Lecture : 69

Dynamic programming

→ Jo kaam Ek baar kar chuke hai usko yaad rakho.

→ Fibonacci series :

$f(n)$: 0, 1, 1, 2, 3, 5, 8, 13, 21, - - - -

→ Dynamic programming problem can solved in two ways :

① Top down Approach + Memoization (Recursion)

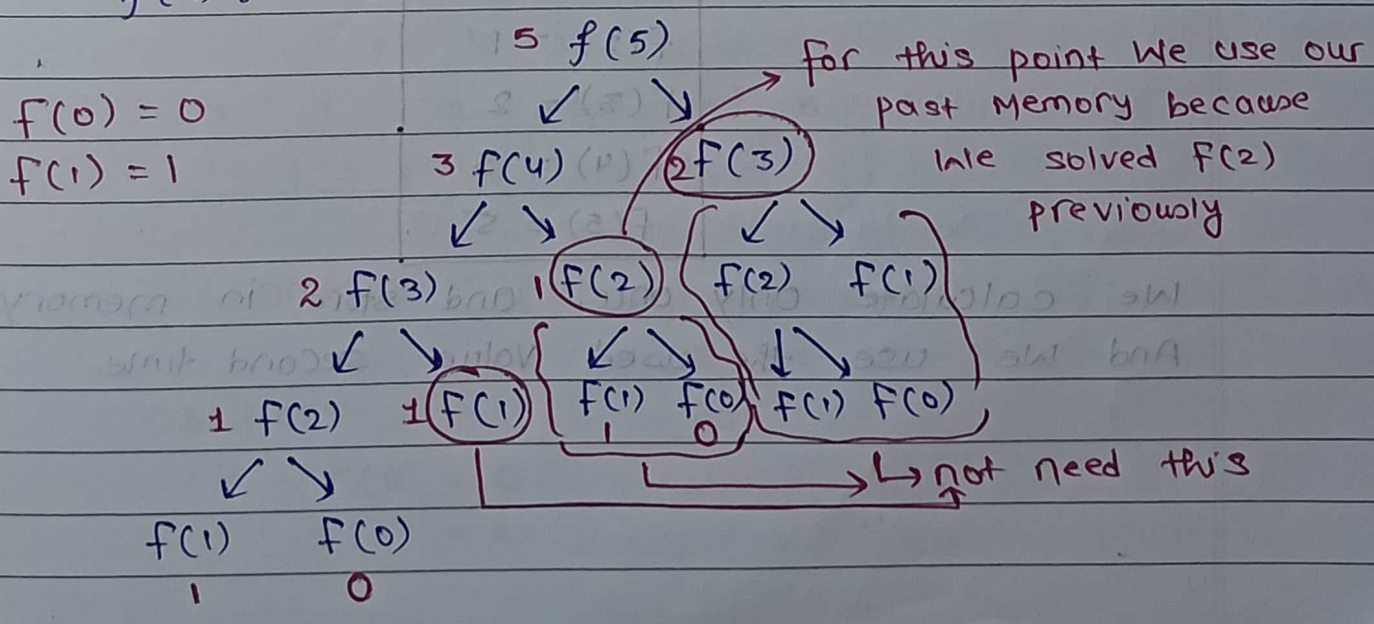
② Tabulation (Bottom up approach) (loop)

space optimization.

→ n^{th} fibanacci number :

$$f(n) = f(n-1) + f(n-2)$$

$f(5)$:



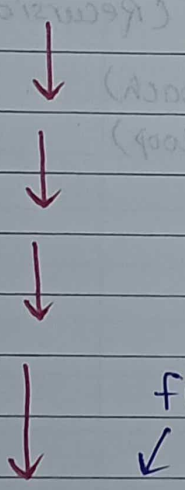
f(0) = 0

f(1) = 1

* previously, We solve all the value
So, Time Complexity
↳ $O(2^n)$

* Now, With the help of Dynamic progamming one value can solved only one time and store the value in memory and used again in $O(1)$ time.

So, Time Complexity :
↳ $O(n)$



We go from top to bottom.

So, this is top-down Approach.

Memory

f(1) = 1
f(0) = 0
f(2) = 1
f(3) = 2
f(4) = 3
f(5) = 5

We calculate only once, and store in memory. And We use the used value second time.

→ Divide and Conquer

↳ DP ke problem ko pahle divide karte jao or uske baad usko ek ek kar ke jodte jao.

→ Overlapping subproblem

↳ Jo problem ka part solve kar chuke hai usko dubara solve nahi karo, Memory me store karwa kar usko direct use kar lo.

Before DP

2^n

After DP

n

For $n = 10$

$2^{10} = 1024$

For $n = 10$

10

See the difference of both

How to store the value which we calculated previously?

Make an array of size $(n+1)$

F(5)

↳ arr[6]

0	1	2	3	4	5
-1	-1	-1	-1	-1	-1

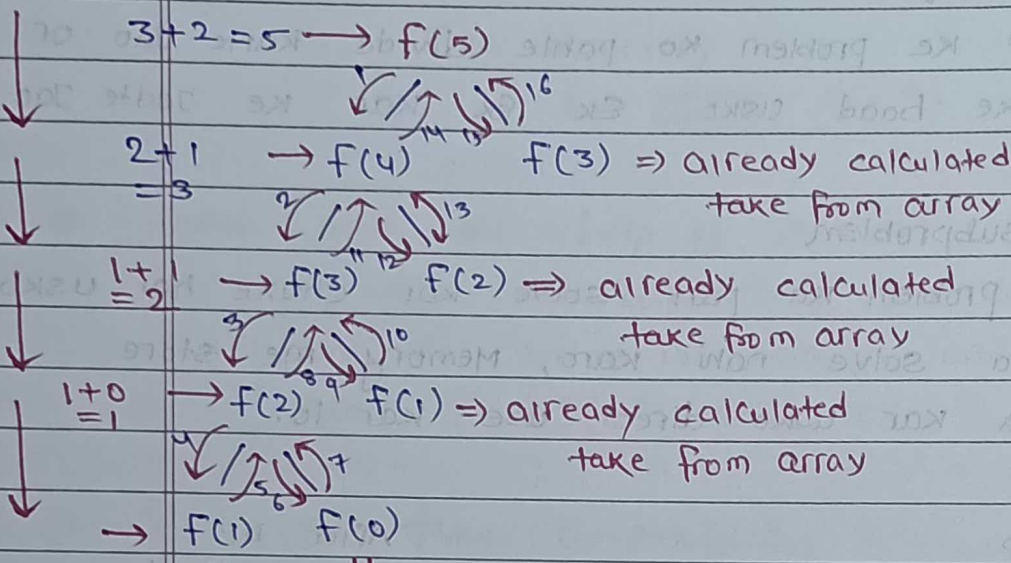
Initially all -1

Initially

F(0) = 0

F(1) = 1

0	1	2	3	4	5
0	1	-1	-1	-1	-1



Top down Approach:

Code:

```
int find (int n, vector<int> &dp) {
        if( n <= 1)
            return n;

        if (dp[n] != -1)
            return dp[n];

        dp[n] = find(n-1, dp) + find((n-2), dp);
        return dp[n];
}
int nthFibonacci (int n){

        vector <int> dp (n+1, -1);
        return find(n, dp);

}
```

Space Complexity : $O(n) + O(n) \Rightarrow O(n)$

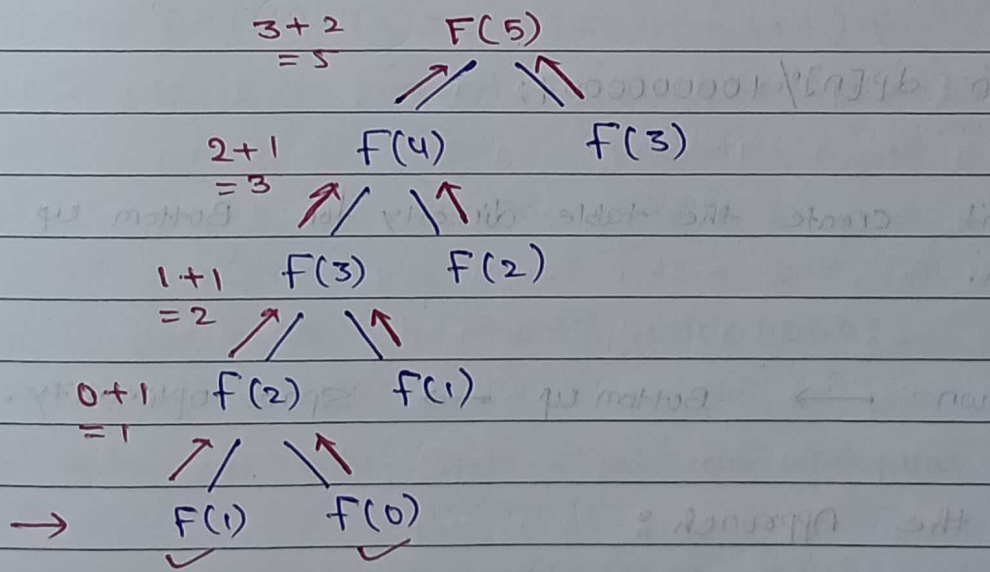
↓ dP ↓ Recursive call

Bottom up Approach :

Jitna bhi boottom ke result hai usko solve kar lo.

Make a table of size n+1.

0	1	2	3	4	5
0	1	1	2	3	5



sabse pahle bottom walo ka → $f(0)$, $f(1)$

↳ Next $f(2) \rightarrow f(1) + f(0) = 1$

↳ Next $f(3) \rightarrow f(2) + f(1) = 2$

↳ Next $F(4) \rightarrow f(3) + f(2) = 3$

↳ Next $f(5) \rightarrow f(4) + f(3) = 5$

→ 0 & 1 ke liye value assign kar denge.

→ 2 se n tak ke liye loop laga denge.

Tabulation Method: (Bottom up Approach)

→ code:

```
int nthFibonacci (int n) {

    vector <int> dp(n+1);
    dp[0] = 0;
    dp[1] = 1;

  for(int i= 2; i<=n; i++){
      dp[i] = (dp[i-1] + dp[i-2])% 1000000007;
    }
  return dp[n]%1000000007;
}
```

We can't create the table directly for Bottom up Approach.

Top down ⟶ Bottom up ⟶ Space optimality.

⇒ Optimise the Approach:

$$f(n) = f(n-1) + f(n-2)$$

f(0)	f(1)	f(2)
0	1	1

↑

for this we required f(0) & f(1)

f(0)	f(1)	f(2)	f(3)
0	1	1	2

↓

for this
we require
f(1) & f(2)
We don't require f(0), so we delete.

