3-d DP : Ninja and his friends

Problem Link: [Ninja and his friends]

Problem Description:

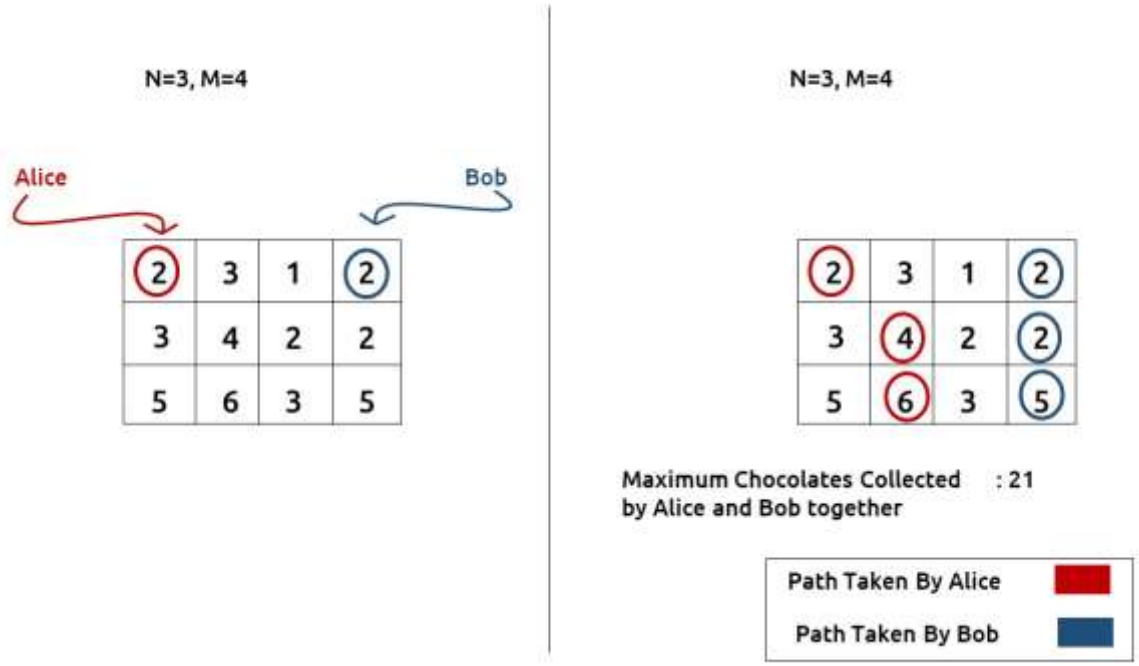
We are given an ‘N*M’ matrix. Every cell of the matrix has some chocolates on it, mat[i][j] gives us the number of chocolates. We have two friends ‘Alice’ and ‘Bob’. initially, Alice is standing on the cell(0,0) and Bob is standing on the cell(0, M-1). Both of them can move only to the cells below them in these three directions: to the bottom cell (↓), to the bottom-right

cell( ), or to the bottom-left cell( ).

When Alica and Bob visit a cell, they take all the chocolates from that cell with them. It can happen that they visit the same cell, in that case, the chocolates need to be considered only once.

They cannot go out of the boundary of the given matrix, we need to return the maximum number of chocolates that Bob and Alice can **together** collect.

Example:



Disclaimer: *Don’t jump directly to the solution, try it out yourself first.*

Pre-req: 2D DP

Solution :

In this question, there are two fixed starting and variable ending points, but as per the movement of Alice and Bob, we know that they will end in the last row. They have to move together at a time to the next row.

Why a Greedy Solution doesn't work?

As we have to return the **maximum** chocolates collected, the first approach that comes to our mind is to take a greedy approach and always form a path by locally choosing the option that gives us more chocolates. But there is no **'uniformity'** in the values of the matrix, therefore it can happen that whenever we are making a local choice that gives us a better path, we actually take a path that in the later stages is giving us fewer chocolates.

As a greedy solution doesn't work, our next choice will be to try out all the possible paths. To generate all possible paths we will use **recursion**.

