Lecture 6: Vacation Problem (Atcoder) (2-D)

Sample n = 3

10	40	70
20	50	60
30	60	90

Ans → 210 (70 + 50 + 90)

Greedy fails because we will consider the maximum point / activity each day, respecting the condition that activity can't be performed consecutive days. Eg.

n = 2

10	50	1
10	100	11

According to greedy → 50 + 11 ⇒ 61

But the ans is 110 (10 + 100)

Where we have to try all possible choices as our next solution. We will use recursion to generate all the possible choices.

Steps to write recurrence

① Express the problem in terms of Indexes

One clear parameter which breaks the prob. in different steps is 'n' or 'number of Day'. So 'Days/n' can be expressed in terms of indexes.

At any day we have three activities (0, 1, 2) to be performed. We can only when we know which activity we choose the previous day. Thus we need another parameter 'last' in order to track the activity done yesterday.

Now there are three options each day (0, 1, 2) which becomes 'last' of the next day. If we are starting from (n-1) day, then for that day we can choose all three options by setting the value of last = 3.

Step-2) Try out all possible choices at a given index.

Our function has 2 parameters f(day, last)

We are writing a top-down recursive func^. We start from (n-1) day to 0th day. Thus our base case will be at (day == 0)

Other than base case, whenever we perform an activity it's points will be given by arr[day][i] and to points from other days, we will let recursion do it's job. f(day-1, i), Passing 'i' so that current day's activity becomes next day's last.

Step-3) Take Maximum of all choices.

Simply we take max of all sol^ provided by Recursion.

Memoization: There are over-lapping subprobs, notice if we make recursion tree. Follow the EXACT SAME steps for memoization as shown previously.

Code

```
int f(int n, int last, vector<vector<int>>&arr, vector<vector<int>>
                                                    &dp){
    if(dp[n][last] != -1) return dp[n][last];
    if(n==0){
        int maxi=0
        for(int i=0; i<3; i++){
            if(i != last){
                maxi = max(maxi, arr[n][i]);
            }
        return maxi;
    }
    int maxi = 0
    for(int i=0; i<3; i++){
        if(i != last){
            int curr = arr[n][i] + f(n-1, i, arr, dp);
            maxi = max(maxi, curr);
        }
    }
    return dp[n][last] = maxi;
}
```

TC → O(N×4×3)

There are (N×4) states and for every state we are running an iterative loop (3) times.

SC → O(N) + O(N×4) ≈ O(N)

↙ Recursion Stack ↘ 2D array

Tabulation

① Declare a dp[] of size [n][4] i.e dp[n][4]
② We initialize the base case, and we know the base case is when day=0, Thus.

```
dp[0][0] = max(arr[0][1], arr[0][2])
dp[0][1] = max(arr[0][0], arr[0][2])
dp[0][2] = max(arr[0][0], arr[0][1])
dp[0][3] = max(arr[0][0], arr[0][1], arr[0][2])
```

③ Set an iterative loop from 1 to n and for every index set its value according to recursive logic.

Code

```
int f(int n, vector<vector<int>> arr) {
    vector<vector<int>> dp(n, vector<int>(4,0));
    dp[0][0] = max(arr[0][1], arr[0][2]);
    dp[0][1] = max(arr[0][0], arr[0][2]);
    dp[0][2] = max(arr[0][0], arr[0][1]);
    dp[0][3] = max(arr[0][0], max(arr[0][1], arr[0][2]));

    for(int day=1; day<n; day++) {
        for(int last=0; last<4; last++) {
            dp[day][last] = 0;
            for(int task=0; task<3; task++) {
                int pts = arr[day][task] + dp[day-1][task];
                dp[day][task] = max(pts, dp[day][task]);
            }
        }
    }
}
```

T.C. $\longrightarrow$ $O(N*4*3)$

S.C. $\longrightarrow$ $O(N*4)$

Space Optimization.

