Todays Content:

1. Iterative Dp

   a) Subseq Sum without Adj : Dp:2
   b) 2D Matrix
      → # no: of paths in blocked : Dp:2
   c) KnapSack : Dp:3
   d) MinPath Sum: Dp:4

→ Recursive Dp / Iterative Dp:

a) Dp State
b) Dp expression
c) Final ans
d) Dp Table / TC/SC
e) Code : of Iterative

Step1: Edge Case : ?
: Input value for which dp expression fails / Invalid
Separately code Edge Cases

Step2: Fill remaining Table ✓

Step3: Space Optimization: ✓

IQ) Given ar[N] calculate max subseq sum

Note1: In a subseq 2 adj elements cannot be picked

Note2: Empty sequence is also valid

ar[3] = 9 14 3 : ans = 14

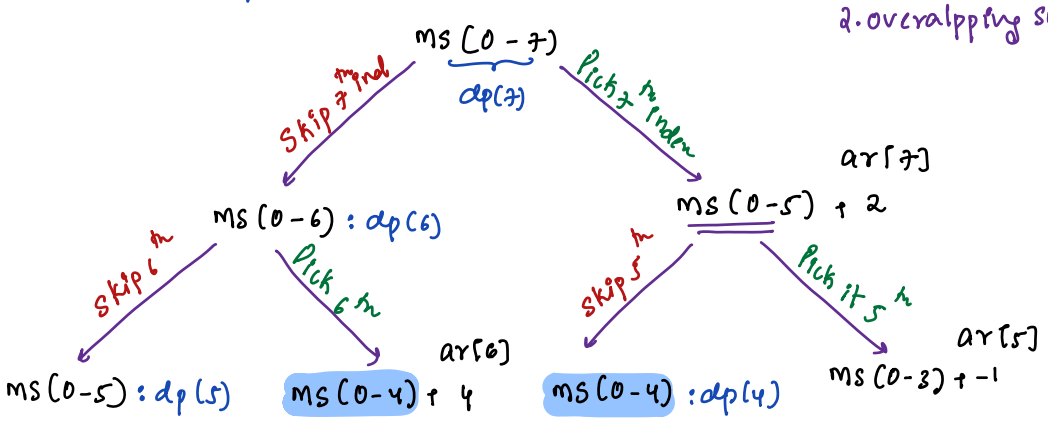
ar[4] = 9 4 13 24 : ans = 33

Idea:

	0	1	2	3	4	5	6	7
ar[8] =	2	-1	-4	5	3	-1	4	2

// max subseq sum from [0-7] without adj elements

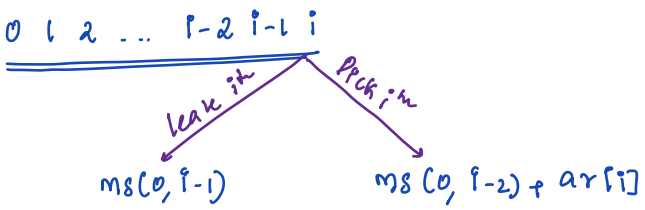
1. Subproblems
2. overalpping subpro



Dp Steps:

dp state : dp(i) = max subseq sum from [0, i] without adj elements

dp Exp : // Calculate state with subproblems



$dp(i) = \max(dp(i-1), dp(i-2) + ar[i])$

Final ans: We want max subsum from {0, n-1} : dp[n-1]

Table size: int dp[n] TC: #states * TC for each state SC: O(N+N)

↳ #n × 1 : O(N)

```
int maxSub ( int A[] ) {
    int n = A.size()
    int dp[n];

    Step1: Edge cases:
        dp[0] = max{ar[0], 0}
        dp[1] = max{ar[1], ar[0], 0}
              ↳ max{ar[1], dp[0]}

    Step2: Fill remaining Table:
        for(i = 2; i < n; i++){
            dp[i] = max( dp[i-1], dp[i-2] + ar[i])
        }

    Step3: Return final ans:
        return dp[n-1]
}
```

dp Expression: ?

