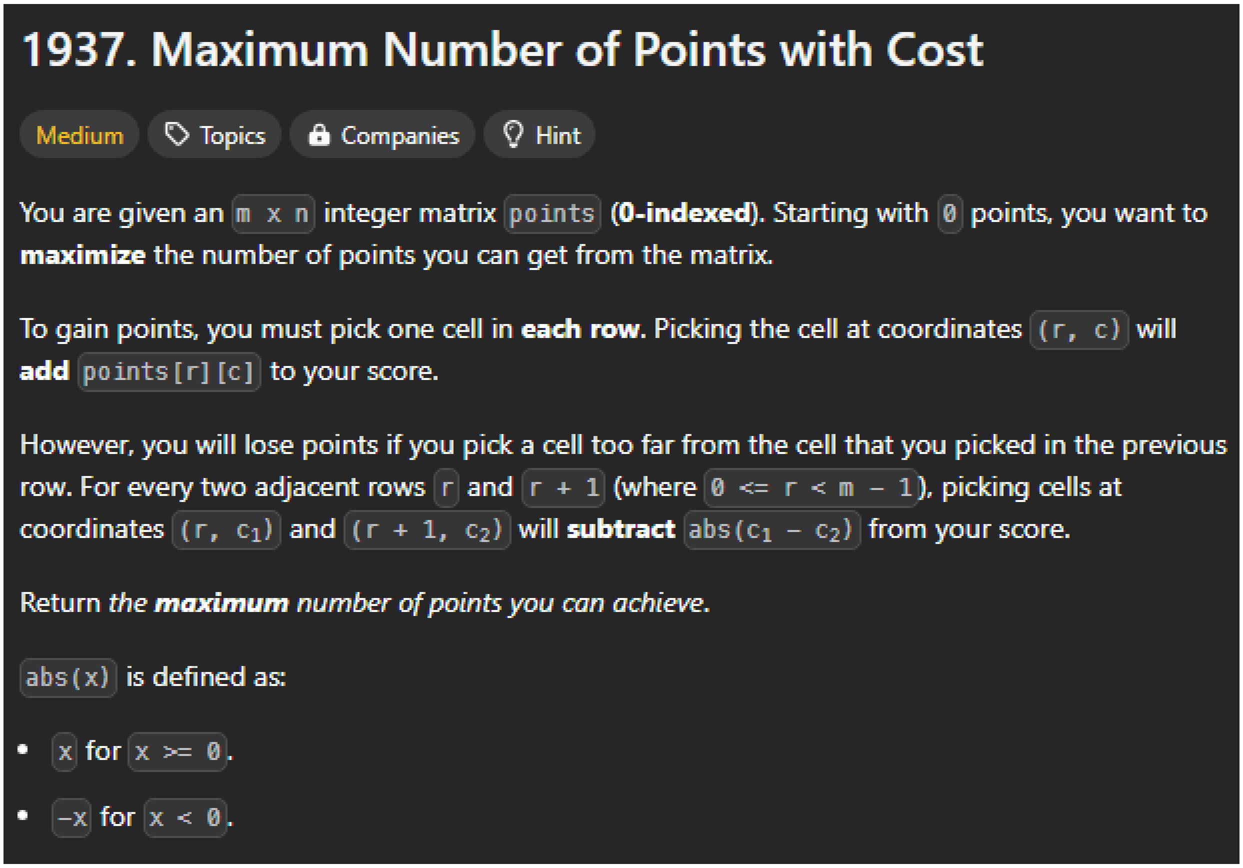
1937. Maximum Number of Points with Cost - 17/08/24 (Medium)

Dynamic Programming



1937. Maximum Number of Points with Cost

You are given an `m x n` integer matrix `points` (**0-indexed**). Starting with `0` points, you want to **maximize** the number of points you can get from the matrix.

To gain points, you must pick one cell in **each row**. Picking the cell at coordinates `(r, c)` will **add** `points[r][c]` to your score.

