Two-dimensional Dynamic programming

When there are two parmeters which define the state, then we call those problems 2D Dynamic Programming problems.

We use a 2D array to cache the results of the subproblems.

- There are two input strings or arrays
- Involves processing input string or array from both the sides.
- Input is a 2D array or list of strings

1. Minimum cost path

You are given a grid. You start at (0,0), top-left corner and you have to reach bottom left corner. You can only move down or right. Find out the minimum cost to reach the destination.

Hint: start from the last

0	47	8	18	1
43	25	39	36	13
22	8	13	38	46
41	41	40	25	44
29	43	22	50	10

Minimum cost path

0	47	8	18	1
43	25	39	36	13
22	8	13	38	46
41	41	40	25	44
29	43	22	50	10

Minimum cost path

0	47	8	18	1
43	25	39	36	13
22	8	13	38	46
41	41	40	25	44
29	43	22	50	10

Minimum cost path

0	47	8	18	1
43	25	39	36	13
22	8	13	38	46
41	41	40	25	44
29	43	22	50	10

Minimum cost path

0	47	8	18	1
43	25	39	36	13
22	8	13	38	46
41	41	40	25	44
29	43	22	50	10

Minimum cost path

0	47	8	18	1
43	25	39	36	13
22	8	13	38	46
41	41	40	25	44
29	43	22	50	10

1. State

1. State

Parameters

i - row of the current cell.

j - column of the current cell.

Cost function

minPath(i,j,G) - minimum cost to reach the cell (i,j) starting from (0,0) by moving only right or down.

Minimum sum path

0	47	8	18	1
43	25	39	36	13
22	8	13	38	46
41	41	40	25	44
29	43	22	50	10

2. Transitions

2. Transitions

Base case

i = 0, j = 0, return G[0][0]

Choices

minPath(i,j,G)

minPath(i-1,j,G)+G[i][j] , if i>0

minPath(i,j-1,G)+G[i][j] , if j >0

Optimal choice

minPath(i,j,G) = MIN(minPath(i-1,j,G),minPath(i,j-1,G))+G[i][j]

3. Recursive solution

3. Recursive solution

Pseudo code

```
minPath(i,j,G)
    if i==0 and j==0:
        return G[0][0]
    min = INFINITE
    if i>0
        min = minPath(i-1,j,G)+G[i][j]
    if j>0
        min = MIN(min,minPath(i,j-1,G)+G[i][j])
    return min
```

Java

```java
public static int minPath(int i, int j, int[][] grid) {
    if (i == 0 && j == 0) {
        return grid[0][0];
    }
    int min = Integer.MAX_VALUE;
    if (i > 0) {
        min = minPath(i - 1, j, grid) + grid[i][j];
    }
    if (j > 0) {
        min = Math.min(min, minPath(i, j - 1, grid) + grid[i][j]);
    }
    return min;
}
```

Python

```python
def min_path(i, j, G):
    if i == 0 and j == 0:
        return G[0][0]
    min_cost = sys.maxsize
    if i > 0:
        min_cost = min_path(i - 1, j, G) + G[i][j]
    if j > 0:
        min_cost = min(min_cost, min_path(i, j - 1, G) + G[i][j])

    return min_cost
```

Minimum cost path

0	47	8	18	1
43	25	39	36	13
22	8	13	38	46
41	41	40	25	44
29	43	22	50	10

Minimum cost path

0	47	8	18	1
43	25	39	36	13
22	8	13	38	46
41	41	40	25	44
29	43	22	50	10

Minimum cost path

0	47	8	18	1
43	25	39	36	13
22	8	13	38	46
41	41	40	25	44
29	43	22	50	10

Minimum cost path

0	47	8	18	1
43	25	39	36	13
22	8	13	38	46
41	41	40	25	44
29	43	22	50	10

4. Memoize

4. Memoization

We can cache the results in a 2D array.

Key -> (i,j)

Value -> Minimum cost to reach the cell, (i,j) from (0,0)

Default value -> -1

Java

```java
public static int minPathMemo(int i, int j, int[][] grid,int[][] cache) {
    if (i == 0 && j == 0) {
        return 0;
    }
    if(cache[i][j] != -1){
        return cache[i][j];
    }
    int min = Integer.MAX_VALUE;
    if (i > 0) {
        min = minPathMemo(i - 1, j, grid,cache) + grid[i][j];
    }
    if (j > 0) {
        min = Math.min(min, minPathMemo(i, j - 1, grid,cache) + grid[i][j]);
    }
    cache[i][j]=min;
    return min;
}
```

Python

```python
def min_path_memo(i, j, G, cache):
    if i == 0 and j == 0:
        return 0
    if cache[i][j] != -1:
        return cache[i][j]

    min_cost = sys.maxsize
    if i > 0:
        min_cost = min_path(i - 1, j, G) + G[i][j]
    if j > 0:
        min_cost = min(min_cost, min_path(i, j - 1, G) + G[i][j])
    cache[i][j] = min_cost
    return min_cost
```

5. Bottom up approach

5. Bottom up approach

minPath(i,j,G) = MIN(minPath(i-1,j,G),minPath(i,j-1,G))+G[i][j]

dp[i][j] = MIN(dp[i-1][j],dp[i][j-1])+G[i][j]

dp[5][5] depends on dp[4][5] and dp[5][4],

so our for loops should go from

i=0,1…M

and

j=0,1,…M

Java

```java
public static int minPathDP(int[][] G){
    int M = G.length;
    int[][] dp = new int[M][M];
    dp[0][0] = G[0][0];
    for(int i=0;i<M;i++){
        for(int j=0;j<M;j++){
            if(i==0 && j == 0){
                continue;
            }
            dp[i][j] = Integer.MAX_VALUE;
            if(i>0){
                dp[i][j] = dp[i-1][j]+G[i][j];
            }
            if(j>0){
                dp[i][j] = Math.min(dp[i][j],dp[i][j-1]+G[i][j]);
            }
        }
    }
    return dp[M-1][M-1];
}
```

Python

```python
def min_path_dp(G):
    M = len(G)
    dp = [[0 for _ in range(0, M)] for _ in range(0, M)]

    dp[0][0] = G[0][0];
    for i in range(0, M):
        for j in range(0, M):
            if i == 0 and j == 0:
                continue
            dp[i][j] = sys.maxsize
            if i>0:
                dp[i][j] = dp[i-1][j]+G[i][j]
            if j>0:
                dp[i][j] = min(dp[i][j],dp[i][j-1]+G[i][j])
    return dp[M-1][M-1]
```

Reconstruct the path

6. Reconstruct the path

We need to print the path.

Along with the minimum cost, we will record the direction in a 2D array ‘dir’

0 - Down

1 - Right

We start from the last position (N-1,N-1), i=N-1,j=N-1

if its 0, then we assign i=i-1, we print (i,j)

if its 1, then we assign j=j-1, we print (i,j)

We repeat the process until i=0 and j=0

Minimum cost path

0	47	8	18	1
43	25	39	36	13
22	8	13	38	46
41	41	40	25	44
29	43	22	50	10

Time and space complexity

Recursive solution

We draw recursion tree.

Its a binary tree.

Height of the tree is 2N.

Number of nodes in the tree = $2^{2N} = 4^N$

Time complexity is , $O(4^N)$, Exponential

Space complexity O(1)

Dynamic programming solution

We have two forloops iterating through every cell in the grid of length N once.

There are NXN cells = N^2

Time complexity is $O(N^2)$

Space complexity is $O(N^2)$