We only require 2 previous value. So, we delete the first value in each step.

Code: (Running)

```
int first  = 0;
int second = 1;
int result;
int third;
for (int i = 2;  i <= n;  i++) {
        third = (first + second) % 1000000007;
        first  = second;
        second = third;
    }
result = third % 1000000007;
return result;
```

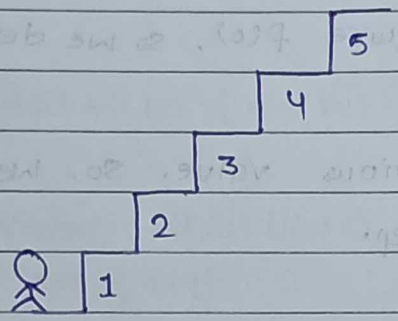
→ On this we only store the two previous value.

Space Complexity : O(1)
Time Complexity : O(N)

* Climbing stairs : (Leetcode)

Example :

N = 5



⇒ At 1 time take 1 step or 2 step

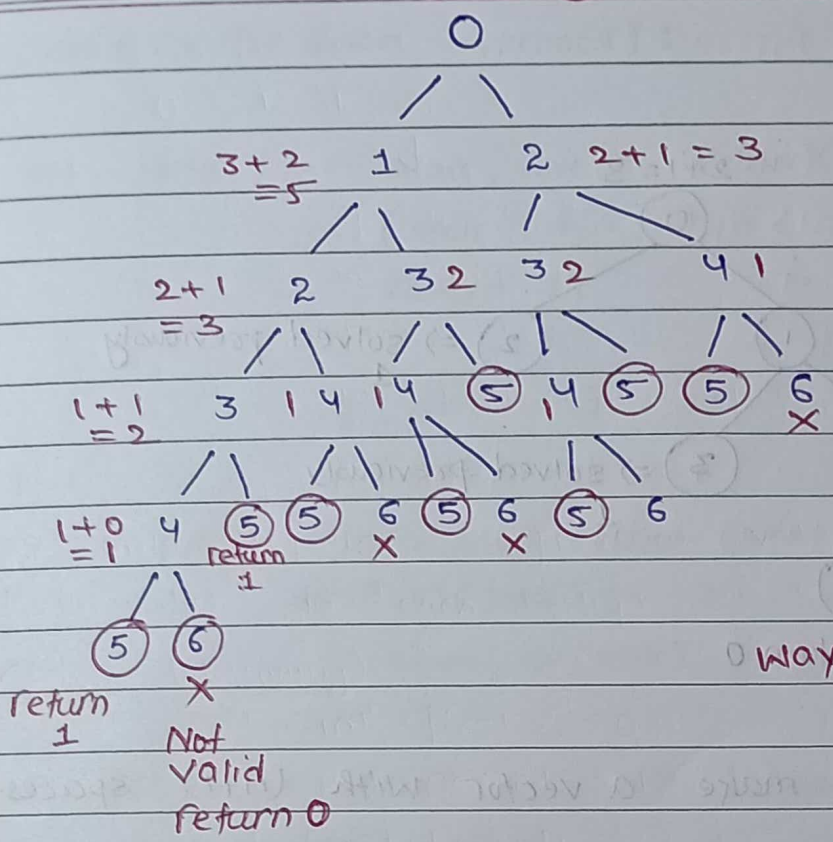
⇒ Find total No. of ways to reach on top ?

→ 1 step + 1 step + 1 step + 1 step + 1 step
→ 1 step + 1 step + 1 step + 2 step
→ 1 step + 1 step + 2 step + 1 step
→ 1 step + 2 step + 1 step + 1 step
→ 2 step + 1 step + 1 step + 1 step
→ 1 step + 2 sep + 2 step
→ 2 step + 1 step + 2 step
→ 2 step + 2 step + 1 step

Total = 8 ways.

We can understand this using graph of stairs.

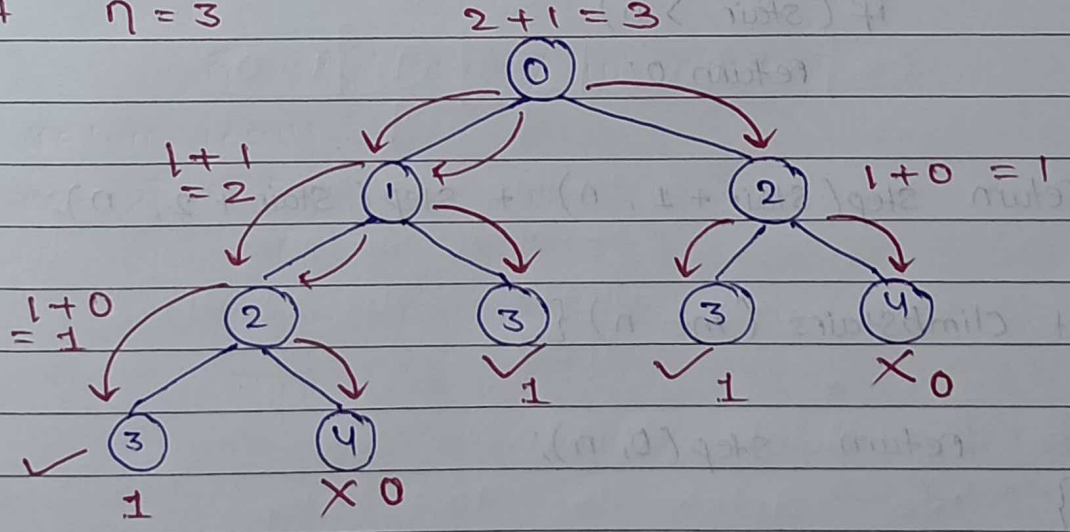
5+3 = (8) → Answer.



Ways = 8

```
n == 5
return 1;
n > 5
return 0;
```

Let n = 3



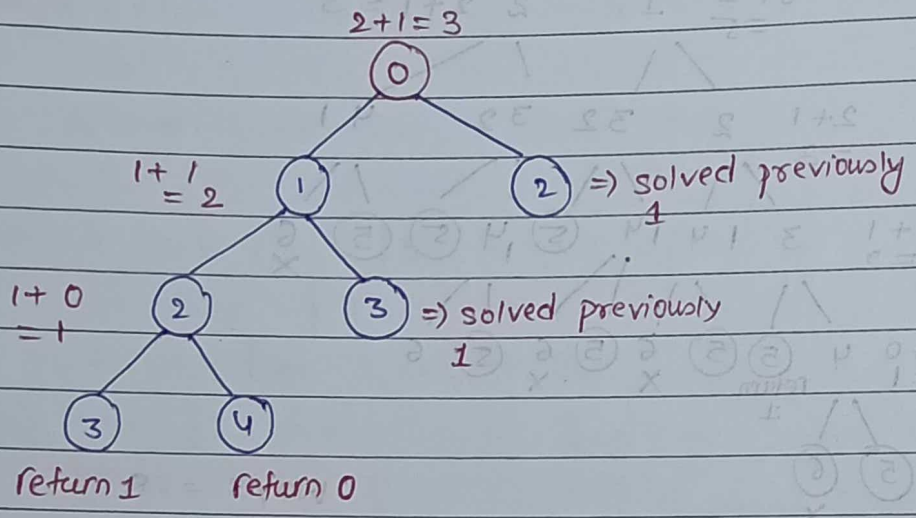
$f(n) = f(n+1) + f(n+2)$

n → return 1

>n → return 0

Optimise :

n = 3



On this we make a vector with (n+1) spaces.

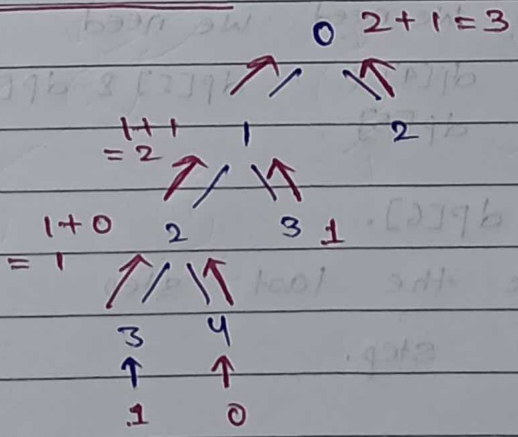
Code (By recursion) :

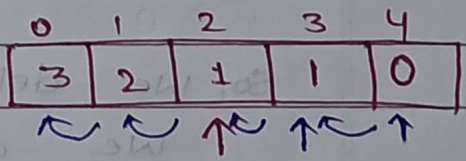
```
int    step (int stair, int n) {
            if (stair == n)
              return 1;
            if (stair > n)
              return 0;

  return  step(stair + 1, n) + step(stair + 2, n);
}
int climbStairs (int n) {

      return  step(0, n);
}
```

Code (By Top down Approach) :

```
int step (int stair, int n, vector<int> &dp) {
        if (stair == n)
            return 1;

        if (stair > n)
            return 0;
    // Already calculated result return
        if (dp[stair] != -1)
            return dp[stair];

    dp[stair] = step(stair+1, n, dp) + step(stair+2, n, dp);
     return dp[stair];
}

int climbStairs (int n) {
      vector <int> dp (n+2, -1);
      dp[n] = 1,  dp[n+1] = 0;
      return   step (0, n, dp);
   }
```

Bottom up Approach :





f(3)+f(4)
= 1

f(1) = f(2) + f(3)
= 2

f(0) = f(1) + f(2) = 3

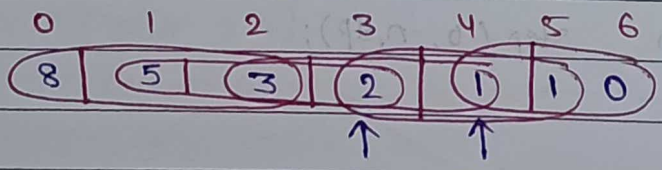
O(N) → space Complexity

Code (Bottom up Approach):

```
int climbStairs(int n) {
    vector<int> dp(n+2);
    dp[n] = 1;
    dp[n+1] = 0;

    for(int i = n-1; i >= 0; i--)
        dp[i] = dp[i+1] + dp[i+2];

    return dp[0];
}
```

Time Complexity : O(N)
Space Complexity : O(N).

* optimise space Complexity :



For this we need dp[4] & dp[5]

For this we need dp[5] & dp[6]

So, we delete dp[6].
We delete the last step in every step.