$dp(i) = \max(dp[i-1], dp[i-2] + ar[i])$

for which all dp state above expression fails?

- i = 0: substitute = $dp(0) = \max(dp(-1), dp(-2) + ar[0])$
- i = 1: substitute = $dp[1] = \max(dp[0], dp[-1] + ar[1])$

i = 2: Substitute = $dp[2] = \max(dp[1], dp[0] + ar[2])$ ✓

	0	1	2	3	4
ar[5] =	-3	5	2	4	7

ans = 12

	0	1	2	3	4	
dp[5] =	0	5	5	9	12	

$dp[2] = \max(dp[1], dp[0] + ar[2]) = \max(5, 0+2) = 5$

$dp[3] = \max(dp[2], dp[1] + ar[3]) = \max(5, 5+4) = 9$

$dp[4] = \max(dp[3], dp[2] + ar[4]) = \max(9, 5+7) = 12$

Space Optimization:

In exp figure out at max how many dp states are needed.

$dp[i] = \max(dp[i-1], dp[i-2] + ar[i])$

We just 3 state value at any point hence 3 variable are enough.

// Eg: ar[6] = { -3 4 2 5 -4 5 }

dp[0] dp[1] dp[2] dp[3] dp[4] dp[5]

a b c

a b c

Dp Space Optimization.

```
a = max(0, ar[0])
b = max(ar[1], a)
for(i=2; i<n; i++){
    c = max(b, a+ar[i])
    a = b
    b = c
}
return c;
```

// Space Optimization 2

ar[6] = { -3 4 2 5 -4 5 }

0 1 2

int dp[3] = { ~~dp[0]~~ dp[1] dp[2] }

~~dp[3]~~ dp[4] dp[5]

dp[6]

index

dp[0] → 0

dp[1] → 1

dp[2] → 2

dp[3] → 0

dp[4] → 1

dp[5] → 2

dp[6] → 0

dp[i] actual → i%3

dp[i-1] actual → (i-1)%3

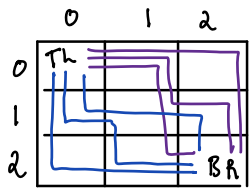
dp[i-2] actual → (i-2)%3

dp[n-1] actual → (n-1)%3

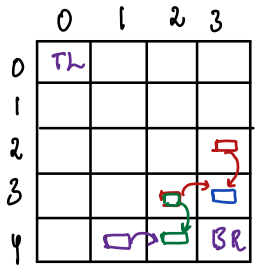
```
int maxSub(int ar[]){
    int dp[3]
    dp[0] = max(0, ar[0])
    dp[1] = max(ar[1], dp[0])
    for(i=2; i<n; i++){
        dp[i%3] = max{dp[(i-1)%3], dp[(i-2)%3]+ar[i]}
    }
    return dp[(n-1)%3]
}
```

28) Number of ways to go from (0,0) → BR cells, mat[N][M]