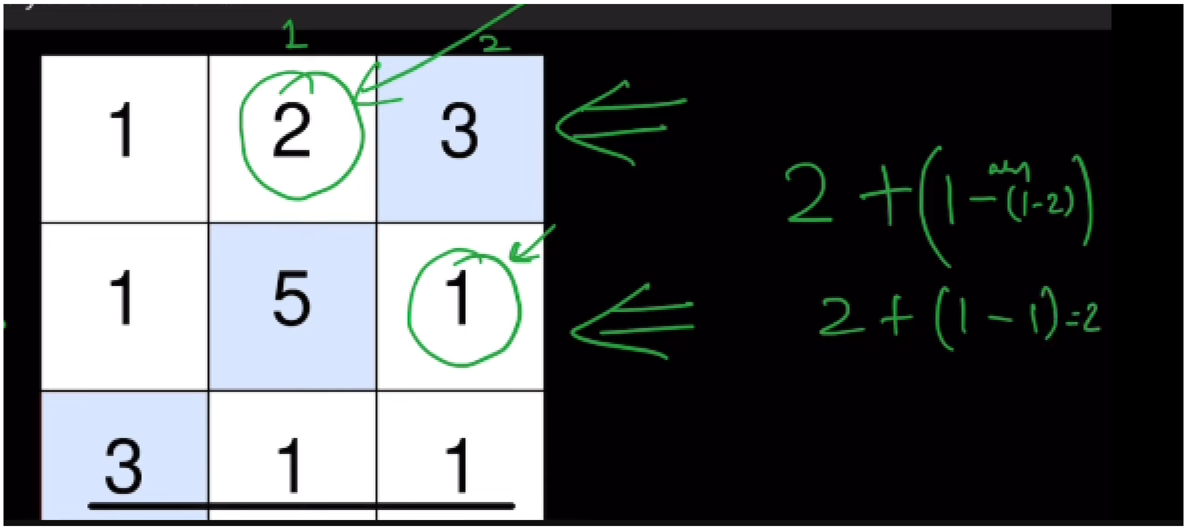
However, you will lose points if you pick a cell too far from the cell that you picked in the previous row. For every two adjacent rows `r` and `r + 1` (where `0 <= r < m - 1`), picking cells at coordinates `(r, c1)` and `(r + 1, c2)` will **subtract** `abs(c1 - c2)` from your score.

Return *the **maximum** number of points you can achieve.*

`abs(x)` is defined as:

- `x` for `x >= 0`.
- `-x` for `x < 0`.

you will loose points if you choose two far the first column

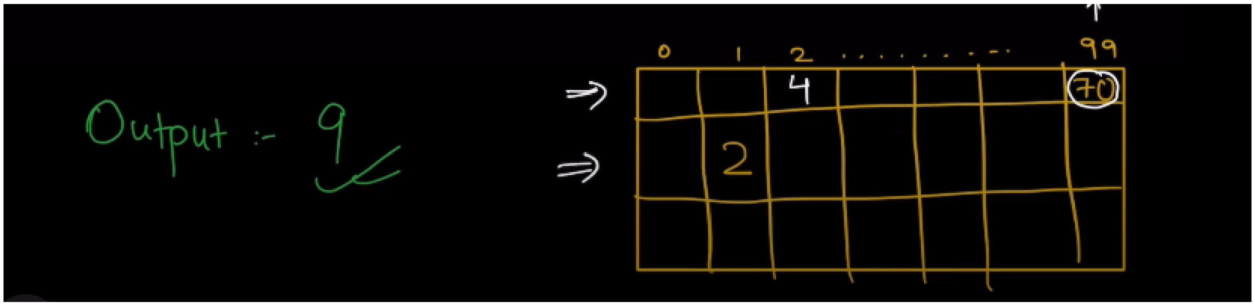


1	2	3
1	5	1
3	1	1

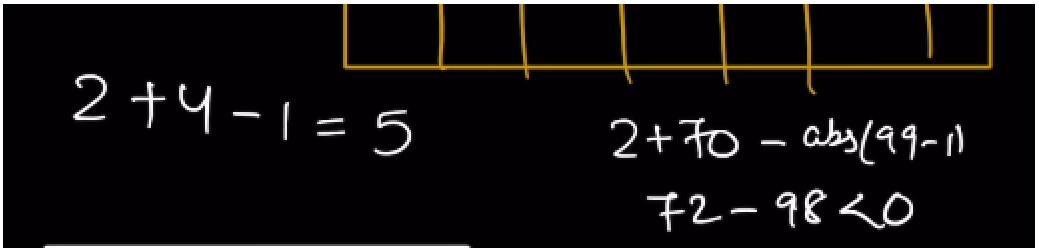
here

💡 how the calculation work
2 is at coulmn 1 row 0
1 is at column 2 row 1
2+(1- abs(column 1 vlaue -column 2 value)
2+(1- abs(1-2))
2+(1-1)
2