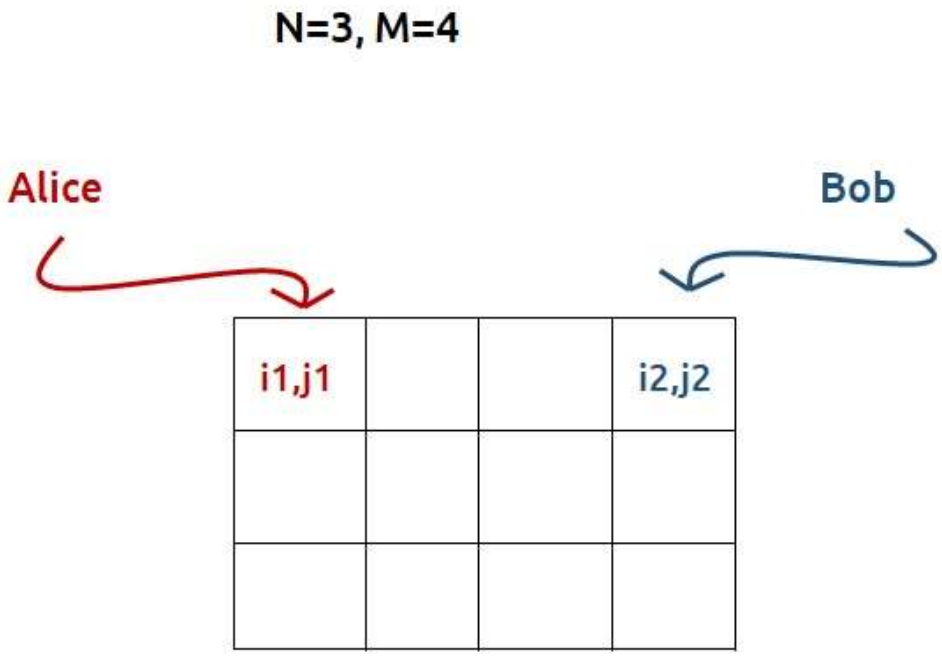
Steps to form the recursive solution:

We will first form the recursive solution by the three points mentioned in Dynamic Programming Introduction.

Step 1: Express the problem in terms of indexes.

This question is slightly different from all the previous questions, here we are given two starting points from where Alice and Bob can move.

We are given an 'N*M' matrix. We need to define the function with four parameters i1,j1,i2, and j2 to describe the positions of Alice and Bob at a time.



If we observe, initially Alice and Bob are at the first row, and they always move to the row below them every time, so they will always be in the same row. Therefore two different variables i1 and i2, to describe their positions are redundant. We can just use single parameter i, which tells us in which row of the grid both of them are.

Therefore, we can modify the function. It now takes three parameters: i,j1, and j2. f(i,j1,j2) will give us the maximum number of chocolates collected by Alice and Bob from their current positions to the last position.

f(i,j1,j2) -> Maximum chocolates collected by Alice from cell[i][j1] and Bob from cell[i][j2] till the last row.

Base Case:

There will be the following base cases:

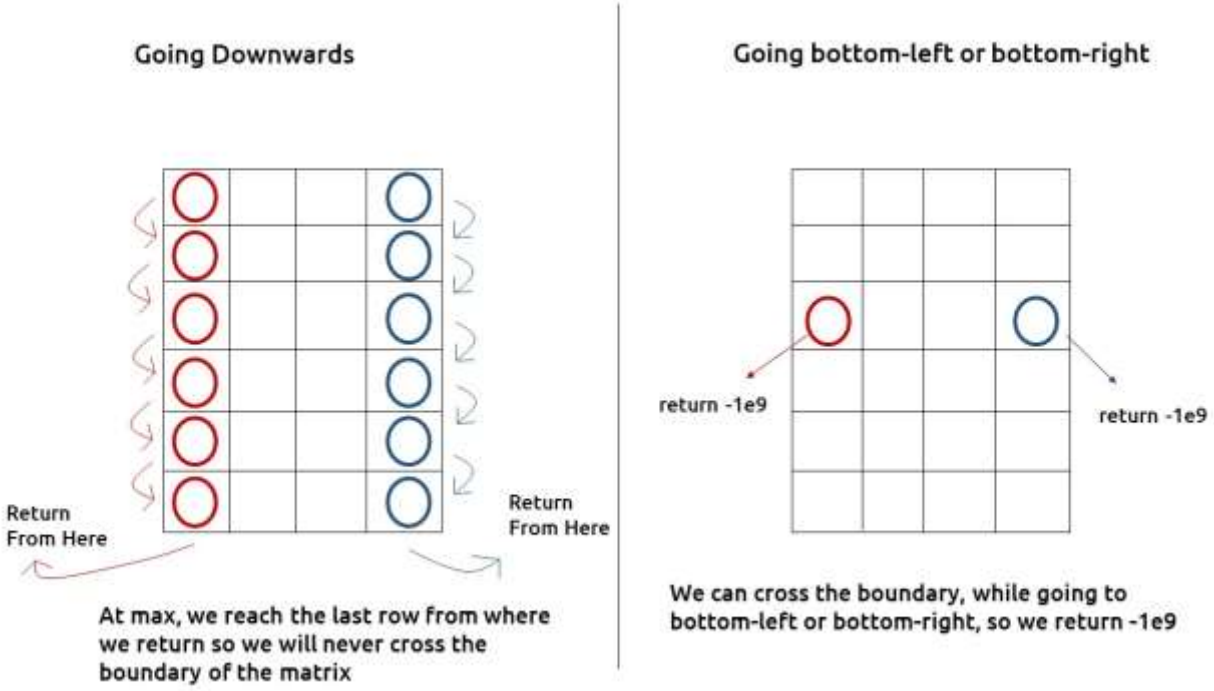
- When i == N-1, it means we are at the last row, so we need to return from here. Now it can happen that at the last row, both Alice and Bob are at the same cell, in this condition we will return only chocolates collected by Alice, mat[i][j1](as question states that the chocolates cannot be doubly calculated), otherwise we return sum of chocolates collected by both, mat[i][j1] + mat[i][j1][j2].