Note: Cell → right r, ↓ bottom

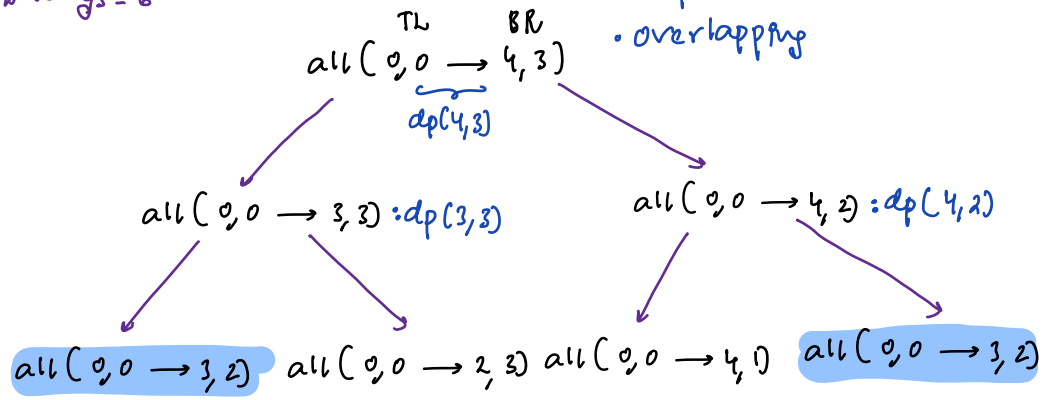


ways = 6



TODO: fix BR & Ty

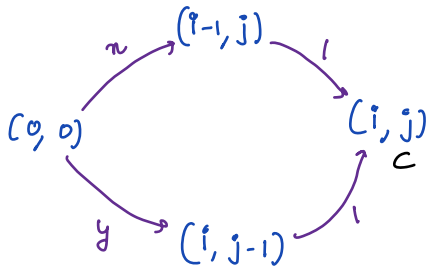
- subproblems
- overlapping



Dp Steps:

Dp state: dp(i,j) = # ways to go from (0,0) - (i,j)

Dp Expression: dp(i,j) = dp(i-1,j) + dp(i,j-1)



Total ways: (0,0) → (i,j) = x+y

x = ways (0,0) → (i-1,j) = dp(i-1,j)

y = ways (0,0) → (i,j-1) = dp(i,j-1)

Final ans = # ways from (0,0) → (n-1, m-1) : dp[n-1][m-1]

(TL = (0,0), BR = (n-1, m-1))

Dp Table = int dp[n][m] TC = #N*M * 1

int dp[N][M]

Step1: Edge Cases:

$dp(i,j) = dp(i-1,j) + dp(i,j-1)$

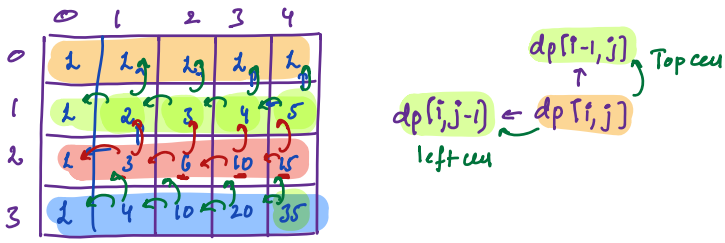
For which all valid dp state will expression fail?

$dp(0,0) = dp(-1,0) + dp(0,-1)$

$dp(0,j) = dp(-1,j) + dp(0,j-1)$ * i==0, it fails

$dp(i,0) = dp(i-1,0) + dp(i,-1)$ ∀ j==0, it fails.

Eg: To fill Edge Cases: dp[4][5]



Total ways to reach (0,0) → (3,4) = 35

(0,0) → (0,1)

```
int ways(int N, int M) // ways to reach (0,0) → (N-1, M-1)
{
    int dp[N][M];
    for(j=0; j<M; j++){
        dp[0][j] = 1
    }
    for(i=0; i<N; i++){
        dp[i][0] = 1
    }
    for(i=1; i<N; i++){
        for(j=1; j<M; j++){
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
        }
    }
    return dp[N-1][M-1]
}
```

Space Optimization:

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

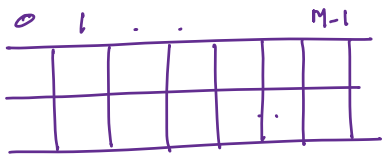
: At any given point we only need 2 rows :?

: i^{th} row ⟶ $(i-1)^{st}$ row

dp[2][M]:

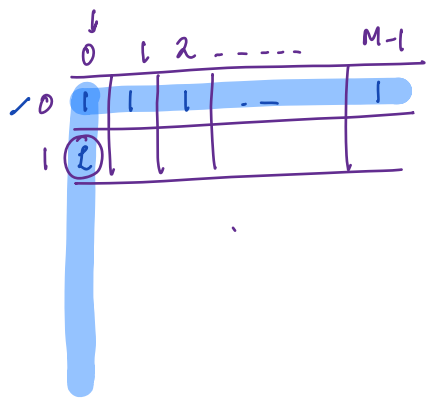
dp[4] dp[2] ~~dp[0]~~ → 0

dp[5] dp[3] ~~dp[1]~~ → 1



dp[i^{th} row] ⟶ Stored in $i\%2^{th}$ row

```
int ways(int N, int M)
{
    int dp[2][M];
    for(j = 0; j < M; j++){
        dp[0][j] = 1
    }
    for(i = 1; i < N; i++){
        // For every row dp[i][0] = 1
        dp[i%2][0] = 1
        for(j = 1; j < M; j++){
            dp[i%2][j] = dp[(i-1)%2][j] + dp[i%2][j-1]
        }
    }
    return dp[(N-1)%2][M-1]
}
```



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

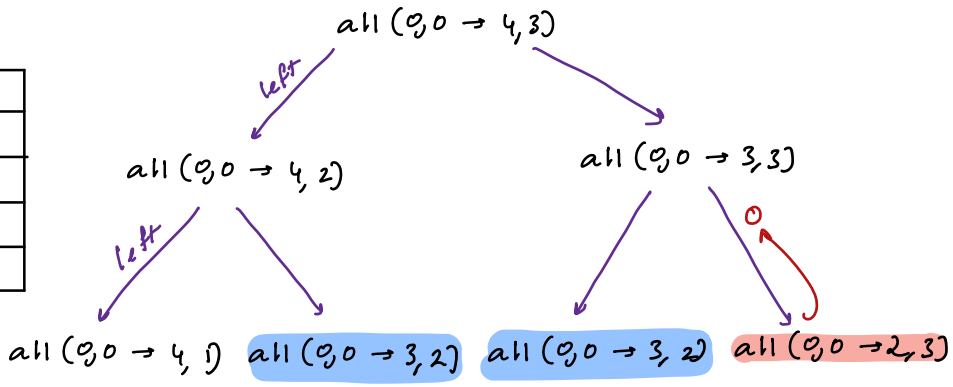
Note1: Cell → right, ↓ bottom

Note2: mat[i,j] = 0, blocked cell mat[i,j] = 1 unblocked cell

Note3: A path cannot go from blocked cell

mat[5][4]

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



Dp Expression:

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

i=0 || j=0 exp fails.

Edge Case:

mat[5][4]

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

dp[5][4]

	0	1	2	3
→0	1	0	0	0
1	1			
2	0			
3	0			
4	0			

cell [0][1] blocked all right = 0

0,0 → 0,1

0,0 → 0,1

cell [2][0] is blocked hence all below 0

```
int ways(int mat[N][M])
{
    int dp[N][M]
    Step1: Edge Case
    bool blo = false
    for(j=0; j<M; j++){
        // cell [0][j] blocked or not?
        if(mat[0][j]==0) { blo = true }
        if(blo==true) { dp[0][j] = 0 }
        else { dp[0][j] = 1 }
    }
    blo = false
    for(i=0; i<N; i++){
        // cell [i][0] blocked or not?
        if(mat[i][0]==0) { blo = true }
        if(blo==true) { dp[i][0] = 0 }
        else { dp[i][0] = 1 }
    }
    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]
}
```

SC: Try Space Optimization: 2 Rows

Given N items each with a weight & value, find max value which can be obtained by picking items such that total weight of all items $\leq k$ (k given)