why we choose abs value with every step



if we were greedy then we could get negative values



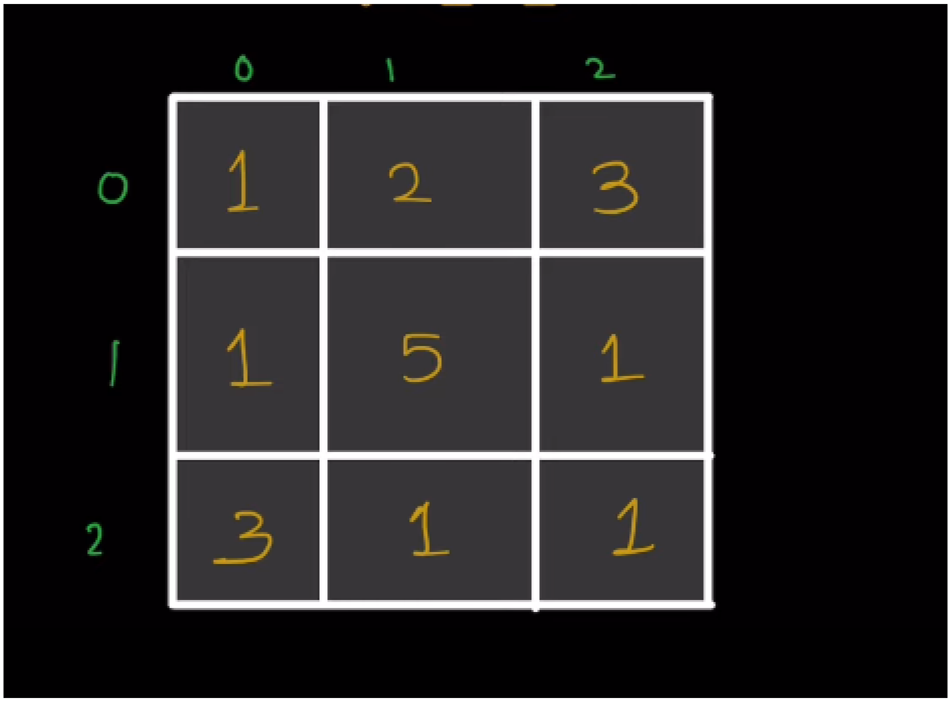
$2+4-1=5$

$2+70-\text{abs}(99-1)$

$72-98<0$

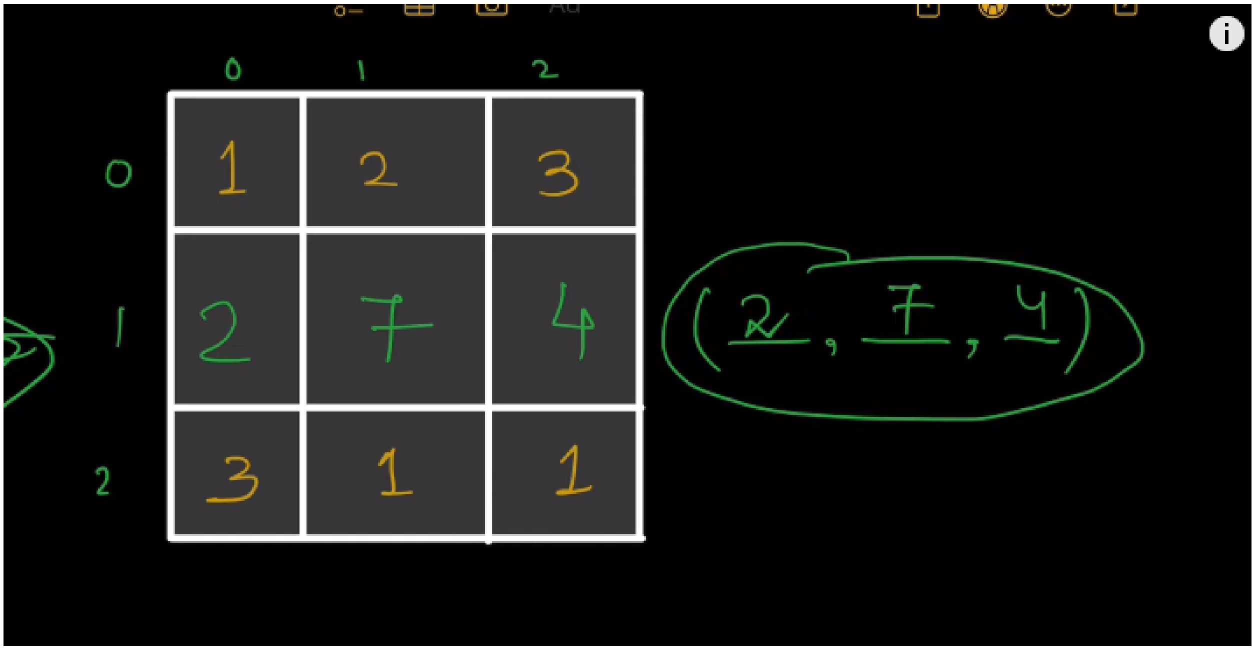
so what we will do here is that we will find maximum for each row by calculating alll posibnle value of adjancet rows of ti

