Todays Content:

- 2D matrices questions
- Number of ways to go from (0,0) → BR cell
  - ↳ // Ways, with blocked cell
- Min Cost to reach BR → TODO
- Dungeons & Dragons

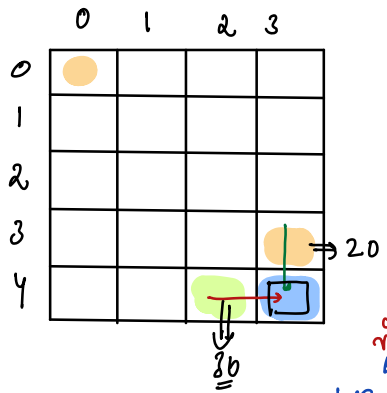
10) Number of ways to go from (0,0) → CBR cell
TL → BR

cell → right or bottom

	0	1	2
0	TL		
1			
2			BR

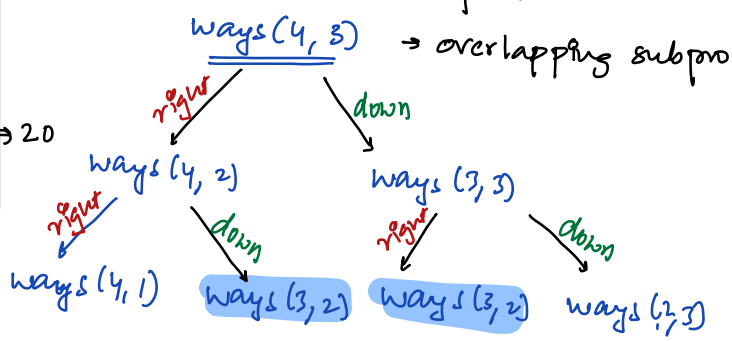
Paths:

(0,0) (0,1) (0,2) (1,2) (2,2)
(0,0) (0,1) (1,1) (1,2) (2,2)
(0,0) (0,1) (1,1) (2,1) (2,2)
(0,0) (1,0) (1,1) (1,2) (2,2)
(0,0) (1,0) (1,1) (2,1) (2,2)
(0,0) (1,0) (2,0) (2,1) (2,2)



// ways to reach from (0,0) → (4,3)

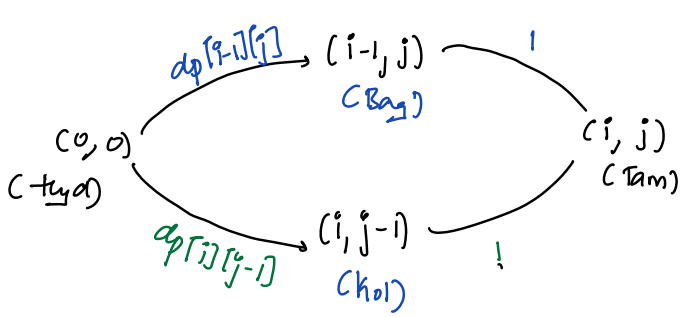
→ optimal substructure
→ overlapping subproblem



7 Steps:

dp[i][j] = # Number of ways to reach from (0,0) → (i,j)

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



Have doubt in dp expression always write the diagram

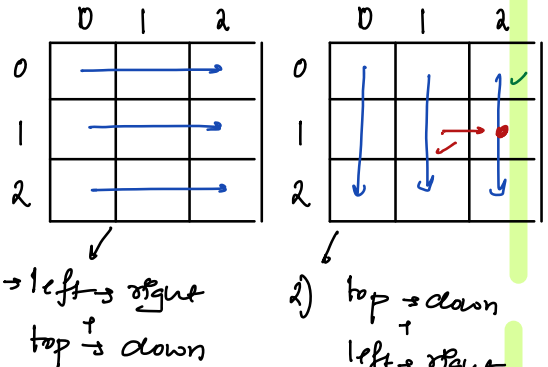
dp[i][j] = dp[i-1][j] + dp[i][j-1]

i > 0 , j > 0

Base Conditions: { i==0 || j==0 } code fails

↳ if (i==0 || j==0) dp[i][j] = 1

Table: dp[N][M]



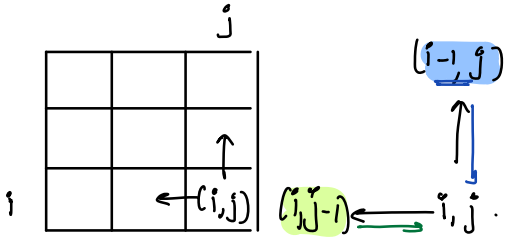
Code:

```
int ways(int N, int M) {
    int dp[N][M];
    i=0; i<N; i++) {
        j=0; j<M; j++) {
            if( i==0 || j==0) { // handle Base Conditions
                dp[i][j] = 1
            }
            else {
                dp[i][j] = dp[i-1][j] + dp[i][j-1]
            }
        }
    }
    return dp[N-1][M-1]
}
```

TC: # O(N*M) + 1

SC: # O(N*M)

depends



Note: In iterative dp while filling table it's also important on how we are iterating in table.