If we look closely at the relation,

$dp[day][last] = max(dp[day][last], points[day][task] + dp[day-1][task])$

Here the task can be anything from 0 to 3 and day − 1 is the previous stage of recursion. So, in order to compute any dp array value, we only require the last row to calculate it.

day \ last	0	1	2	3
0	✓	✓	✓	✓
1				
⋮				
n-1				

This row is initially filled.

This row only needs previous row values.

* So rather than storing the entire 2-D Arrays of size $N*4$, we can just store values of size 4 (say prev).
* We can take dummy array, again of size 4 (temp) and calculate the next row's value using the array we stored in step 1.
* After that whatever we move to the next day, the temp array becomes our prev for the next step.

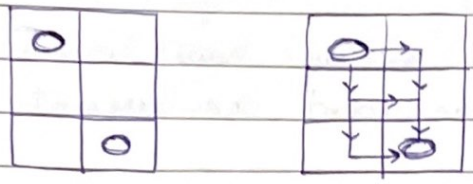
(*) At last of prev[3] will give us the soln.

```
int f(int n, vector<vector<int>>& points)
{

    vector<int> prev(4,0);

    prev[0] = max(points[0][1], points[0][2]);
    prev[1] = max(points[0][0], points[0][2]);
    prev[2] = max(points[0][0], points[0][1]);
    prev[3] = max(points[0][0], max(points[0][1], points[0][2]));

    for(int day = 1; day < n; day++) {
        vector<int> temp(4,0);
        for(int last = 0; last < 4; last++) {
            temp[last]
            for(int task=0; task<=2; task++) {
                if(task != last) {
                    temp[last] = max(temp[last],
                        points[day][task] + prev[task];
                }
            }
        }
    }
    prev = temp;
}

return prev[3];
}
```

Very Imp Ques^n to understand all types of Grid Probs.

Lecture : 7 → Unique Paths (Leetcode) (DP on Grids).

Sample : m = 3 , n = 2.



output - 3

Steps to form Recurrence :-

Step 1) Express the problem interms of indexes.

We are given two variable M and N, representing the dimension of a 2D matrix. for every different prob, this M & N will change.

We can define the function with two parameters i & j, where i & j represent the row and column of the matrix

∴ f(i, j) = Total amount of unique paths from matrix[0][0] to matrix[i][j]

↳ Basically it gives ans for the subproblem.

⊙ We will be doing Top-down recursion, i.e we will move from [M-1][N-1] and try to find our way to the cell [0][0].
Therefore at every index, we will try to move up and towards left.
(opposite to Ques^n because we are going down to top)

Base Case

There are two base Cases:-

① When $i=0$ & $j=0$ that is we have reached our destination so we can count the current path hence we return 1.

② When $i<0$ (or) $j<0$, it we mean we went ~~of~~ out of the matrix and couldn't find right path, hence return 0.

Step2) Try out all possible choices.

We have two choices, either we could go up (↑) (or) we could go left (←), by reducing row no. & column no. respectively.

Step3) Take maximum of all choices.

We have to count all possible paths, we will return. Sum of all choices (up & left)

```
f(i,j) {
    if (i==0 && j==0) return 1;
    if (i<0 || j<0) return 0;

    up = f(i-1, j);
    left = f(i, j-1);

    return up + left;
}
```

Memoization.

- Declare dp[M][N] = -1.
- Store all the states in dp, as.
  dp[M][N] = up + left.
- Check if dp[M][N] != -1, then return the calc value.

Time Complexity ⟶ O(N x M).

Space Complexity ⟶ O((N-1) + (M-1)) + O(M * N).

O((N-1) + (M-1)) → Recursion Stack Space. here, (N-1) + (M-1) is max path length

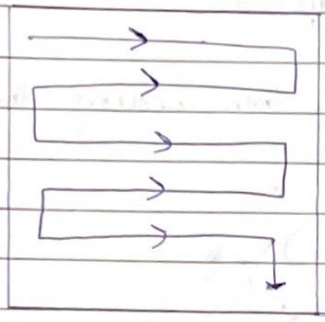
O(M * N) → DP array.