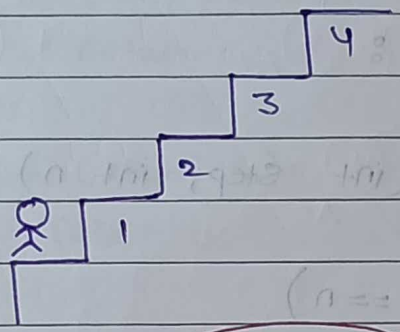
* Count Number Of Hops :

At one frog can jump either one, two or three steps.

1 or 2 or 3

Example :

N = 4



possible :

1 + 1 + 1 + 1

1 + 1 + 2

1 + 2 + 1

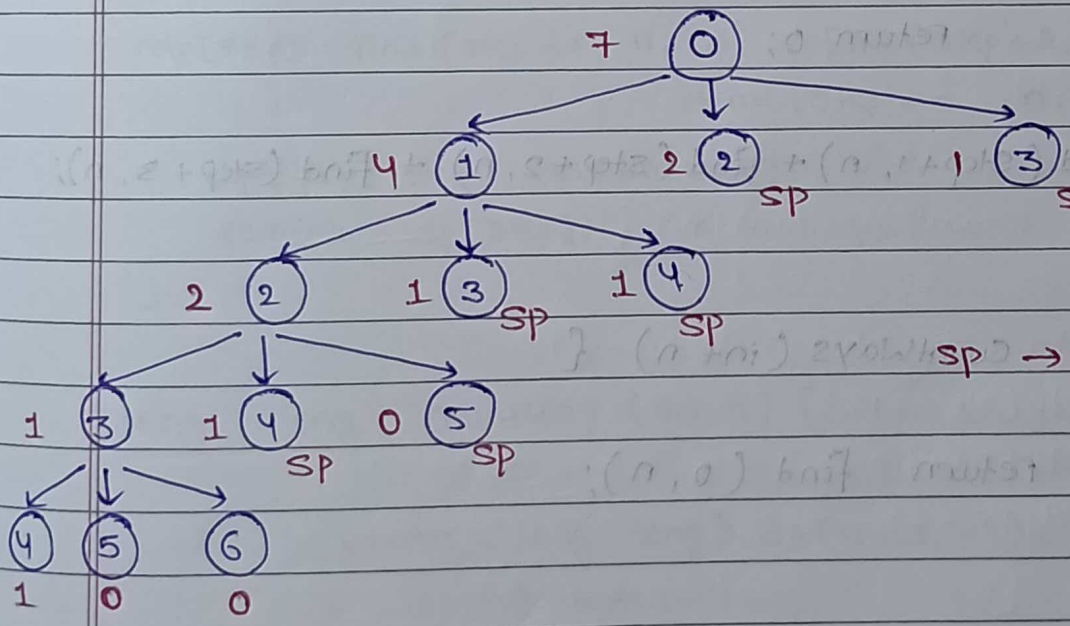
2 + 1 + 1

2 + 2

1 + 3

3 + 1

Total = 7



sp → solved previously.

For n → return 1

For n+1 → return 0

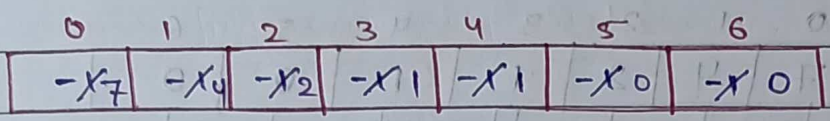
For n+2 → return 0

```
if (step == n)
    return 1

if (step > n)
    return 0;

return For 1 step + For 2 step + For 3 step;
```

Code (by recursion):

```
long long find (int step, int n) {

    if ( step == n)
        return 1;

    if ( step > n)
        return 0;

return find (step+1, n) + find (step+2, n) + find (step+3, n);
}

long long countWays (int n) {

        return find (0, n);
  }
```

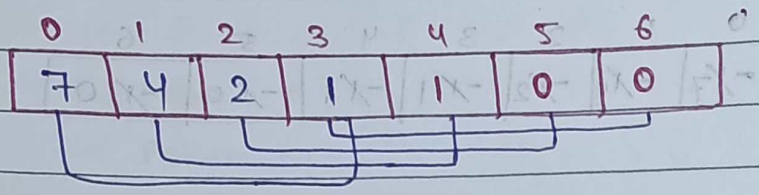
Code (by top down Approach) :

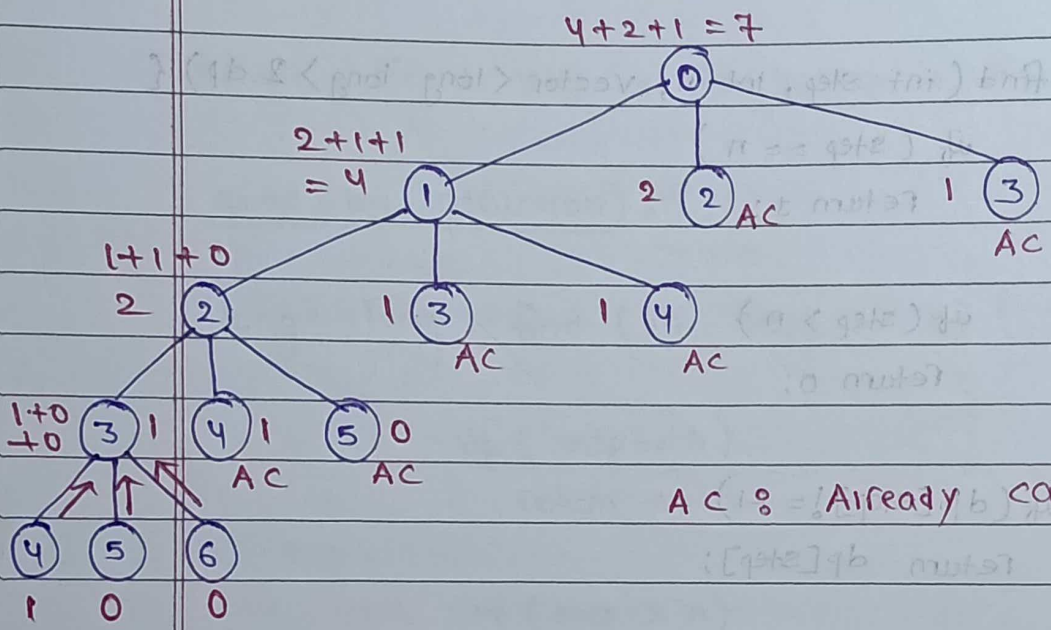


code :

```
long long find(int step, int n, vector<long long>& dp) {
        if (step == n)
            return 1;

        if (step > n)
            return 0;

        if (dp[step] != -1)
            return dp[step];

    dp[step] = find(step+1, n, dp) + find(step+2, n, dp) +
                                    find(step+3, n, dp);
    dp[step] %= 1000000007;
    return dp[step];
}

long long countWay(int n) {

        vector<long long> dp(n+3, -1);
        dp[n] = 1;
        dp[n+1] = 0;
        dp[n+2] = 0;

        return find(0, n, dp);

}
```

Bottom up Approach:





AC : Already calculated.

code :

```
long long countWays(int n) {

    vector<long long> dp(n+3);
    dp[n] = 1;
    dp[n+1] = 0;
    dp[n+2] = 0;

  for (int i = n-1; i>=0; i--){
      dp[i] = (dp[i+1] + dp[i+2] + dp[i+3]) % 1000000007;
  }
  return dp[0];
}
```

Lecture - 70

House Robber Coin Change

* House Robber (Leetcode)

You don't enter into two adjacent house, if you enter the alarm starts ringing.

You have an array "nums" representing the amount of money at each house.

Return the maximum amount of money you rob tonight without ringing the alarm.

nums = {1, 2, 3, 1}

possible Combination :

1 + 3 = 4 (Max) return
1 + 1 = 2
2 + 1 = 3

We can understand this with help of tree.

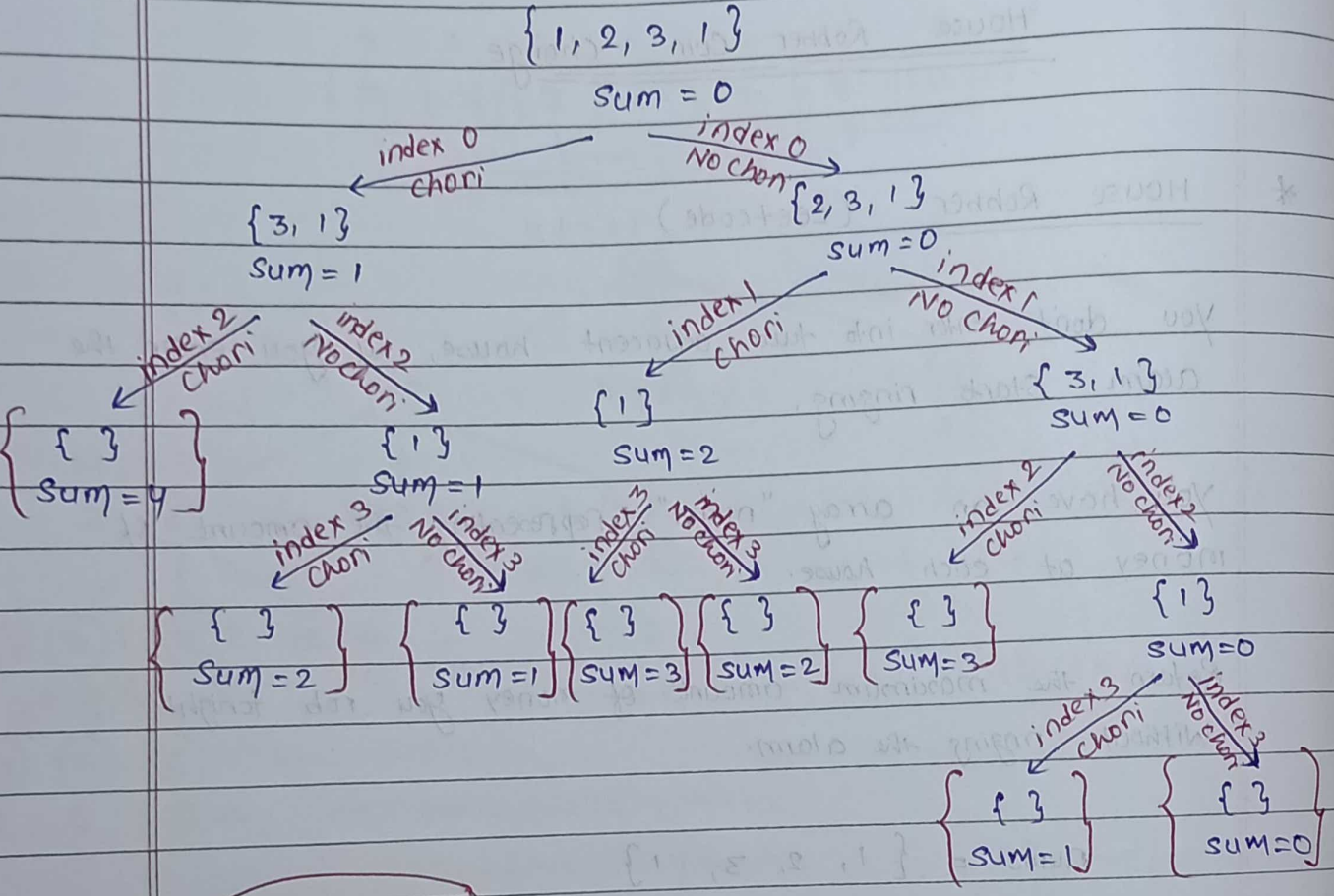
We start with all houses.