1) Resolve top → down

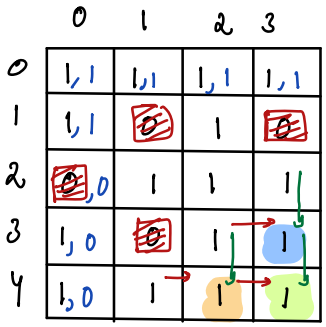
2) Resolve left → right

28) Number of ways to go from (0, 0) → (BR cells)

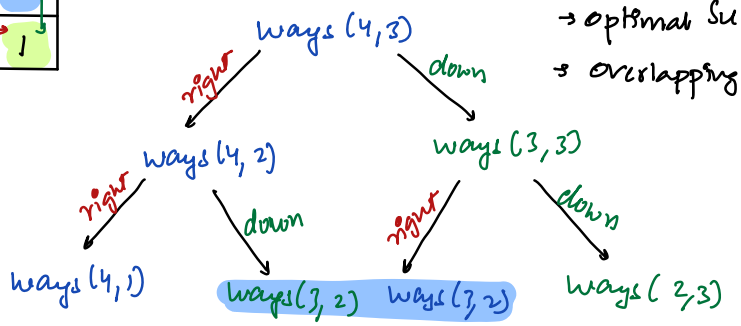
// cell → right / cell → bottom

// mat[i][j] = 0, it means it blocked

// We cannot traverse a blocked cell



// ways to reach from (0, 0) → (4, 3)



→ optimal Substructure

→ overlapping subproblem

#Steps :

dp[i][j] = # Number of ways to reach from (0, 0) → (i, j)

dp[i][j] = { if (mat[i][j] == 0) { dp[i][j] = 0 } (cell blocked)
else { dp[i-1][j] + dp[i][j-1] } }

Base Condition :

if (i == 0 || j == 0) { Code fails }

dp table : int dp[N][M] = {0} → default initialize to zero

// Case - I :

Iterate on 0th jth col

```
i = 0; i < n; i++) {
    if (mat[i][0] == 1) { dp[i][0] = 1 }
    else { break }
}
```

Case - II

Iterate on 0th row

```
j = 0; j < m; j++) {
    if (mat[0][j] == 1) { dp[0][j] = 1
    else { break }
}
```

Code:

```
int ways(int mat[][], int N, int M){
    int dp[N][M] = {0} // initialize to 0
    Iterate on 0th jth col
    for(i=0; i<n; i++){
        if(mat[i][0]==1) { dp[i][0] = 1}
        else {break}
    }
    Iterate on 0th row
    for(j=0; j<m; j++){
        if(mat[0][j]==1) { dp[0][j]=1}
        else {break}
    }
    for(i=1; i<N; i++){
        for(j=1; j<m; j++){
            if(mat[i][j]==0) { dp[i][j]=0}
            else{
                dp[i][j] = dp[i-1][j] + dp[i][j-1]
            }
        }
    }
    return dp[N-1][M-1]
}
```

```
// int ways(int mat[][], int N, int M){
    int dp[N][M] = {-1}   // ways (0,0) → (N-1, M-1)
    if(mat[0][0] == 0) {return 0}
    dp[0][0] = 1 // Base condition // Vimp
    func(mat, dp, N-1, M-1)
                   ↳ // It always reducing
                   ↳ // It can never touch N, M
}

int func(int mat[][], int dp[][], int i, int j) {
    if(i<0 || j<0) {return 0}
    if(mat[i][j] = 0) {return 0}
        } In recursive codes advantage, writing base conditions becomes easy

    if(dp[i][j] == -1) {
        ↳ // calculating dp[i][j] for first time

        dp[i][j] = { func(mat, dp, i-1, j)
                              +
                   { func(mat, dp, i, j-1) }

    }
    return dp[i][j]
}
```

11:00 pr → 11:05pm

TC: $O(N*M) * O(1)$ SC: $O(N*M)$

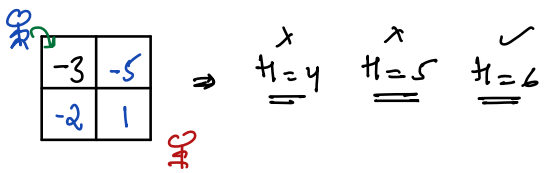
30) Min cost to reach from (0,0) → BR cell