Note1: Every item can be picked at max 1 time

Note2: We cannot take a part of item

Eg: N=4 items k=50

Items:	0	1	2	3
Weight[]:	20	10	30	40

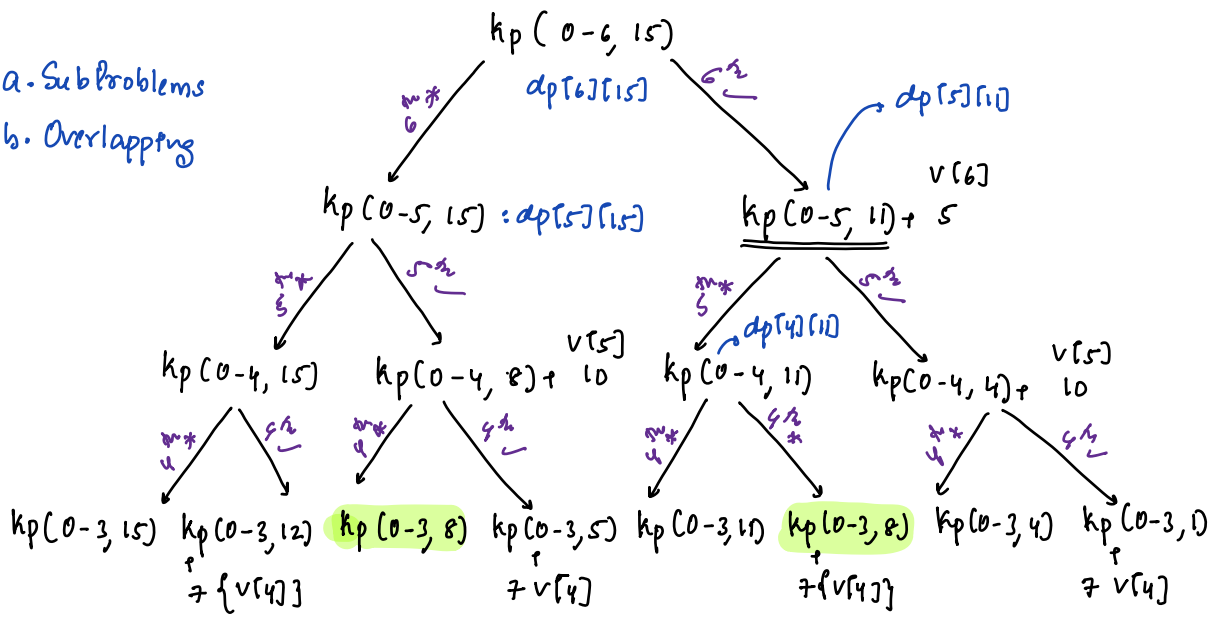
0/1 knapSack

k=15		0	1	2	3	4	5	6
N=7	w[]	4	1	5	4	3	7	4
.	v[]	3	2	8	3	7	10	5

// Using items (0-6) & max $w \leq 15$, get max value can be obtained.

a. Subproblems

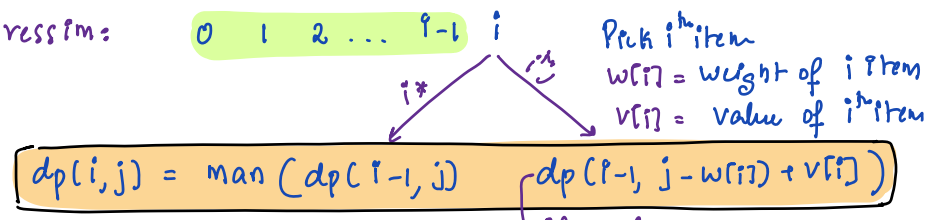
b. Overlapping



dp Steps:

→ dp State : dp(i,j) : From items [0, i] & Pick items such that
: total weight <= j {rem weight} & we get max value

→ dp Expression:



if: j {rem_weight} >= w[i].
: only than we can pick i^th item
↳ exists: if j >= w[i]

→ dp ans: From items [0, N-1]
Pick items such that
Total weight <= k & pick max val
Problem = dp[N-1][k]

State * TC for each

→ dp table: int dp[N][k+1] TC: O(N*k) * ! SC: O(N*k) + stacksize

Edge Cases:

$$dp(i,j) = \max\left(dp(i-1,j),\ \text{if } j>=w[i]\ dp(i-1,\ j-w[i]) + v[i]\right)$$