like first in first row it remain as it is



	0	1	2
0	1	2	3
1	1	5	1
2	3	1	1

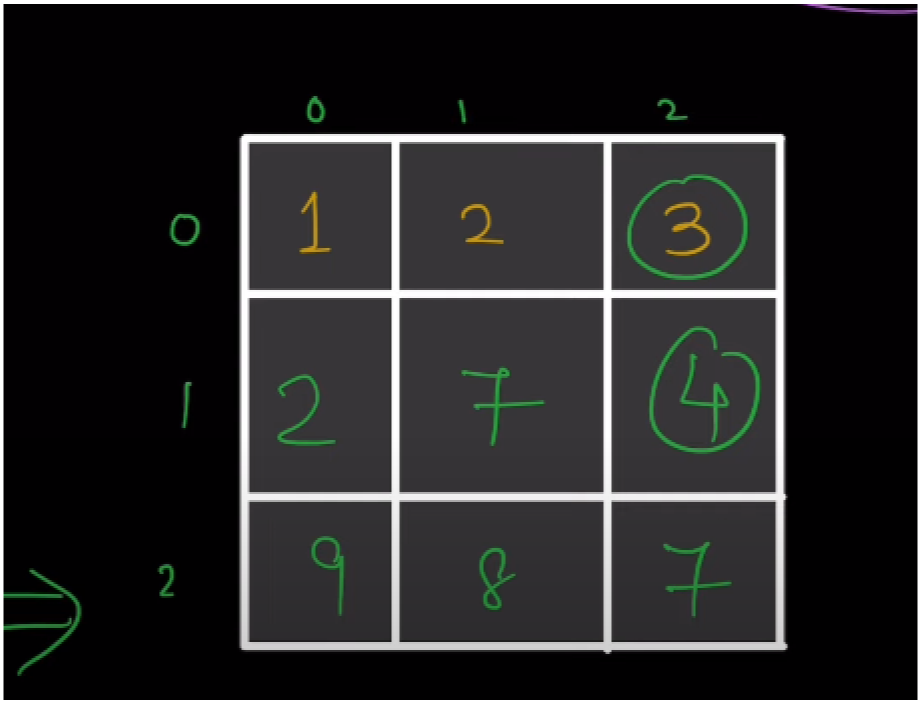
for second row



	0	1	2
0	1	2	3
1	2	7	4
2	3	1	1

(2, 7, 4)

for final row



	0	1	2
0	1	2	3
1	2	7	4
2	9	8	7

and the end we know that it the each row will have highest value is thje cell we will choose