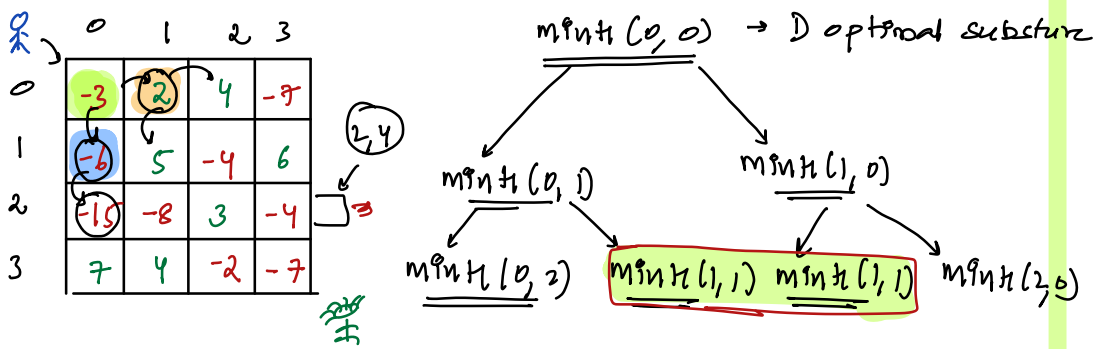
	0	1	2	3	4
0	2	1	3	7	2
1	4	2	1	3	1
2	3	2	3	1	7
3	6	1	3	2	4
4	1	5	2	7	3

→ TODO

// Given mat[N][M], where each cell mat[i][j] indicates health gained, minimum health required at 0,0 so that we can reach [N-1][M-1].

1) cell → right cell → bottom
2) If health reaches zero at any place you are dead
3) We are starting at (0, 0) → (N-1, M-1)

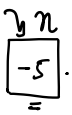




dp[i][j] = min Health required to start (i, j) to reach N-1, M-1

Expression :

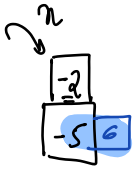
Case-I :



$n - 5 = 1$

$n = 6$

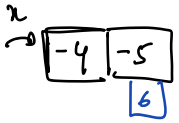
Case-II



$n + (-2) = 6$

$n = 8$

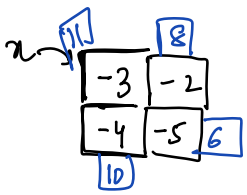
Case-III



$n + (-4) = 6$

$n = 10$

Case-IV

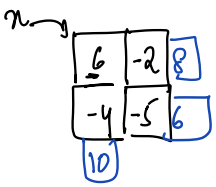


$n + -3 = \min(8, 10)$

$n = \min(8, 10) + 3$

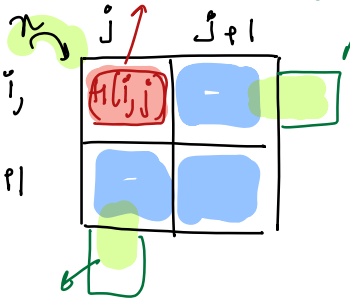
$n = 11$

Case-V



$n + 6 = \min(8, 10)$

$n + 6 = 8, \quad n = 2$



$n + H[i,j] = \min(dp[i,j+1], dp[i+1,j])$

$n = \min(dp[i,j+1], dp[i+1,j]) - H[i,j]$

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

↳ Edge Case, atleast $dp[i][j]$ should be 1

Case-VI

$n \rightarrow$ atleast $n = 1$

Case-7

$n + 9 = \min(8, 10)$

$n + 9 = 8$

$n = -1$

dpState: dp[i][j] = min health required [i,j] → [N-1, M-1]

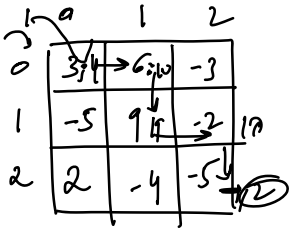
dpExpression: dp[i][j] = max(1, min(dp[i,j+1], dp[i+1,j]) - H[i,j])

BaseCaseCond?: { i == N-1 || j == M-1 } // won't work

dp table: dp[N][M]

```
minH(int H[][], int N M){
    int dp[N][M] = {-1}
    dp[N-1][M-1] = max(1, 1 - H[N-1][M-1])   // EdgeCase
    // mincost to start at (0,0)
    func(H, N, M, dp, 0, 0)
}

int func(int H[][], int N, int M, int dp[][], int i, int j){
    if(i == N || j == M) { return INT_MAX }   // EdgeCase
    if(dp[i][j] == -1){
        int a = func(H, N, M, dp, i, j+1)
        int b = func(H, N, M, dp, i+1, j)
        dp[i][j] = max(1, min(a, b) - H[i][j])
    }
    return dp[i][j]
}
```



	0	1	2
0	1	1	11
1	6	1	8
2	8	10	6