→ 1st house se chori kar liye to uske baad wala hata denge option se.

By recursion :

House :

0	1	2	3
1	2	3	1



Max = 4

return 4.

nums = {1, 2, 3, 1}

House = {1, 2, 3, 1}

find (index + 1)

↳ Agar chori nahi kiye to

(House[index] + find(index + 2))

↳ Agar chori kar liye to

Dono me se jo max ayega wo answer hoga.

Base case :

```
if(index == n)
    return 0;
```

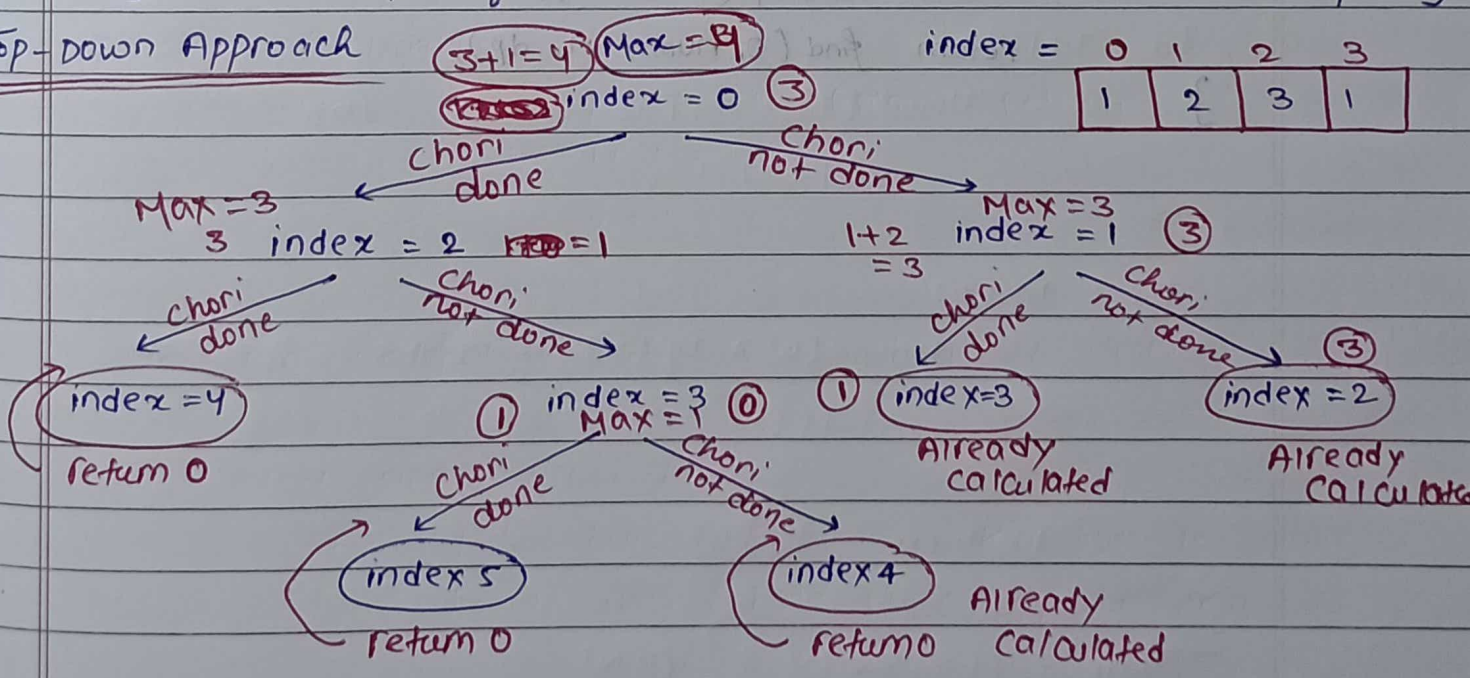
Code :

```
int find (int index, vector<int>& nums, int n) {

        if (index >= n)
            return 0;

    return max (nums[index] + find (index+2, nums, n),
                find (index+1, nums, n));

}

int rob (vector<int>& nums) {
        int n = nums.size();
        return find (0, nums, n);
    }
```

Time Complexity : $O(2^n)$ (TLE)

By the help of DP we can decrease the Time Complexity.

Top-Down Approach



Code : (Top down Approach)

```
int find(int index, vector<int>& nums, int n, vector<int>& dp){

        if(index >= n)
            return 0;

        if(dp[index] != -1)
            return dp[index];

return dp[index] = max(nums[index] + find(index+2, nums, n, dp),
                       find(index+1, nums, n, dp));
}

int rob(vector<int>& nums){
        int n = nums.size();
        vector<int> dp(n+2, -1);
        return find(0, nums, n, dp);
}
```

Bottom up Approach:

	0	1	2	3
House :	1	2	3	1

	0	1	2	3	4	5
DP :	4	3	3	1	0	0

DP[n] = 0

DP[n+1] = 0

For DP[3]:

House[3] + DP[5] = 1 Max

DP[4] = 0

For DP[2]

House[2] + DP[4] = 3 Max

DP[3] = 1

For DP[1]

House[1] + DP[3] = 3 Max

DP[2] = 3

For DP[0]

House[0] + DP[2] = 1 + 3 = 4 Max

DP[1] = 3 = 3

return DP[0]

Time Complexity : O(N)

space Complexity : O(N)

code :

```
int rob(vector<int>& nums) {
        int n = nums.size();
        vector<int> dp(n+2);
        dp[n] = dp[n+1] = 0;
        for(int i=n-1; i>=0; i--)
          dp[i] = max(nums[i]+dp[i+2], dp[i+1]);

    return dp[0];
}
```

* More optimised in space complexity :
We doesn't require full size of vector dp.
We only need two value in each step.

So, we don't make any dp vector.

Goal : space Complexity : O(1)

dp[n] → First
dp[n+1] → second

```
ans = max(nums[i] + second, first);
        second = first;
        first = ans;
```

Time Complexity : O(N)

Space Complexity : O(1)

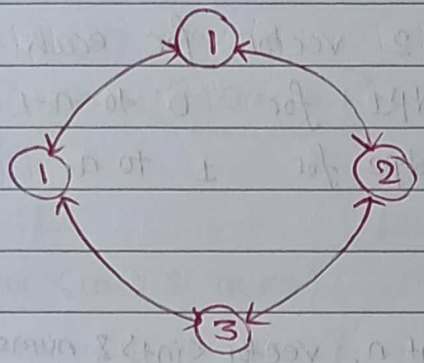
Code :

```
int rob(vector<int>& nums){
        int n = nums.size();

        int first = 0, second = 0;
        int ans;

      for(int i=n-1; i>=0; i--){
          ans = max(nums[i]+second, first);
           second = first;
           first = ans;
      }

   return ans;
}
```

* House Robber II (Leetcode)

The last house is connected with first house in circular form.

0	1	2	3
1	2	3	1



On this. We give an extra condition here:

Ya to 0 se n-1 tak kar lo ya 1 se n tak

code: (Recursion)

```
int find (int index , int n, vector <int> & nums ) {
        if (index >= n)
            return 0;

    return max (nums [index] + find (index + 2, n, nums),
                find (index + 1, n, nums));
}
int rob (vector <int> & nums) {
        int n = nums. size();
    return max (find (0, n-1, nums) , find ( 1, n, nums));
}
```

```
{ if (n == 1)
    return nums[0];
```

Top Down Approach:

We find in 2 ways:

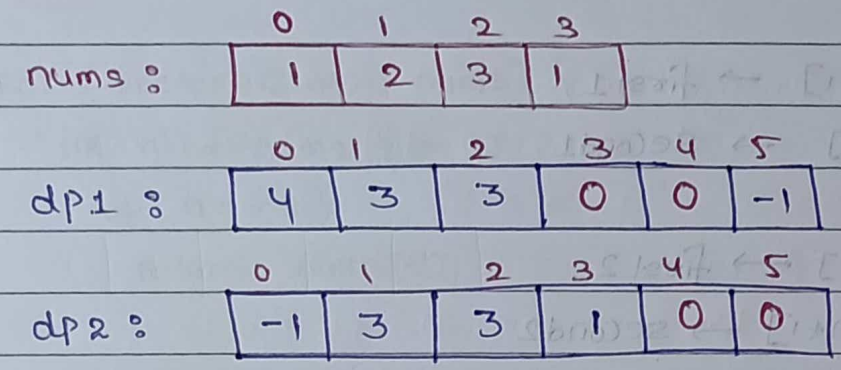
① 0 to n-1
② 1 to n

So, We make 2 vector for each.
1 vector dp1 for 0 to n-1
2nd vector dp2 for 1 to n.

Code:

```
int find(int index, int n, vector<int>& nums, vector<int>& dp) {
        if (index >= n)
            return 0;

        if (dp[index] != -1)
            return dp[index];

return dp[index] = max(num[index] + find(index+2, n, nums, dp),
                       find(index+1, n, nums, dp));
}
int rob(vector<int>& nums) {
        int n = nums.size();
        if(n==1)
            return nums[0];

        vector<int> dp1 (n+2, -1);
        vector<int> dp2 (n+2, -1);

return max(find(0, n-1, nums, dp1), find(1, n, nums, dp2));
}
```

Bottom up Approach:



dp1[0], dp2[1]

(4, 3) → Max = 4 Ans

code:

```
int rob(vector<int>& nums){
    int n = nums.size();
    if(n==1)
    return nums[0];

    vector<int> dp1(n+2, -1);
    vector<int> dp2(n+2), -1);

dp1[n-1] = dp1[n] = dp2[n] = dp2[n+1] = 0;

for(int i=n-2; i>=0; i--)
dp1[i] = max(nums[i] + dp[i+2], dp[i+1]);

for(int i=n-1; i>0; i--)
dp2[i] = max(nums[i] + dp[i+2], dp[i+1]);

return max(dp1[0], dp2[1]);
}
```

T.C : O(N)
S.C : O(N)

* Optimise the Space Complexity:

dp1[n-1] → first1

dp1[n] → second1

dp2[n] → first2

dp2[n+1] → second2

ans1 = max(nums[i] + second1, first1);

ans2 = max(nums[i] + second2, first2);

second1 = first1

first1 = ans1

second2 = first2

first2 = ans2.