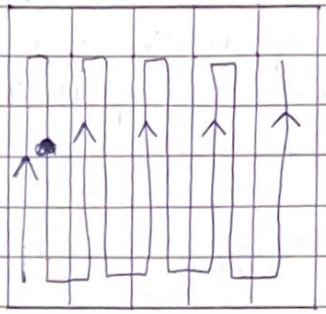
Tabulation.

Steps to Convert Recursion to tabulation

- Declare dp[][] array of siz [M][N].
- First initialize the base condition i.e dp[0][0] = 1.
- Our answer should get stored in dp[M-1][N-1]. We want to move from (0,0) to (M-1, N-1). But we can't move arbitrarily, we should move such that at a particular i & j, we have all the values required to compute dp[i][j].
- If we see the memoized code, value required for dp[i][j] are : dp[i-1][j] & dp[i][j-1]. So we only use the previous row & column value.
- We have already filled the top-left corner, if we move in any of the two ways (next page), at every

cell we do have all the previous values required to compute its value.





- We can use two Nested loops to have this traversal.
- At every cell we calculate up & left as we had done in the recursive solution and then assign the cell's value as (up + left)

Code

```
int f(int m, int n, vector<vector<int>> &dp) {

    for(int i=0; i<m; i++) {
        for(int j=0; j<n; j++) {
            if(i==0 && j==0) dp[i][j] = 1;
            else {
                int left, up = 0;
                if(i>0)
                    up = dp[i-1][j];
                if(j>0)
                    left = dp[i][j-1];
                dp[i][j] = up + left;
            }
        }
    }
    return dp[m-1][n-1];
}
```

Time Complexity — O(M * N).

Space Complexity — O(M * N)

Space Optimization.

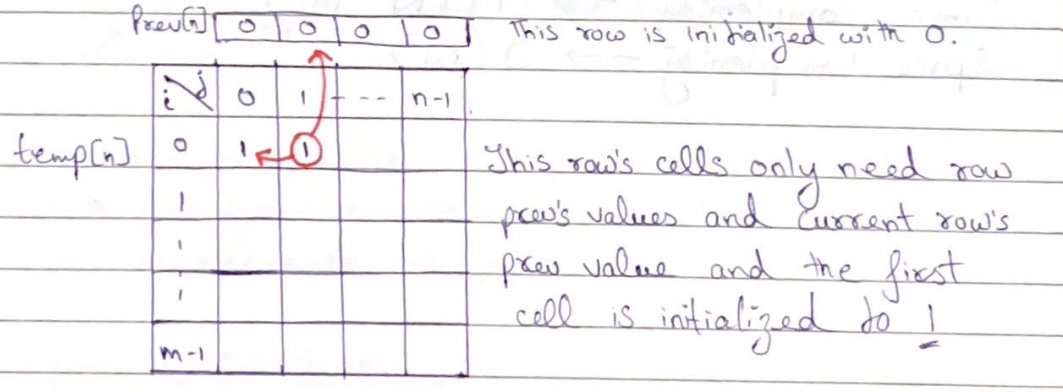
if we look closely the relation;

$$dp[i][j] = dp[i-1][j] + dp[i][j-1]$$

We can see we only need the previous row & column to compute dp[i][j]. Thus we can optimize even further.

Initially, we take a temp row and initialize it with 0.

Now the current row only needs the prev. row value and current row's prev. column value to compute dp[i][j].



At the next step, the temp[] becomes the prev of the step and using its values we can calc. the next row's values. At last prev[n-1] will give answer.

```
int f(int m, int n) {

    vector<int> prev(n,0);
    for(int i=0; i<m; i++) {
        vector<int> temp(n,0);
        for(int j=0; j<n; j++) {
            if(i==0 && j==0) temp[j]=1;
            Else {
                int up=0, left=0;
                if(i>0)
                    up= prev[j];
                if(j>0)
                    left = temp[j-1];
                temp[j]= ~~prev~~ up + left;
            }
        }
        prev = temp;
    }
    return prev[n-1];
}
```

Time Complexity → O(N * M)
Space Complexity → O(N).