At every cell, we have three options to go: to the bottom cell (↓), to the

bottom-right cell( ) or to the bottom-left cell( )

As we are moving to the bottom cell (↓), at max we will reach the last row, from where we return, so we will never go out of the bounding index.

To move to the bottom-right cell( ) or to the bottom-left cell( ), it can happen that we may go out of bound as shown in the figure(below). So we need to handle it, we can return -1e9, whenever we go out of bound, in this way this path will not be selected by the calling function as we have to return the maximum chocolates.



- If j1<0 or j1>=M or j2<0 or j2>=M , then we return -1e9

  The pseudocode till this step will be:

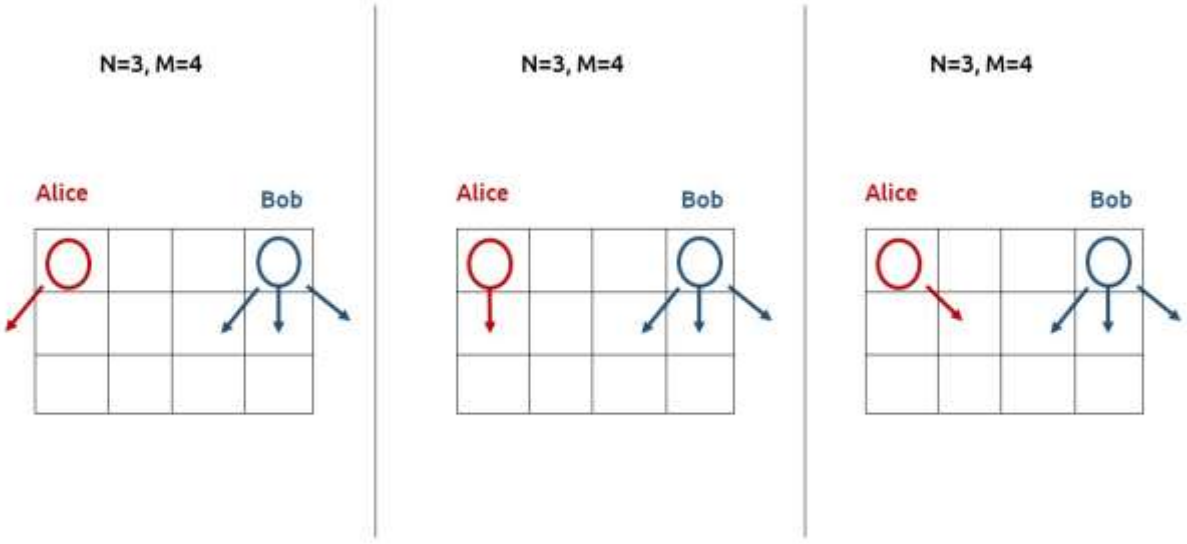
```
f(i,j1,j2) {
    if( j1<0 || j1>=m||j2<0 || j2>=m)
       return -1e9
    if( i==N-1){
          if( j1==j2)
             return mat[i][j1]
          else
             return mat[i][j1] + mat[i][j2]
    }
}
```