In this, we don't require any vector and by this we can minimise our space Complexity.

In top-down approach and bottom up Approach we use vector.

But we need only 2 values each time from vector.

So, we can't use vector, in place of vector we use 2 variable.

Space Complexity: O(1)

Code :

```
int rob(vector<int>& nums) {
    int n = nums.size();
    if(n==1)
      return nums[0];

    int first1 = 0;
    int second1 = 0;
    int first2 = 0;
    int second2 = 0;
    int ans1, ans2;

  for(int i=n-2; i>-1; i--) {
      ans1 = max(nums[i] + second1, first1);
        second1 = first1;
        first1 = ans1;
    }

  for(int i=n-1; i>0; i--){
      ans2 = max(nums[i] + second2, first2);
          second2 = first2;
          first2 = ans2;
    }

  return max(ans1, ans2);
}
```

classmate
Date
Page

* Coin change II (Leetcode)

Given : Integer array
↳ Coins of different denomination

Amount
↳ Total amount of money

Return : Number of Combination that make up that amount.

If not possible → return 0.

Assume you have infinite Number of each kind of Coin.

Example 1 :

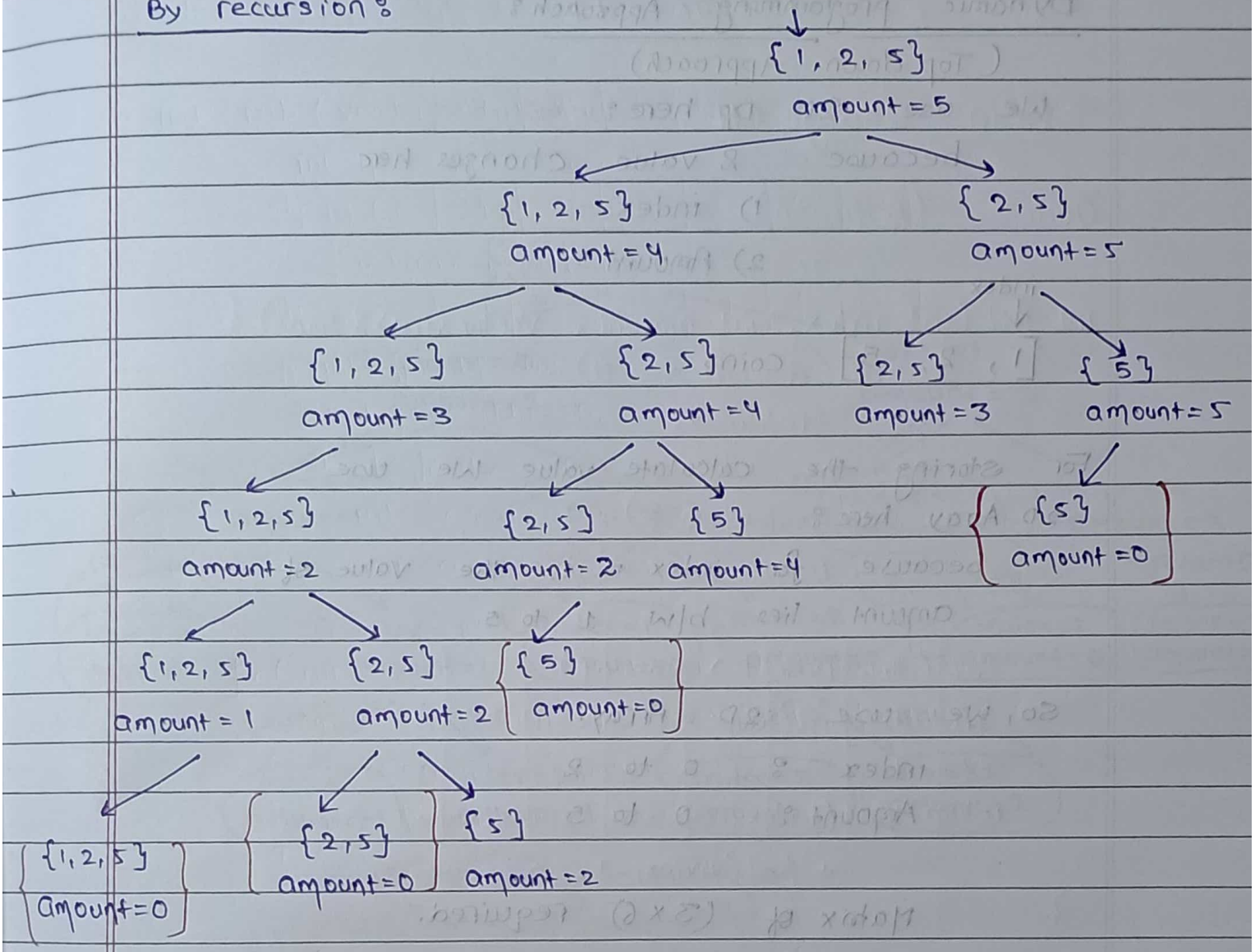
Coins : {1, 2, 5}

Amount : 5

possible : 5 = 5
5 = 2 + 2 + 1
5 = 2 + 1 + 1 + 1
5 = 1 + 1 + 1 + 1 + 1

Total = 4 possibilities.

By recursion:



Code:

```
int find (int index, int amount, vector<int>& coins, int n) {
        if (amount == 0)
            return 1;
        if (amount < 0 || index >= n)
            return 0;
return find (index, amount - coins[index], coins, n) + find (index + 1,
                amount, coins, n);
}
int change (int amount, vector<int>& coins) {
        int n = coins.size();
    return find (0, amount, coins, n);
}
```

classmate
Date
Page

Dynamic programming Approach:

(Top down Approach)

We use 2D Dp here:

because 2 value changes here

1) index
2) Amount

index
↓
[1, 2, 5] coins

For storing the calculate value we use 2D Array here:

because for index = i, the value of amount lies b/w 1 to 5.

So, we use 2D Array.

index : 0 to 2
Amount : 0 to 5

Matrix of (3 x 6) required.

Index : 0 - n => (n+1)

Amount : amount + 1

2D DP of size : (n+1) * (amount + 1)

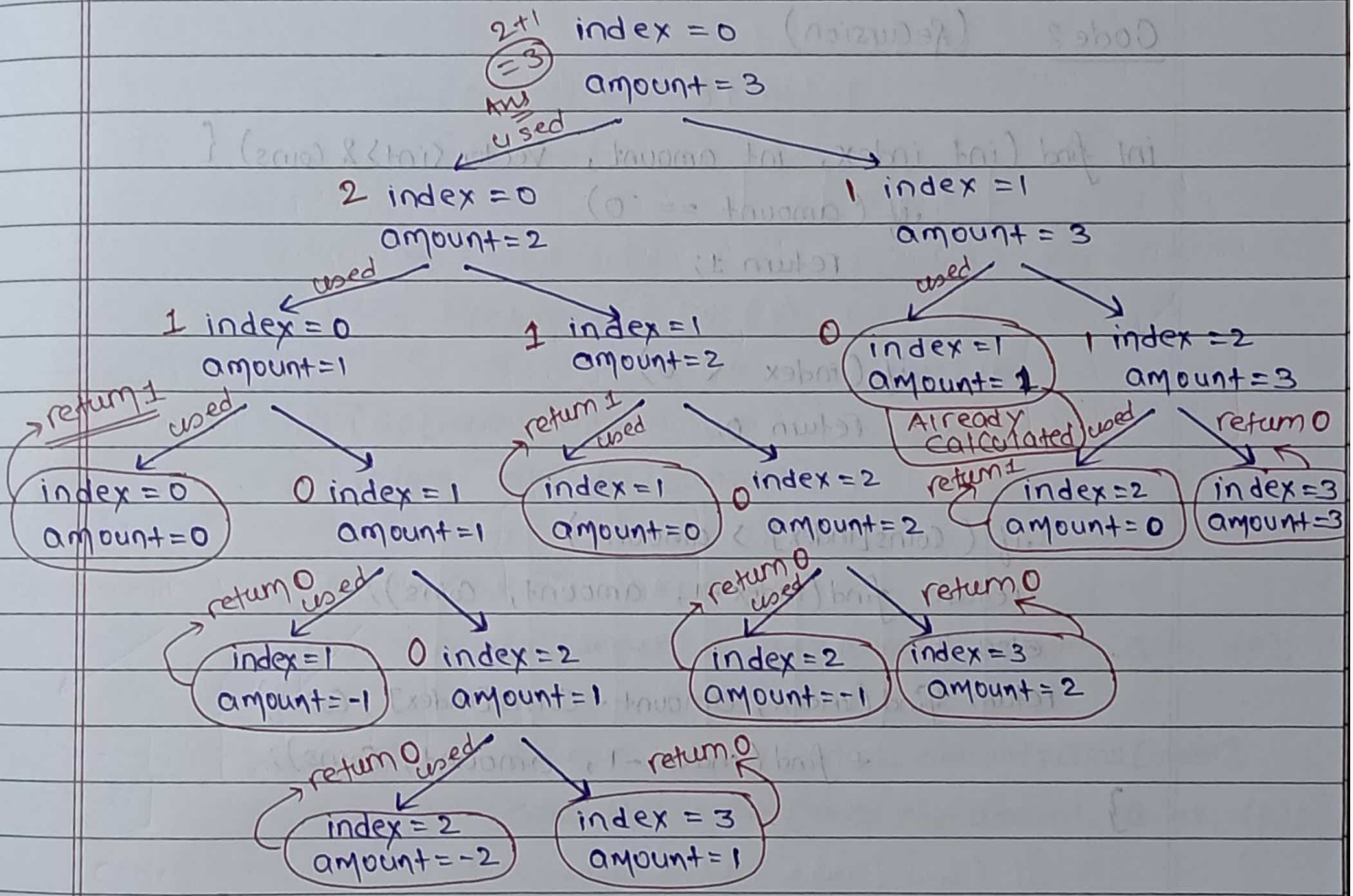
We understand with an Example:

coins : [1, 2, 3]
amount = 5

0 1 2

{ 1 , 2 , 3 }

amount = 3



Time Complexity : $O(n+1)(amount+1)$

→ If the Coins array is unsorted, Make it sorted

↳ then try to solve the problem

↳ otherwise, it gives ~~error~~ (orrect answer.