Lecture 8: Minimum Path Sum (Leetcode) (DP on Grid)

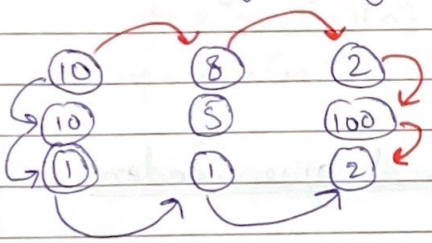
Problem Statement

We are given an `N x M` matrix with values and we ~~the~~ have to calculate the minimum sum / path to travel from top left to Bottom Right.
Note : We can move only in Rightward and downward motion.

Solution

This question is slight modification of prev problem.

Why greedy fails?
If we try greedy, then at any position we have two options, either go right or go down. Now let's take the following Eg:-



This is Greedy Sol^n i.e 122

This is the ans i.e 24

Thus we have to generate all possible paths and see which path will give us minimum sum. Thus we use Recursion to Generate all paths.

Steps to form Recurrence :-

(1) Express the Problem in terms of Indexes -
We can define a func^n with two parameters i & j where i and j represent the row and column

of the matrix.

$f(i,j) \rightarrow$ Minimum path Sum from Matrix[0][0] to matrix[i][j].

Note :- We will be doing top-down Recursion, we will move from cell [M-1][N-1] and try to find our way to the cell [0][0]. Therefore at every index, we will try to move up and towards the left.

Base Case

(1) When i=0 (&) j=0, hence we have reached our final destination and we have to add it up in the soln.

(2) If i<0 (OR) j<0, hence we have gone out of bounds and thus we need to return a big value (1e9) so as to reject this path.

(2) Try out all possibles choices at given Index

Since we are writting top-down Recursion, we reduce 'i' by 1 to go up (OR) we can reduce 'j' to left. And we also add the current index value too.

(3) Taking Minimum of all Choices

As per the question, we return the minimum as answer

Memoization.

Same → Declare 2D dp array of size M×N Store the minimum result/overlapping subproblems and return it whenever possible.

Code

```
int f(int m, int n, vector<vector<int>>& dp,
                    vector<vector<int>>& arr)
{
    if(m==0 && n==0) return arr[m][n];
    if(m<0 || n<0) return INT_MAX;
    if(dp[m][n] != -1) return dp[m][n];

    int left = INT_MAX, up = INT_MAX;

    left = arr[m][n] + f(m, n-1, dp, arr);
    up = arr[m][n] + f(m-1, n, dp, arr);

    return dp[m][n] = min(up, left);
}
```

Time Comp → O(N * M).
Space Comp → O((M-1)+(N-1)) + O(N * M).

Tabulation

Same :- Completely Same as prev problem just update the if/else condition acc. to ques^n and Memoization as shown above.

Code

```
int f(int m, int n, vector<vector<int>>& arr)
{
    vector<vector<int>> dp(m, vector(n, -1));

    for(int i=0; i<m; i++) {
        for(int j=0; j<n; j++) {
            if(i==0 & j==0) dp[i][j] = arr[i][j];
            else {
                int left, up = ~~maar~~ INT_MAX;
                if(i>0) up = arr[i][j] + dp[i-1][j];
                if(j>0) left = arr[i][j] + dp[i][j-1];

                dp[i][j] = min(up, left);
            }
        }
    }
    return dp[m-1][n-1];
}
```

Time Comp - O(N*M)
Space Comp - O(N*M).

Space Optimization.

It also Complety same, take two array prev and temp.
One for prev row and one for current columns.

Time Comp - O(N * M)
Space Comp - O(N).