it is brute force approach

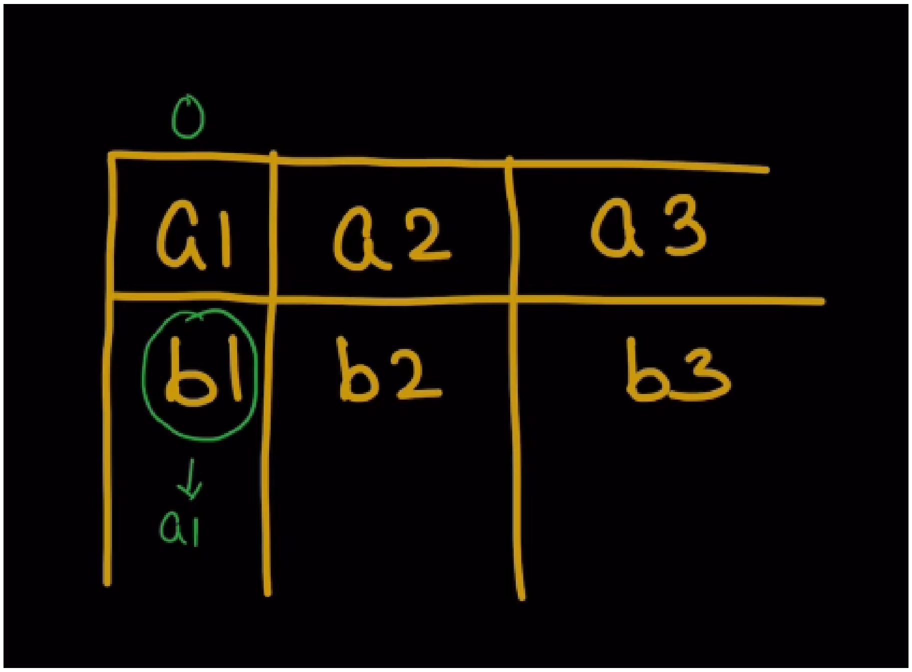
```
class Solution {
public:
    long long maxPoints(vector<vector<int>>& points) {
          int m = points.size();
        int n = points[0].size();
        vector<long long> prev(n);
        int score = 0;

        for(int col = 0; col < n; col++) {
            prev[col] = points[0][col];
        }

        for(int i = 1; i<m; i++) {
            vector<long long> curr(n);
            for(int j = 0; j<n; j++) {
                for(int k = 0; k < n; k++) {
                    curr[j] = max(curr[j], prev[k] + points[i][
                }
            }
            prev = curr;
        }
        return *max_element(prev.begin(), prev.end());
    }
};
```