Note:

* You can solve problem with both sorted & unsorted.

Lecture - 71

Coin Change II

Code: (Recursion)

```
int find (int index, int amount, vector<int>& coins) {
        if (amount == 0)
            return 1;

        if (index < 0)
            return 0;

    if (coins[index] > amount)
     return find(index-1, amount, coins);
    else
    return find(index, amount - coins[index], coins)
            + find(index-1, amount, coins);
}

int change (int amount, vector<int>& coins) {
        int n = coins.size();
      return find(n-1, amount, coins);
    }
```

Code : (Top down Approach)

```
int find (int index, int amount, vector<int> &Coins,
                vector<vector<int>> &dp) {

        if (amount == 0)
            return 1;

        if (index < 0)
            return 0;

        if (dp[index][amount] != -1)
            return dp[index][amount];

        if ( Coins [index] > amount)
        return dp[index][amount] = find (index -1, amount, Coins, dp);
        else
        return dp[index][amount] = find (index, amount-Coins[index],
                        Coins, dp) + find (index -1, amount, Coins, dp);
}

int change ( int amount, vector <int> & Coins) {
            int n = Coins.size();

 vector <vector<int>> dp (n+1, vector<int>(amount +1, -1));
    return find ( n-1, amount, Coins, dp);
}
```

Bottom up Approach :

→ check first Base cases

```
if (amount == 0)
    return 1;
```

0	1	2
1	2	5

Amount = 5

dp[n+1][amount+1]

amount

coins	0	1	2	3	4	5
0	1	0	0	0	0	0
1	1	1	1	1	1	1
2	1	1	2	2	3	3
3	1	1	2	2	3	4

```
if (Coins[index - 1] > amount)
    find (index - 1, amount, Coins)

else
    find (index, amount - Coin[index - 1], Coins)
        + find (index - 1, amount, Coins);
```

Size : (n+1) * (amount+1)

Lecture 72

Coin Change II :
(Bottom up approach)

(i) \ (j)	0	1	2	3	4	5
(Coins) 0	1	0	0	0	0	0
(1) 1	1	1	1	1	1	1
(2) 2	1	1	2	2	3	3
(5) 3	1	1	2	2	3	4

(amount)

```
if (index == 0)
    return 0;
```

```
if (Coins[i-1] > amount)
    dp[i][j] = dp[i-1][amount];
else
    dp[i][j] = dp[i][j - Coin[i-1]] + dp[i-1][amount];
```

→ i = 1, j = 1 (J = amount)

0 1 2
{1, 2, 5}

Coins[1-1] > amount

1 > 1 (False)

else

↳ dp[1][1] = dp[1][0] + dp[0][1] = 1+0 = 1

→ i = 2, 1, j = 2

Coins[1-1] > amount

1 > 2 (false)

dp[1][2] = dp[1][1] + dp[0][2] = 1

By this, We solve all value.

Code : Bottom up Approach

```
int change (int amount, vector<int>& coins) {

    int n = coins.size();

    vector<vector<int>> dp(n+1, vector<int>(amount + 1, 0));

    for (int i=0; i<=n; i++) {
        dp[i][0] = 1;

    for (int i=1; i<=n; i++) {
        for (int j=1; j<=amount; j++) {

            if (coins[i-1] > j)
            dp[i][j] = dp[i-1][j];
            else
            dp[i][j] = dp[i][j] - coins[i-1] + dp[i-1][j];
        }
    }
    return dp[n][amount];
}
```

Time Complexity : O(n+1)(amount+1)

Space Complexity : O(n+1)(amount+1)

Optimise the space Complexity :

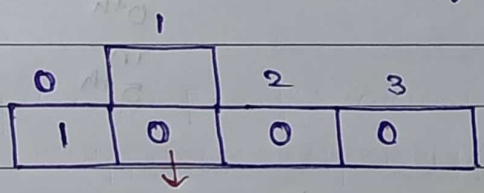
We have to fill the 2D array and we need the last value as the answer.

For finding any value of 2D DP, we require only two rows for finding value.

① previous row
② Current row

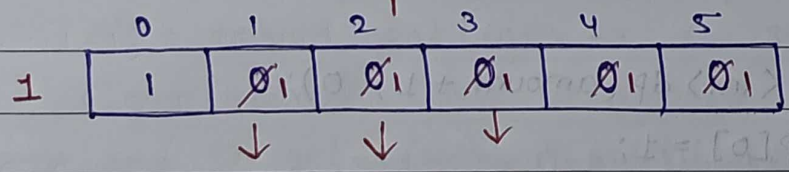
We want to optimize more :

In previous row, We only need one value which is upper of my value (Current).



For this we require that row and a upper block.

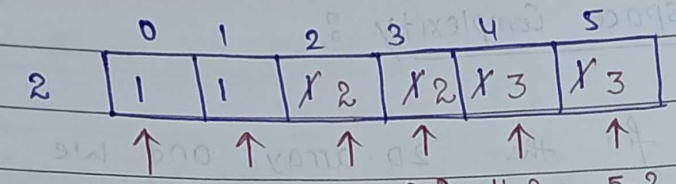
(2-1) = ① → 1 wala add kar do



1 rupee ke Coin se 1 rupya bnnahai kitna possible ways hai

(3-1) = 2 → 2 wala add kar do

3 - 1 = ②
↓ ↓
rupee Coin
↳ rupee (value)
↳ add in 3 col.



0-2 = -2 does not exist

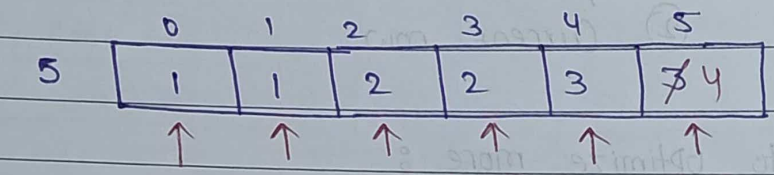
1-2 = -1 does not exist

2-2 = 0 Add 0th elem in 2nd

3-2 = 1 Add 1th in 3rd

4-2 = 2 Add 2nd in 4th

5-2 = 3 Add 3rd in 5th



0-5 = -5 X

1-5 = -4 X

2-5 = -3 X

3-5 = -2 X

4-5 = -1 X

5-5 = 0 ↑ Add 0th in 5th

Final answer = 4

code :

```
int change (int amount, vector<int> & Coins) {
        int n = Coins.size();

    vector <int> dp (amount + 1, 0);
        dp[0] = 1;

    for (int i = 1; i<=n; i++) {
        for (int j = Coins[i-1]; j<=amount; j++) {
            dp[j] += dp[j - Coins[i-1]];
        }
    }

    return dp[amount];
}
```

* Count Ways to N'th stair (Order does not matter)

→ There are N stairs.

→ person can climb either 1 stair or 2 stair at a time.

→ Here {1, 2, 1}, {2, 1, 1}, {1, 1, 2} are considered as same.
Because frequency of 1 & 2 are same in all.

Code : Recursion

```
int find(int index, int n, int step[]) {
        if (n == 0)
            return 1;

        if (index == 0)
            return 0;

        if (step[index - 1] > n)
            return find(index - 1, n, step);
        else
            return find(index, n - step[index - 1], step) +
                   find(index - 1, n, step);
    }
    int nthStair(int n) {

        int step[2] = {1, 2};
        return find(2, n, step);

    }
```

On this problem
↳ 2 values changes
↳ index & n

So, 2D DP required here.

Top Down Approach (code) :

```
int find (int index, int n,, int step[], vector<vector<int>>& dp) {
        if (n == 0)
        return 1;

        if (index == 0)
        return 0;

        if (dp[index][n] != -1)
        return dp[index][n];

        if (step[index-1] > n)
        return dp[index][n] = find (index-1, n, step, dp);
        else
        return dp[index][n] = find (index, n-step[index-1],
            step, dp) + find (index-1, n, step, dp);
    }
    int nthStair (int n) {
        int step[2] = {1, 2};
    vector<vector<int>>dp(3, vector<int>(n+1, -1));
        return find (2, n, step, dp);
    }
```

Bottom up Approach : (code)

```
int nthStair (int n) {

    int step[2] = {1, 2};
    vector<vector<int>>dp (3, vector<int>(n+1, 0));

    for (int i=0; i<3; i++) {
        dp[i][0] = 1;
    }

    for(int i=1; i<=2; i++) {
        for (int j=1; j<=n; j++) {
            if (step[i-1] > j)
            dp[i][j] = dp[i-1][j];
            else
            dp[i][j] = dp[i][j-step[i-1]] + dp[i-1][j];
        }
    }
    return dp[2][n];
}
```

→ Optimised Code

```
return 1 + n/2;
```

step	ways.
1	1
2	2
3	2
4	3
5	3
6	4
7	4
8	5
9	5

* Fractional knapsack :

Given : Weight and values of N items

Task : put the items in knapsack to get Maximum total value.

0/1 knapsack means, item will be keep totally.

you ~~dont~~ allow to break the item.

Example : 1

N = 3 W = 50

Values [] = { 60, 100, 120 }

Weight [] = { 10, 20, 30 }.

Valuesperunit [] = { 6, 5, 4 }

Take first the maximum from valuesperunit

(1) 6

Weight = 10

10 < 50

ans = 6 x 10 = 60

W = 50 - 10 = 40

(2) Next larger = 5

Weight = 20

20 < 40

ans = 60 + 20 x 5

= 160

$W = 40 - 20 = 20$

(3) next larger = 4

weight = 30

$30 > 20$

We take only 20 kg

$ans = 160 + 4 \times 20$

$= 240.00.$

↳ ans.

* Knapsack 0-1

We allowed to take either full weight of an item or we cannot take that item.

put the item in knapsack to get Maximum total values.