Path

Lecture :9 – Minimum Sum In Triangle Grid (Triangle) (LeetCode)

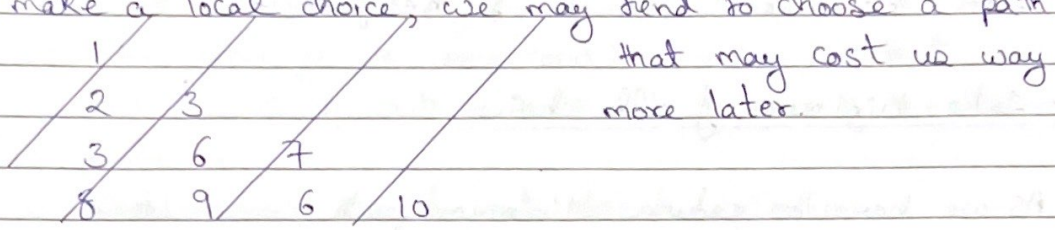
Problem

Given a Triangular Matrix. We have to calculate the minimum path sum to reach the bottom-most row.

Solution.

It is a slight modification of the prev problem

Greedy fails here too, as we know that whenever we make a local choice, we may tend to choose a path that may cost us way more later.



Steps to form the Recursive Solution :-

① Express the problem in terms of Indexes.

Our Ultimate aim is such that we reach the last row. We can defin f(i,j) such that it gives us the minimum path sum from the cell [i][j] to last row.

f(i,j) → Minimum Path Sum from matrix [0][0] to the last row of matrix.

We want to find f(0,0) and return it as our answer

Base Case :- when i == N-1, that is when we have reached last row, so the min path from that cell to the last row will be the value of cell itself, hence return mat[i][j].

Since we have only Two options, to go either down (or) down right. Observe that we'll never go out of bounds Therefore only one base case if sufficient.

(2) Try all possible choices.

Since we have 2 options, we can either increase (i) by 1 to go down (or) increase both (i) & (j) by 1 to go down-right. Make them recursive functions to get answer Add the current cell too, to the function call.

(3) Take Minimum of all choices

As we have to return Minimum path, we store the minimum of down & diagnol.

Memoization.

Same as all prev. Problems :-

1. Declare a dp[] of size (N)(M).
2. Store minimum of down & diagnol.
3. Return dp values whenever they are not (-1).

Code

```
int f(int i, int j, vector<vector<int>>& dp, vector<vector<int>>& arr)
{
    if(i==n-1) return arr[i][j];
    if(dp[i][j]!=-1) return dp[i][j];

    int down = arr[i][j] + f(i+1, j, dp, arr);
    int diagnol = arr[i][j] + f(i+1, j+1, dp, arr);
    return dp[i][j] = min(down, diagnol);
}
```

Time Comp → $O(N * N)$.

Space Comp → $O(N) + O(N * N)$.

Tabulation.

As in Recursion / Memoization, we moved from 0 to N-1, in tabulation we move from (N-1) to 0, i.e last row to first.

- Declare DP[][] of size [N][N].
- First initialize the base conditions value, i.e ~~that~~ the last row of dp matrix to the last row of the triangle matrix.
- Our answer should get stored dp[0][0].
- Since we have already filled values from (N-1) row we'll start from (N-2) and move upwards.
- Use 2 Nested loops.

Code

```
int f(vector<~~int~~vector<int>>& arr, int n){
    vector<vector<int>> dp(n, vector<int>(n,0));
    for(int i=0; i<N; i++) dp[n-1][i]= arr[n-1][i];

    for(int i=n-2; i>=0; i--){
        for(int j=i; j>=0; j--){

            int down = arr[i][j] + dp[i+1][j];
            int dignol= arr[i][j] + dp[i+1][j+1];
            dp[i][j]= min(down, diagnol);
        }
    }
    return dp[0][0];
}
```

Time Comp — O(N*N)
Space Comp — O(N*N)

Space Optimization

Observe at the relation.

dp[i][j] = matrix[i][j] + min(dp[i+1][j], dp[i+1][j+1]);

We only need the next row, to calc dp[i][j].
Thus we can optimize it.