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

At every cell, we have three options to go: to the bottom cell (↓), to the

bottom-right cell( ) or to the bottom-left cell( )

Now, we need to understand that we want to move Alice and Bob together. Both of them can individually move three moves but say Alice moves to bottom-left, then Bob can have three different moves for Alice's move, and so on. The following figures will help to understand this:



For every single move of Alice, Bob have three different moves

Hence we have a total of 9 different options at every f(i,j1,j2) to move Alice and Bob. Now we can manually write these 9 options or we can observe a pattern in them, first Alice moves to one side and Bob tries all three choices, then again Alice moves, then Bob, and so on. This pattern can be easily captured by using two nested loops that change the column numbers(j1 and j2).

Note: if (j1===j2), as discussed in the base case, we will only consider chocolates collected by one of them otherwise we will consider chocolates collected by both of them.

```
f(i,j1,j2) {
    if( j1<0 || j1>=m||j2<0 || j2>=m)
       return -1e9
    if( i==N-1){
          if( j1==j2)
            return mat[i][j1]
          else
            return mat[i][j1] + mat[i][j2]
    }
     for(int di= -1; di<=1; di++){
         for(int dj= -1; dj<=1; dj++){
        if( j1==j2)
          return   mat[i][j1] + f(i,j1+di,j2+dj)
        else
          return   mat[i][j1] + mat[i][j2] + f(i,j1+di,j2+dj)
        }
      }
  }
```

Step 3: Take the maximum of all choices

As we have to find the **maximum chocolates collected** of all the possible paths, we will return the **maximum** of all the choices(the 9 choices of step 2). We will take a maxi variable(initialized to INT_MIN). We will update maxi to the maximum of the previous maxi and the answer of the current choice. At last, we will return maxi from our function as the answer.

The final pseudocode after steps 1, 2, and 3:

```
f(i,j1,j2) {
    if( j1<0 || j1>=m||j2<0 || j2>=m)
       return -1e9
    if( i==N-1){
          if( j1==j2)
             return mat[i][j1]
          else
             return mat[i][j1] + mat[i][j2]
    }
     maxi = INT_MIN
     for(int di= -1; di<=1; di++){
          for(int dj= -1; dj<=1; dj++){
        if( j1==j2)
          ans =  mat[i][j1] + f(i,j1+di,j2+dj)
        else
          ans =    mat[i][j1] + mat[i][j2] + f(i,j1+di,j2+dj)
          maxi = max(maxi,ans)
        }
      }
    return maxi
  }
```

Steps to memoize a recursive solution:

Before moving to the memoization steps, we need to understand the dp array we are taking. The recursive function has three parameters: i,j1, and j2. Therefore, we will also need to take a 3D DP Array. Its dimensions will be [N][M][M] because when we are moving, i can go from 0 to N-1, and j1 and j2 can go from 0 to M-1.

If we draw the recursion tree, we will see that there are overlapping subproblems. In order to convert a recursive solution the following steps will be taken:

1. Create a dp array of size [N][M][M], initialized to -1.
2. Whenever we want to find the answer of a particular row and column (say f(i,j1,j2)), we first check whether the answer is already calculated using the dp array(i.e dp[i][j1][j2]!= -1). If yes, simply return the value from the dp array.
3. If not, then we are finding the answer for the given values for the first time, we will use the recursive relation as usual but before returning from the function, we will set dp[i][j1][j2] to the solution we get.

Code:

- C++ Code
- Java Code

```cpp
#include<bits/stdc++.h>

using namespace std;

int maxChocoUtil(int i, int j1, int j2, int n, int m, vector < vector < int >>
& grid, vector < vector < vector < int >>> & dp) {
  if (j1 < 0 || j1 >= m || j2 < 0 || j2 >= m)
    return -1e9;

  if (i == n - 1) {
    if (j1 == j2)
      return grid[i][j1];
    else
      return grid[i][j1] + grid[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 = grid[i][j1] + maxChocoUtil(i + 1, j1 + di, j2 + dj, n, m, grid, dp);
```

```
      else
        ans = grid[i][j1] + grid[i][j2] + maxChocoUtil(i + 1, j1 + di, j2 + dj, n,
        m, grid, dp);
      maxi = max(maxi, ans);
    }
  }
  return dp[i][j1][j2] = maxi;
}

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

  vector < vector < vector < int >>> dp(n, vector < vector < int >> (m, vector <
int
  > (m, -1)));

  return maxChocoUtil(0, 0, m - 1, n, m, grid, dp);
}

int main() {

   vector<vector<int> > matrix{
      {2,3,1,2},
      {3,4,2,2},
      {5,6,3,5},
  };

  int n = matrix.size();
  int m = matrix[0].size();

  cout << maximumChocolates(n, m, matrix);
}
```

Output:

21

Time Complexity: O(N*M*M) * 9

Reason: At max, there will be N*M*M calls of recursion to solve a new problem and in every call, two nested loops together run for 9 times.