$N = 3$ $W = 50$

values[] = {60, 100, 120}

weight[] = {10, 20, 30}

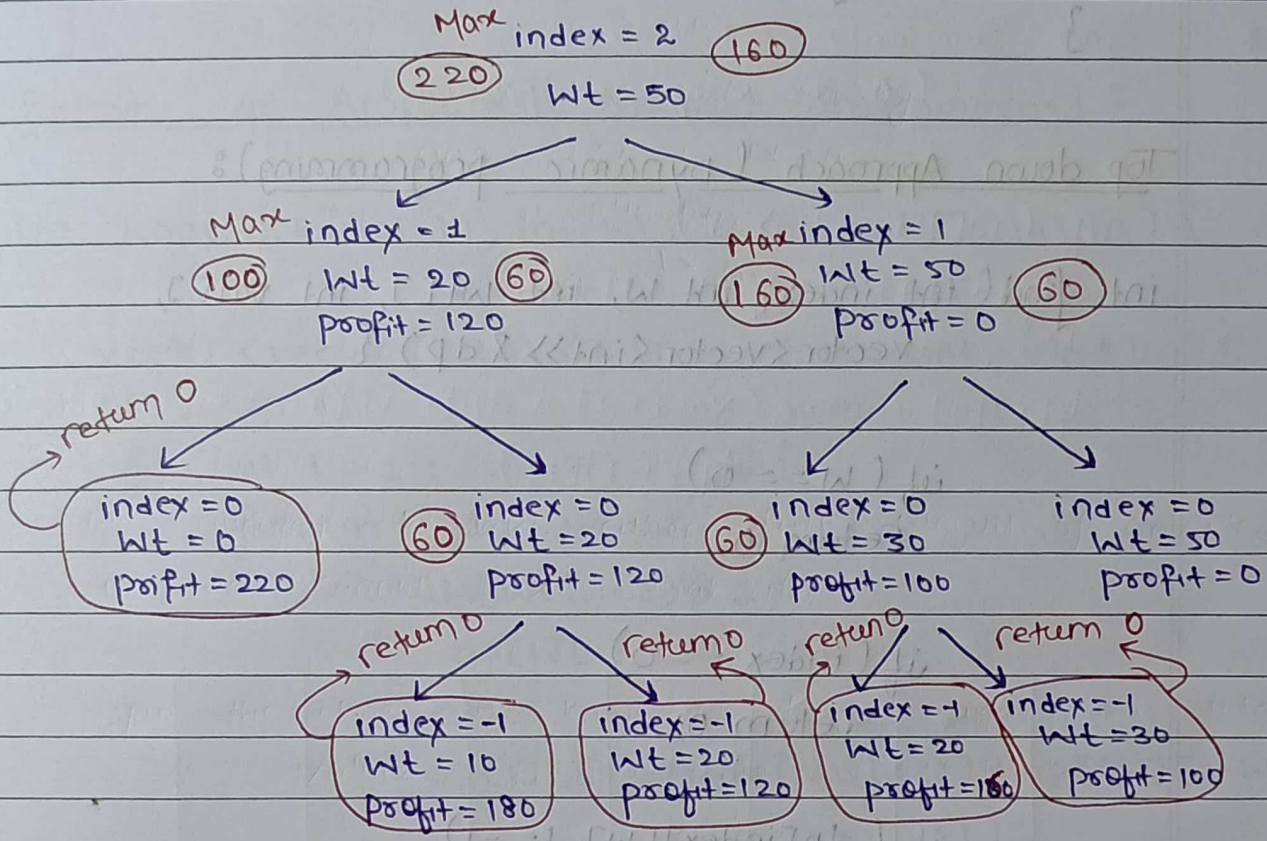
In this we solve by recursion tree.

Either we take the full Item or we don't take it.

0 1 2

Val = {60, 100, 120}

Wt = {10, 20, 30}



Top down Approach: (Recursion)

```
int find (int index, int W, int Wt[], int val[]) {
        if (W == 0)
        return 0;

        if (index == 0)
          return 0;

        if( Wt[index - 1] > W)
        return find (index - 1, W, Wt, val);

        else
        return max (val[index-1] + find(index-1, W-Wt[index-1],
                Wt, val), find(index-1, W, Wt, val);
}
```

```
int knapsack (int W, int wt[], int val[], int n)
 {

    return find (n, W, wt, val);
 }
```

Top down Approach (Dynamic programming):

```
int find( int index, int W, int wt[], int val[],
          vector<vector<int>> &dp) {

      if (W == 0)
        return 0;

      if (index == 0)
        return 0;

      if ( dp[index][W] != -1)
        return dp[index][W];

    if ( wt[index] > W)
    return dp[index][W] = find ( index -1, W, wt, val, dp);

    else
    return dp[index][W] = max (val[index-1] +
      find( index-1, W-wt[index-1], wt, val, dp),
      find ( index-1, W, wt, val, dp));

}
```

```
int knapsack(int W, int wt[], int val[], int n) {
    vector<vector<int>> dp(n+1, vector<int>(W+1, -1));

    return find(n, W, wt, val, dp);
}
```

Bottom up Approach (Dynamic programming) :

```
int knapsack(int W, int wt[], int val[], int n) {

    vector<vector<int>> dp(n+1, vector<int>(W+1, 0));

    for(int i=1; i<=n; i++) {
        for(int j=1; j<=W; j++) {
            if(wt[i-1] > j)
                dp[i][j] = dp[i-1][j];
            else
                dp[i][j] = max(val[i-1] + dp[i-1][j-wt[i-1]],
                               dp[i-1][j]);
        }
    }
    return dp[n][W];
}
```

* Optimise the Space Complexity :

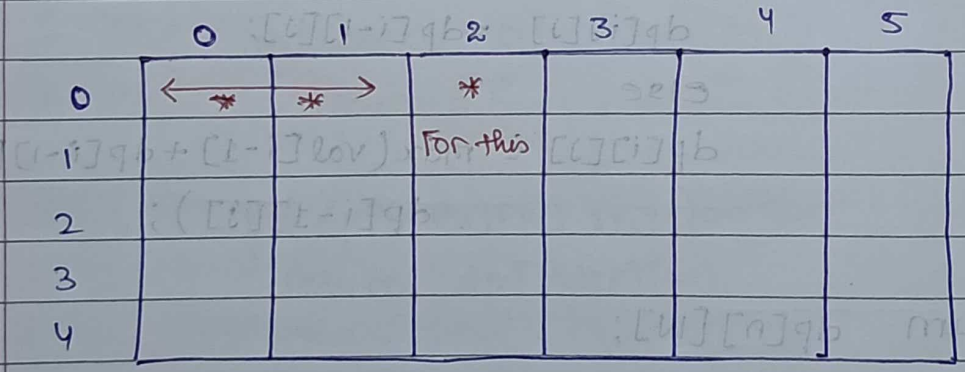
```
int knapsack(int W, int wt[], int val[], int n) {
    vector<int> dp(W+1, 0);

    for(int i=1; i<=n; i++) {
        for(int j=W; j>=wt[i-1]; j--) {
            dp[j] = max(dp[j], val[i-1] + dp[j-wt[i-1]]);
        }
    }
    return dp[W];
}
```

Lecture - 73

LCS + pallindrome

* Space Complexity optimised
(0-1 knapsack problem)

```
if (wt[i-1] > j)
  dp[i][j] = dp[i-1][j];
else
dp[i][j] = max (val[i-1] + dp[i-1][j - wt[i-1]],
                dp[i-1][j]);
```

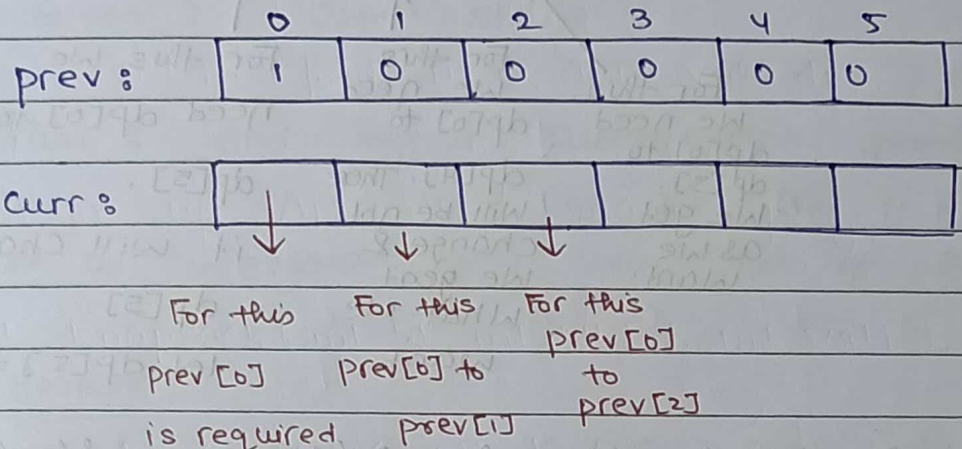


2D DP → 1D DP

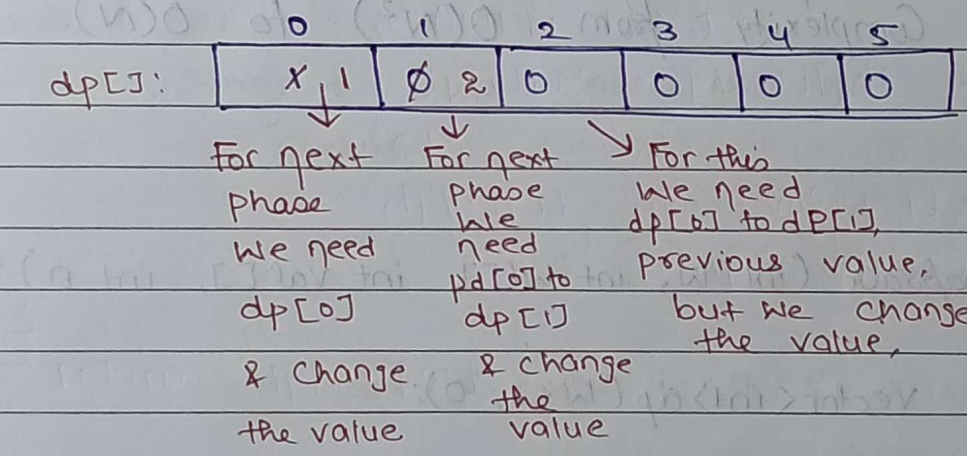
for solving dp[1][2]
we need.
↳ dp[0][2]
↳ dp[0][0] todp[0][1]

these all are from upper row.

So, insted of whole 2D-DP we only need 1 row for solving any index.



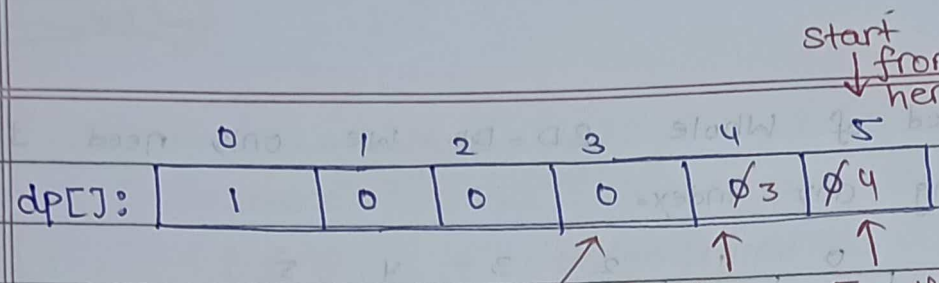
But we have to solve with one 1-D DP.
Lets try to solve:



So, by this process, it is not possible to solve this and decrease the space Complexity.