Take a dummy [] (say front) and initialize it with triangle [] last row as done in tabulation.

Now the current row (say curr) only needs front row to calc dp[i][j].

Code

```
int f(vector<vector<int>>& arr, int n) {
    vector<int> front(n,0), curr(n,0);
    for(int i=0; i<n; i++)
        front[i] = arr[n-1][i];
    for(int i=n-2; i>=0; i--) {
        for(int j=i; j>=0; j--) {
            int down = arr[i][j] + front[j];
            int diagnol = arr[i][j] + front[j+1];
            curr[j] = min(down, diagnol);
        }
        front = curr;
    }
    return front[0];
}
```

Time Comp = O(N*N)
Space Comp = O(N).

Lecture 10 - Minimum falling Path Sum (leetcode)
(Variable Starting and Ending pt)

We are given an 'N * N' matrix and we need to find the minimum sum path from first row to last row.
We can move in Three Direction → Down.
Bottom Left
Bottom Right

Solution.
This problem is very similiar to previous problems

The Recurrence
f(i,j) → Minimum path Sum from the first row to cell[i][j].

Since, we are computing from last row to our way up (first row).
We take every element in the last row and compute the minimum path sum and return the smallest answer out of all of them.

Base Case

① When i==0 that mean we have reached the first row, and we just simply want to return the Element we landed on.
return arr[0][j];

② At very cell we have three options (we are writing ~~top-down~~ Bottom-up). up, upleft & upright.

for going (up) there is no problem because we'll reach top row and return the value as it is.

But for top-left & top-Right, we can go out of bounds. To prevent that we return INT_Max in order to ignore that path.

Performing Tasks & Computing Minimum

We'll make 3 Recursive calls for up, upleft & upright, compute them. Take the minimum of the three choices and store it in a dp[][].
And return dp.

Code

```
int f(int r, int c, vector<vector<int>>& dp, vector<vector<int>>& arr)
{
    if(c<0 || c>=arr.size()) return INT_MAX;
    if(r==0) return arr[0][c];
    if(dp[r][c]!=-1) return dp[r][c];
    int up = arr[r][c] + f(r-1, c, dp, arr);
    int upleft = arr[r][c] + f(r-1, c-1, dp, arr);
    int upright = arr[r][c] + f(r-1, c+1, dp, arr);

    return dp[r][c] = min(up, min(upleft, upright));
}
int main {
    int mini;
    for(int i=0; i<r; i++) {
        mini = min(mini, f(r, i, dp, arr));
    }
    return mini; }
```

~~Tabulation~~ Time Comp → O(N*N).

Space Comp → O(N) + O(N*N)

Tabulation.

1. Declare dp array size [N][N]
2. first initialize base conditions, the first row of the dp array to the first row of input matrix
3. If we look at the memoized code, values required to compute dp[i][j] is...

   dp[i][j] = dp[i-1][j] , dp[i-1][j-1] & dp[i-1][j+1].
   Thus we need values from (i-1) row.
4. Since we have filled the (i=0) row, we start from i=1 and move downwards.
5. Use 2 nested loops for traversal.

Time Comp — O(N*N).
Space Comp — O(N*N).