but it will show a TLE error

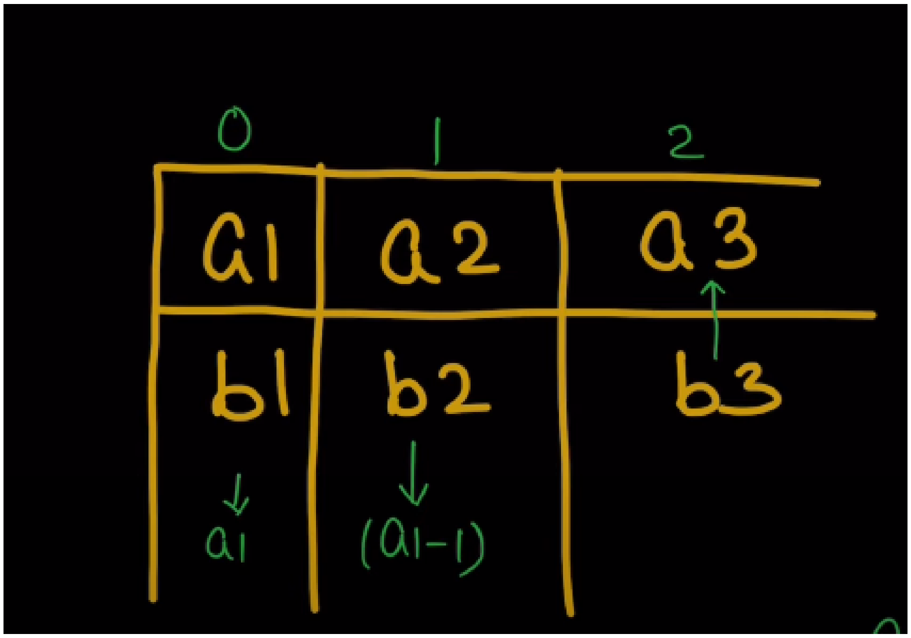
Now a Optimal Approach

in this method just chek above only and ask neighbor for thier best

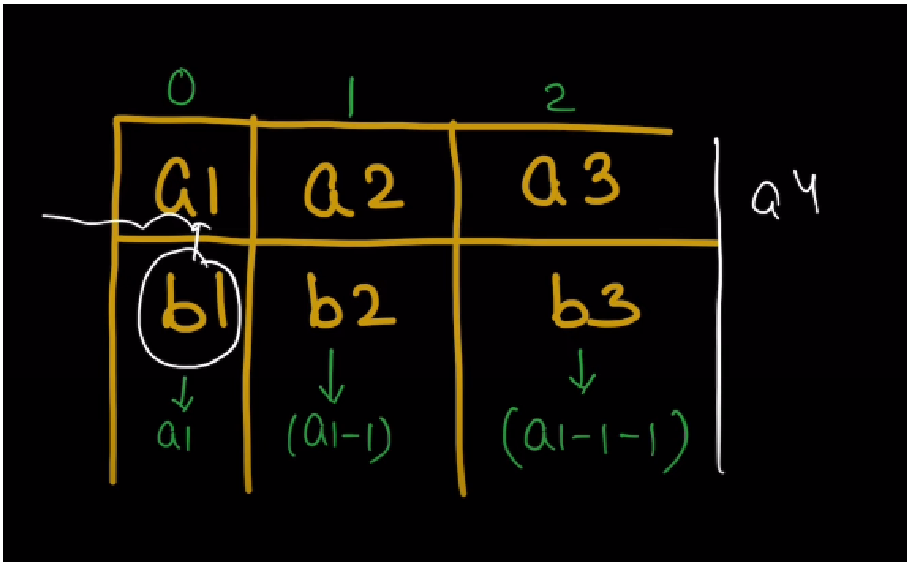
for first it will check b1 above a1 only



for b2 it will check above a2 and ask b1 for it best



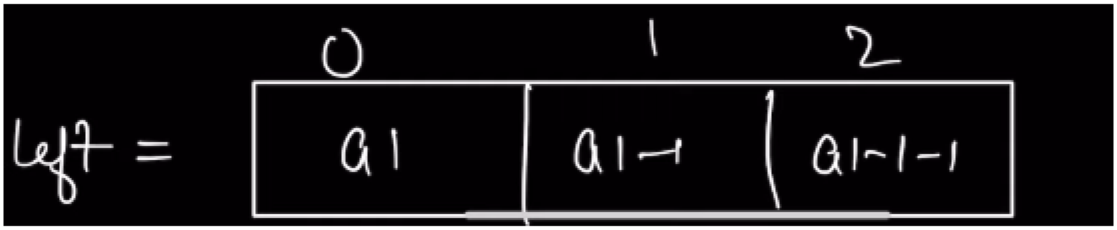
for b3 it will chekc a3 and ask b2 for it best



simple method check above and left side element ask your left neighbor

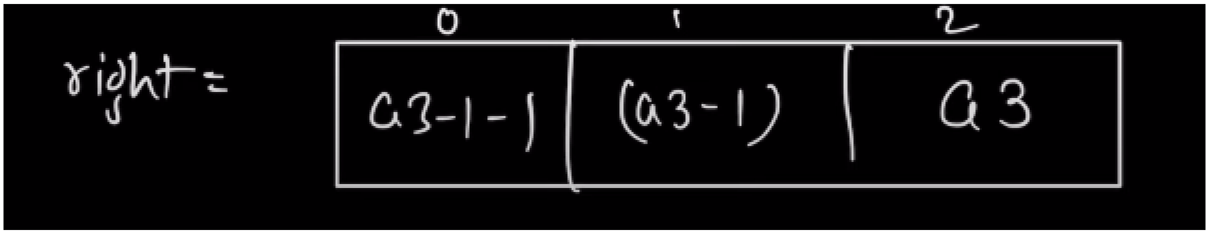
so we will make two different array : one a left array and right array

left will follow above method where check above and ask left neighbor



	0	1	2
left =	a1	a1-1	a1-1-1

while in right array
we will check above and right neighbor to get the best answer