Space Complexity: O(N) + O(N*M*M)

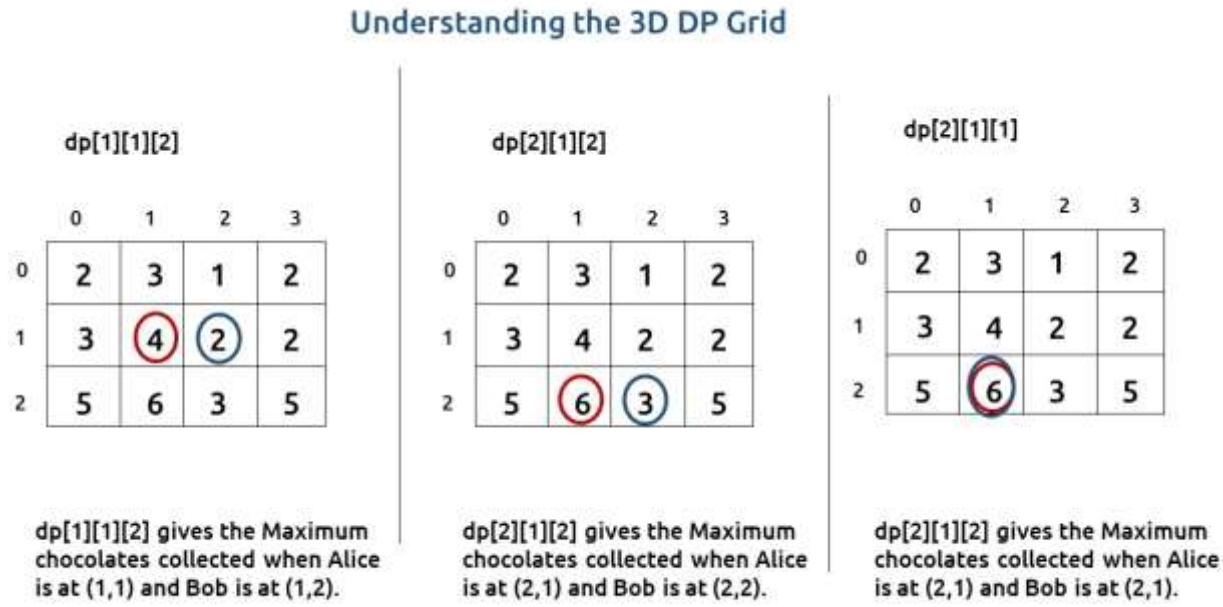
Reason: We are using a recursion stack space: O(N), where N is the path length and an external DP Array of size ‘N*M*M’.

Steps to convert Recursive Solution to Tabulation one.

For the tabulation approach, it is better to understand what a cell in the 3D DP array means. As we had done in memoization, we will initialize a dp[] array of size [N][M][M].

So now, when we say dp[2][0][3], what does it mean? It means that we are getting the value of maximum chocolates collected by Alice and Bob, when Alice is at (2,0) and Bob is at (2,3).

The below figure gives us a bit more clarity.



Next we need to initialize for the base value conditions. In the recursive code, our base condition is when we reach the last row, therefore in our dp array we will also initialize dp[n-1][][], i.e (the last plane of 3D Array) as the base condition. Dp[n-1][j1][j2] means Alice is at (n-1,j1) and Bob is at (n-1,j2). As this is the last row, its value will be equal to mat[i][j1], if (j1==j2) and mat[i][j1] + mat[i][j2] otherwise.

Once we have filled the last plane, we can move to the second-last plane and so on, we will need three nested loops to do this traversal.