PTO

Code

```
int f (vector<vector<int>> arr)
{
    int n, m = ~~matrix~~si arr.size();
    vector<vector<int>> dp(n, vector<int>(m, 0));

    for(int i=0 ; i<m; i++)
        dp[0][i] = arr[0][i];

    for(int i=1 ; i<n; i++) {
        for(int j=0; j<m ; j++) {
            int up, upleft, upright= maxi;
            up= arr[i][j] + dp[i-1][j];
            if (j-1>=0)
                upleft= arr[i][j] + dp[i-1][j-1];
            if (j+1<m)
                upright = arr[i][j]+ dp[i-1][j+1];

            dp[i][j]= min(up, min(upleft, upright));
        }
    }
    int ans = INT_MAX;
    for(int i=0; i<m; i++) {
        ans = min(ans, dp[n-1][i]);
    }
    return ans;
}
```

Space Optimization

If we look closely,

$dp[i][j] = arr[i][j] + \min(dp[i-1][j], \min(dp[i-1][j-1], dp[i-1][j+1]))$;

We can see that we only need previous row, like we required in previous problems.

Initially we can take dummy [] (prev). Initialise it with input matrix first row.

Use the another curr[] and prev[] to calculate dp[i][j].

Lecture 11 - Cherry pickup 2 (3D DP) (DP on Grids).

Problem Statement:

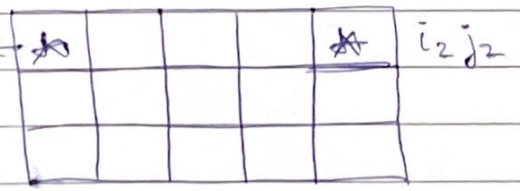
We are given 'N * M' matrix. Every cell has some cherries ~~it~~ in it.
We are given 2 users. One is standing on top left row (0, 0) and the other is standing on top right (0, M-1).
They can travel in any direction (↙ ↓ ↘) to reach the last row.
Find the maximum no. of cherries collected by both of them combined
Note If both land on the same cell then it will be counted only once.

Solution.

This question is clearly an example of ~~variable~~ fixed starting point and ~~fixed~~ variable ending point.
As we have done previously, it is recommended to solve it in top-Down Recursion fashion.

Recurrence.

(1) We need four parameters to describe the position of both users
(i_1, j_1, i_2, j_2).



If we observe, both users are at the same row and will always be at same row simultaneously irrespective of the column.
Thus we can remove the row redundancy.
And update the parameters as [i][j1][j2]

$f(i, j_1, j_2)$ ⇒ Maximum chocolates/cherries collected by Alice from cell [i][j1] and bob from cell [i][j2] till last row.

Base Case.

① When i == N-1, we would have reached last row and we will check if both the users have reached on the same cell or not. If we yes, we will return cell[i][j1]. Else return cell[i][j1]+ cell[i][j2]

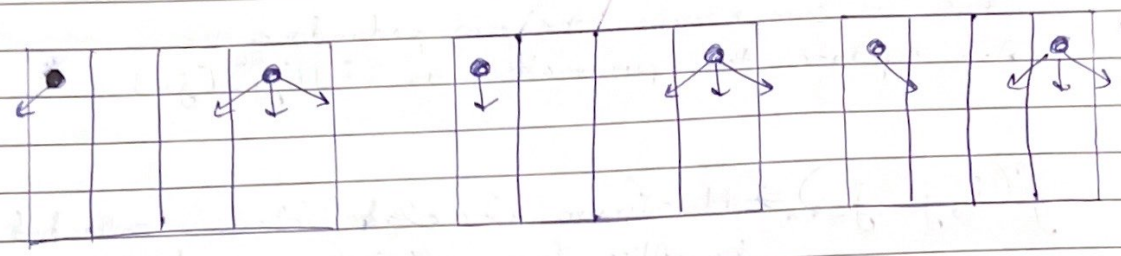
② Checks for Out of Bound Cases. We should go $j_1, j_2 < 0$ || $j_1, j_2 >= m$.

Note :- for Base Case there are 2 things to keep in mind.
① Destination
② Out of Bounds
Always write ② if first.

Try Out All Possible Choices

At every cell we have 3 options to traverse :-
(↙ ↓ ↘)

Try to Understand if we want to move both Alice And Bob together. And both of them can move in 3 Direction.



for Every Single Move of Alice, Bob has three options

Therefore, we have a total of 9 options.