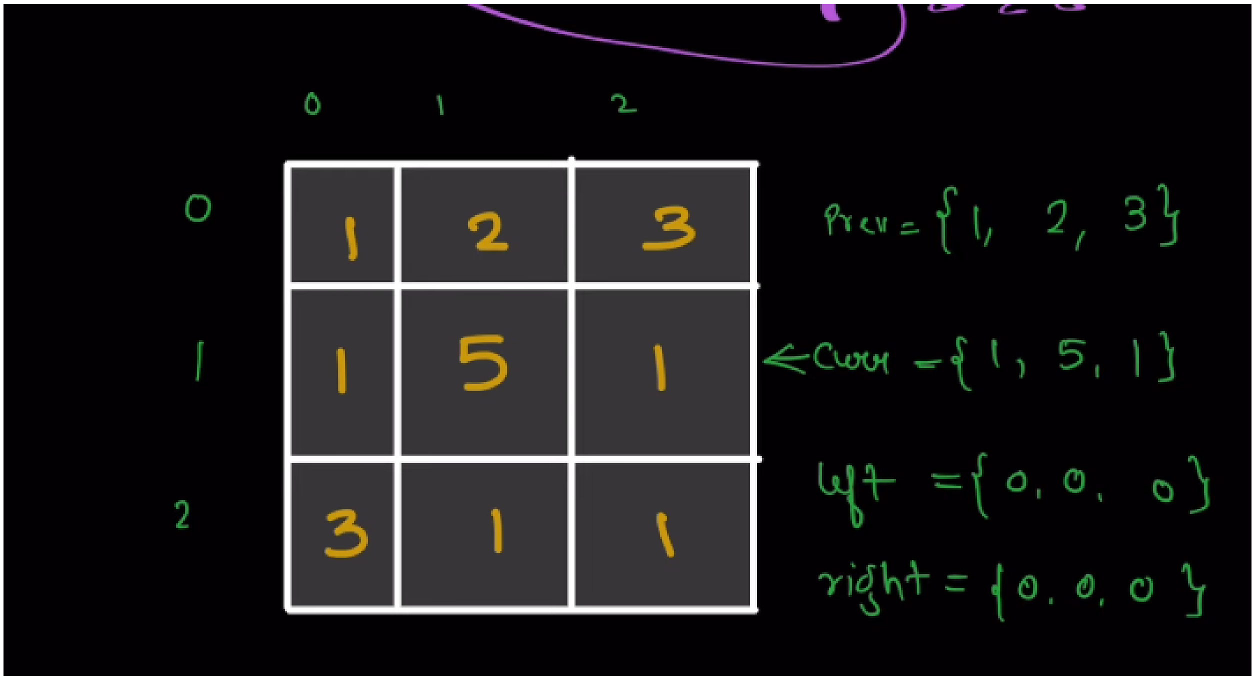


	0	1	2
right =	a3-1-1	(a3-1)	a3

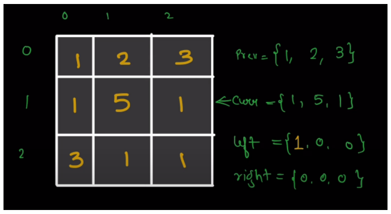
formula is b2+max(left[i] , right[i])