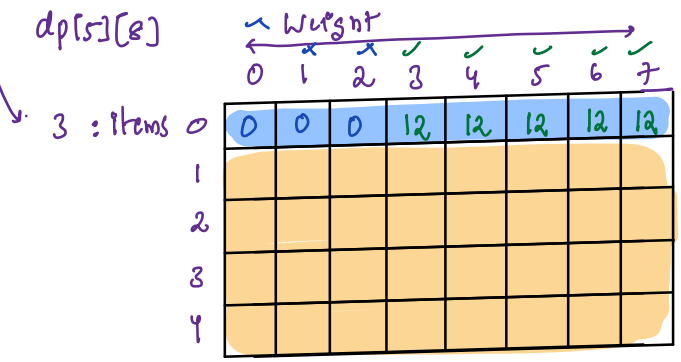
i=0, dp(0,j) = max (dp(-1, j), dp(-1, j - w[i]) + v[i])

↳ j=0, dp(i,0) = max (dp(i-1,0), if (0 >= w[i]) { dp(i-1, 0 - w[i]) }
↳ not Edge Case: ↳ It won't even occur

items :	0	1	2	3	4
w[] :	3	6	5	2	4
v[] :	12	20	15	6	10

items = 5 k = 7

dp[5][8]




```
int 0/1knapSack( int N, int k, int w[], int v[]){
    int dp[N][k+1]  ⟶  dp[2][k+1]

    Step1: Edge Case:
    j=0; j<=k; j++){ // i=0
        // With weight j, check if we can pick 0th item or not
        if( j>=w[0]){ dp[0][j]=v[0]}
        else{ dp[0][j]=0}
    }
    Step2: Fill all rem
    i=1; i<N; i++){
        j=0; j<=M; j++){
            dp[i][j] = dp[i-1][j]  ⟶  dp[i%2][j] = dp[(i-1)%2][j]
            if( j>=w[i]){
                dp[i][j] = max( dp[i][j], dp[i-1][j-w[i]]+v[i])
            }}  dp[i%2][j] = max( dp[i%2][j], dp[(i-1)%2][j-w[i]]+v[i])
    }
    return dp[N-1][k]  ⟶  dp[(N-1)%2][k]
}
```

Minimum Falling Path Sum II

Given N*N matrix find min sum we can get such that

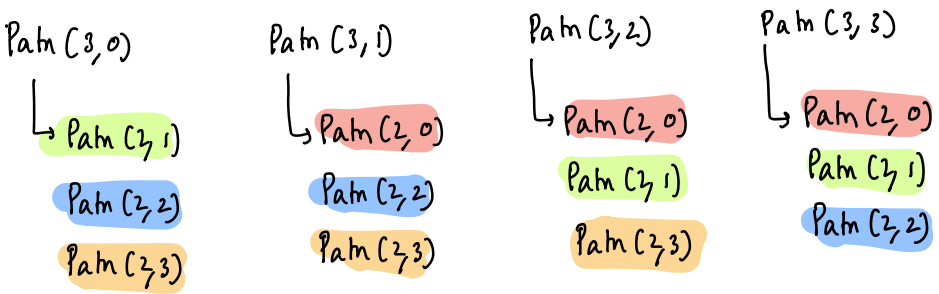
a) In Every row, we should pick 1 element

b) No 2 elements chosed in adjacent rows, should be in same column

Ex: Mat[4][4] : Pick 4 ele, 1 ele in each row.

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

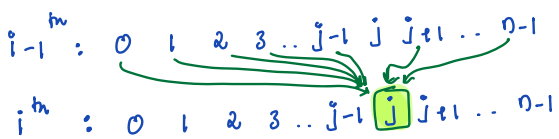
min Path Cost ending at (3,0) such that, no 2 adjacent rows same column Picked.



Dp Steps:

dpState: dp(i,j) = minPath Cost ending at (i,j) no 2 adjacent rows same column Picked.

dpExpression: dp(i,j) =



$$dp(i,j) = \min\left[\begin{array}{l} \text{if } k=0 \\ k \neq j \end{array}^{n-1} dp(i-1,k)\right] + mat[i][j]$$

Final ans: Min cost to reach last row

: min value of last row

: min{ dp[n-1][0] dp[n-1][1] dp[n-1][2] ... dp[n-1][3] }

states * TC for each state

Dp Table: dp[N][N]

TC: O(N²) * O(N)