Note if $(j_1 == j_2)$ as discussed in base case, we will consider chocolates collected by one of them.

Take Maximum Of All

As per question, Keep a track of Maximum path possible. Keep it stored it in 3D dp[][][].

Memoization

As Always The Same :- Create a 3D Array for dp[i][j1][j2] Initialize it with -1. Store the maxi variable in it. and return the ans.

Code

```
int f( int i, int j1, int j2, int r, int c, vector<vector<int>>& arr,
                    vector<vector<vector<int>>>& dp )
{
    if (j1<0 || j2<0 || j1>=c || j2>=c) return -1e8
    if ( i== r-1){
    {   if (j1==j2){
            return arr[i][j];
        Else
            return arr[i][j1]+ arr[i][j2];
    }
    if (dp[i][j1][j2] != -1) return dp[i][j1][j2];

    int maxi = INT_MIN;
    for(int di = -1 ; di<=1 ; di++) {
        for (int dj = -1 ; dj <= 1 ; dj++) {
            int ans;
            if (j1==j2)
                ans = arr[i][j] + f(i+1, j1+di, j2+dj, r, c, arr, dp);
            Else
                ans = arr[i][j2] + arr[i][j1] + f(i+1, j1+di, j2+dj,
                                                  r, c, arr, dp);
            maxi = max(ans, maxi);
        }
    }
    return dp[i][j1][j2] = maxi ;
}
```

Time Comp — $O(N*M*M) \times 9$

At max, there will be (N*M*M) recursive calls to solve a new problem and in every call, two nested loops together run for 9 times

Space Complexity – $O(N*M*M) + O(N)$

$O(N*M*M)$ → dp[]

$O(N)$ → recursion Stack

Tabulation

- For tabulation, we need to understand what a cell in 3D DP mean. So ~~we~~ when we say dp[2][0][3]. It means that we are getting the value of maximum chocolates collected by Alice And Bob, which Alice [2][0] and Bob [2][3].

In the recursive code, our base condition is when we reach the last row. Thus initialize. dp[n-1][][], as the base condition, dp[n-1][j1][j2]. if (j1 == j2) arr[i][j1]
Else arr[i][j1] + arr[i][j2].

- Now we will move from dp[N-2][][] to dp[0][][].
- We need 3 nested loop for the traversal.
- The outer three loops for traversal & the inner two loops are for those 9 options at every cell.
- We will then set dp[i][j1][j2] as the max of all 9 options.
- At last we will return dp[0][0][m-1] as our answer.

Time Comp → $O(N*M*M) \times 9$.
Space Comp → $O(N*M*M)$.

Code

```
int f(int n, int m, vector<vector<int>>& grid) {

    vector<vector<vector<int>>> dp(n, vector<vector<int>>
                (m, vector<int>(m, 0)));

    for(int j1=0; j1<m; j1++) {
        for(int j2=0; j2<m; j2++) {
            if(j1==j2)
                dp[n-1][j1][j2] = grid[n-1][j1];
            else
                dp[n-1][j1][j2] = grid[n-1][j1] + grid[n-1][j2];
        }
    }
    for(int i=n-2; i>=0; i--) {
        for(int j1=0; j1<m; j1++) {
            for(int j2=0; j2<m; j2++) {
                int maxi = INT_MIN;
                for(int di=-1; di<=1; di++) {
                    for(int dj=-1; dj<=1; dj++) {
                        int ans;
                        if(j1==j2)
                            ans = grid[i][j1];
                        else
                            ans = grid[i][j1] + grid[i][j2];
                        if((j1+di<0 || j1+di>=m) ||
                           (j2+dj<0 || j2+dj>=m))
                            ans += -1e8;
                        else
                            ans += dp[i+1][j1+di][j2+dj];
                        maxi = max(ans, maxi);
                    }
                }
                dp[i][j1][j2] = maxi; } } }
```

```
    return dp[0][0][m-1];
}.
```