The steps to convert to the tabular solution are given below:

- Declare a dp[] array of size [N][M][M]
- First initialize the base condition values as explained above.
- We will then move from dp[n-2][][] to dp[0][][]. We will set three nested loops to do this traversal.
- Inside the three nested loops(say i,j1,j2 as loop variables), we will use the recursive relations, i.e we will again set two nested loops to try all the nine options.
- The outer three loops are just for traversal, the inner two loops that run for 9 times mainly decide, what should be the value of the cell. If you are getting confused, please see the code.
- Inside the inner two nested loops, we will calculate an answer as we had done in the recursive relation, but this time using values from the next plane of the 3D DP Array(dp[i+1][x][y] instead of recursive calls, where x and y will vary according to inner 2 nested loops).
- At last we will set dp[i][j1][j2] as the maximum of all the 9 options.
- After the outer three nested loops iteration has ended, we will return dp[0][0][m-1] as our answer.

Code:

- C++ Code
- Java Code

```cpp
#include<bits/stdc++.h>


using namespace std;


int maximumChocolates(int n, int m, vector < vector < int >> & grid) {
  // Write your code here.
  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];
    }
  }

  //Outer Nested Loops for travering DP Array
  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;
```

```
        //Inner nested loops to try out 9 options
        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 += -1e9;
            else
              ans += dp[i + 1][j1 + di][j2 + dj];

            maxi = max(ans, maxi);
          }
        }
        dp[i][j1][j2] = maxi;
      }
    }
  }

  return dp[0][0][m - 1];

}

int main() {

  vector<vector<int> > matrix{
      {2,3,1,2},
      {3,4,2,2},
      {5,6,3,5},
  };

  int n = matrix.size();
  int m = matrix[0].size();

  cout << maximumChocolates(n, m, matrix);
}
```

Output:

21

Time Complexity: O(N*M*M)*9

Reason: The outer nested loops run for (N*M*M) times and the inner two nested loops run for 9 times.

Space Complexity: O(N*M*M)

Reason: We are using an external array of size ‘N*M*M’. The stack space will be eliminated.

Part 3: Space Optimization

If we look closely, to compute dp[i][j1][j2], we need values only from dp[i+1][][]. Therefore it is not necessary to store a three-dimensional array. Instead, we can store a two-dimensional array and update it as we move from one plane to the other in the 3D Array.

The Steps to space optimize the tabulation approach are as follows:

- Initially we can take a dummy 2D Array (say front).We initialize this 2D Array as we had done in the Tabulation Approach.
- Next we also initialize a 2D Array(say cur), which we will need in the traversal.
- Now we set our three nested loops to traverse the 3D Array, from the second last plane.
- Following the same approach as we did in the tabulation approach, we find the maximum number of chocolates collected at each cell. To calculate it we have all the values in our ‘front’ 2D Array.
- Previously, we assigned dp[i][j1][j2] to maxi, now we will simply assign cur[j1][j2] to maxi.
- Then whenever the plane of the 3D DP(the first parameter) is going to change, we assign front to cur.

At last, we will return front[0][m-1] as our answer.

Code:

- C++ Code
- Java Code

```
#include<bits/stdc++.h>


using namespace std;


int maximumChocolates(int n, int m, vector < vector < int >> & grid) {
  // Write your code here.
  vector < vector < int >> front(m, vector < int > (m, 0)), cur(m, vector < int >
  (m, 0));

  for (int j1 = 0; j1 < m; j1++) {
    for (int j2 = 0; j2 < m; j2++) {
      if (j1 == j2)
        front[j1][j2] = grid[n - 1][j1];
      else
        front[j1][j2] = grid[n - 1][j1] + grid[n - 1][j2];
```

```
    }
  }

  //Outer Nested Loops for travering DP Array
  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;

        //Inner nested loops to try out 9 options
        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 += -1e9;
            else
              ans += front[j1 + di][j2 + dj];

            maxi = max(ans, maxi);

          }
        }
        cur[j1][j2] = maxi;
      }
    }
    front = cur;
  }

  return front[0][m - 1];

}

int main() {

  vector<vector<int> > matrix{
      {2,3,1,2},
      {3,4,2,2},
      {5,6,3,5},
  };

  int n = matrix.size();
  int m = matrix[0].size();

  cout << maximumChocolates(n, m, matrix);
}
```

Output:

Time Complexity: O(N*M*M)*9

Reason: The outer nested loops run for (N*M*M) times and the inner two nested loops run for 9 times.

Space Complexity: O(M*M)

Reason: We are using an external array of size ‘M*M’.