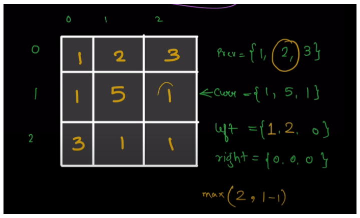
Now Solution example

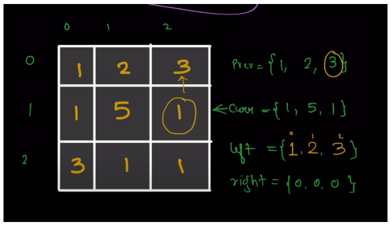
initially left and right ={0,0,0}



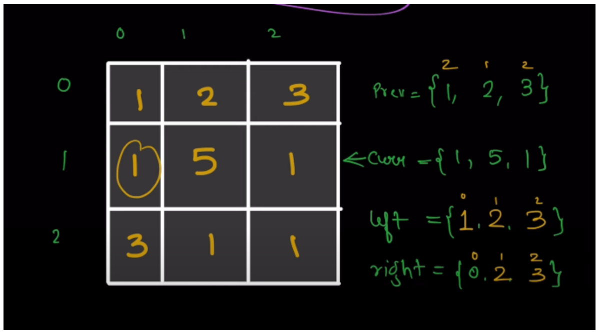
left

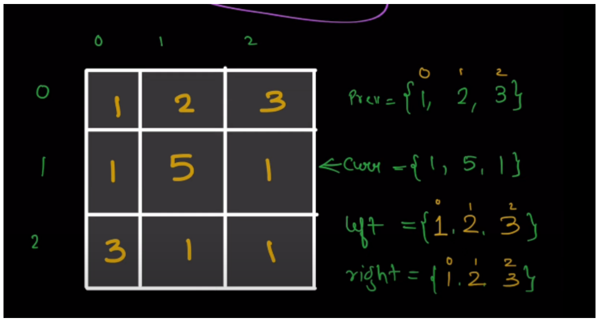




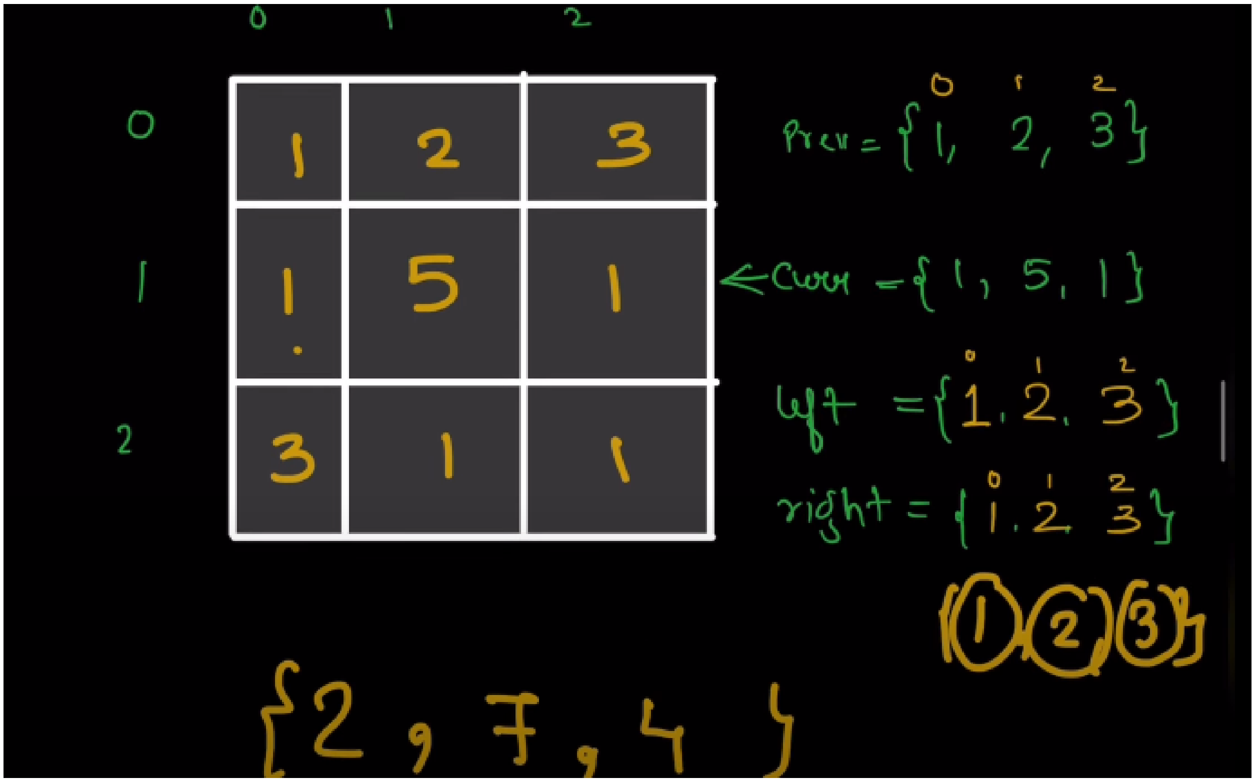


right





b1 +{1,2,3) ⇒ {2,7,4}



	0	1	2
0	1	2	3
1	1	5	1
2	3	1	1

optimal Answer

T.C = O(m*n)

```
class Solution {
public:
    long long maxPoints(vector<vector<int>>& points) {
        int m = points.size(), n = points[0].size();
        vector<long long> prev(n);
        int score = 0;

        for(int col = 0; col < n; col++) {
            prev[col] = points[0][col];
        }

        for(int i = 1; i<m; i++) {
            vector<long long> curr(n);
            auto left = curr, right = curr;

            //Fill left
            left[0] = prev[0];
            for(int j = 1; j<n; j++) {
                left[j] = max(prev[j], left[j-1]-1); // points[i
            }

            //Fill right
            right[n-1] = prev[n-1];
            for(int j = n-2; j >= 0; j--) {
                right[j] = max(prev[j], right[j+1]-1); // points
            }

            for(int j = 0; j<n; j++)
                curr[j] = points[i][j] + max(left[j], right[j])

            prev = curr;
        }
        return *max_element(prev.begin(), prev.end());
    }
};
```