* If we do this process from the last element of the array, then the required element is not changed for any element

Let's understand with an example.



→ This is the right approach when we start solving from last.

→ So, by this method we can optimise the space Complexity from $O(N^2)$ to $O(N)$.

Code :

```
int knapsack (int W, int wt[], int val[], int n) {

        vector<int> dp(W+1, 0);

    for(int i=1; i<=n; i++){
        for(int j=W; j>0; j--) {
            if(wt[i-1] <= j)
            dp[j] = max(val[i-1]+dp[j-wt[i-1]], dp[j]);
        }
    }
    return dp[W];
}
```

* Longest Common Subsequence :

Given : Two string

Task : find longest subsequence present in both.

Let us understant with the help of Example

Example :

Str1 : A B C D G H

Str2 : A E D F H R

Dono string me se sabe bada sub string find karna hai.

↳ String jaruri nhi hai ki lagatar hona chaiye.

(A) B C (D) G (H)

(A) E (D) F (H) R

(Size : 3)

Example 2 (A) (C) D

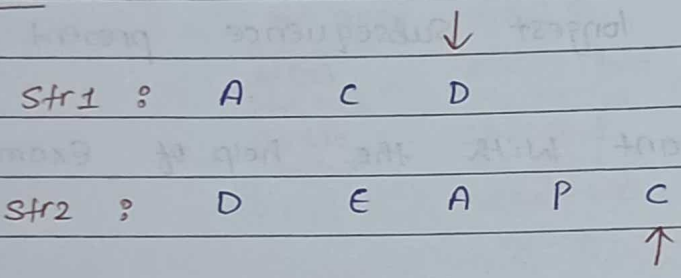
D E (A) P (C)

(Size : 2)

Real life Example : (DNA)

We have to find
↳ the longest pattern.

Approach :



Start from last

D != C

Break in two part

↳ from 1st part Remove last element of str1.

↳ on 2nd part, Remove last character of str2.

If Matched :

C == C

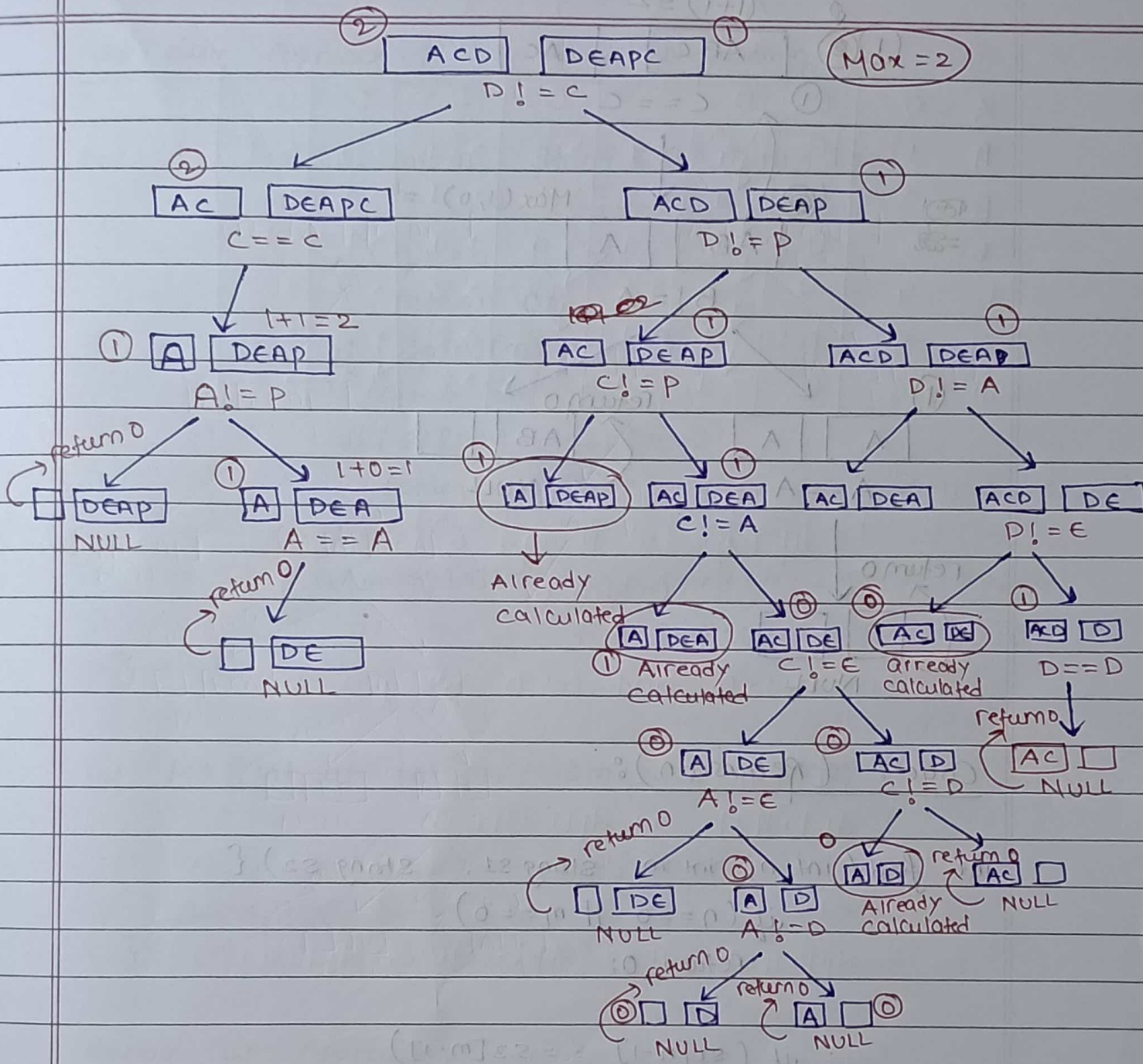
↳ then remove last alphabet from both the string.

↳ on this Condition, We can't break in two part. we only create one part.

If one string is empty :

↳ then return 0.

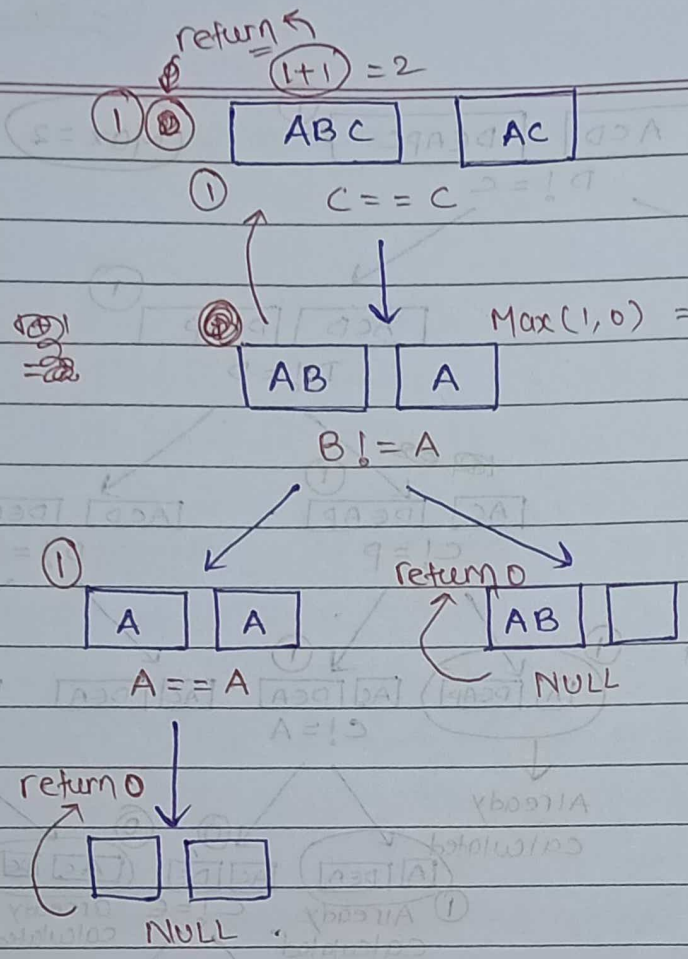
```
str1[i] == str2[j];
return 1 + find(i-1, j-1);

return max(find(i-1, j), find(i, j-1));
```



Let us Clear this Concept With a Small Example :

Str 1 : A B C

Str 2 : A C



Code : (Recursion) :

```
int find (int n, int m, string s1, string s2) {
        if (n==0 || m==0)
          return 0;

        if (s1[n-1] == s2[m-1])
        return 1 + find (n-1, m-1, s1, s2);

        else
        return max (find (n-1, m, s1, s2),
                          find (n, m-1, s1, s2));
}
int lcs (int n, int m, string s1, string s2) {

        return find (n, m, s1, s2);
}
```

Top down Approach : Dynamic programming :

```
int find (int n, int m, String &s1, String &s2,
          vector<vector<int>>& dp ) {
    if ( n == 0 || m == 0)
        return 0;
    if (dp[n][m] != -1)
        return dp[n][m];
    if ( s1[n-1] == s2[m-1])
        return dp[n][m] = 1 + find ( n-1, m-1, s1, s2, dp);
    else
    return dp[n][m] = max (find (n-1, m, s1, s2, dp),
                           find (n, m-1, s1, s2, dp));
}

int lcs ( int n, int m, string s1, string s2) {

    vector <vector<int>> dp (n+1, vector <int> (m+1, -1));
    return find(n, m, s1, s2, dp);
}
```

Bottom up Approach :

String 1 = A B C D G H

String 2 = A E D F H R

We make a 2D Dp with size of str1.length + 1 & str2.length + 1.

dp[7][7];

			A	E	D	F	H	R
		0	1	2	3	4	5	6
~~A~~	0	0	0	0	0	0	0	0
A	1	0	A = A 1	1	1	1	1	1
B	2	0	1	1	1	1	1	1
~~D~~C	3	0	1	1	1	1	1	1
D	4	0	1	1	2	2	2	2
G	5	0	1	1	2	2	2	2
H	6	0	1	1	2	2	3	3

```
if (n==0 || m==0)
    return 0;
```

In row 0 and Col 0 we put 0.

```
if(MATCH)
dp[i][j]= 1+ dp[i-1][j-1];

else
dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
```

Common String: ADH

Code:

```
int lcs (int n, int m, String s1, String s2) {

    vector<vector<int>>dp (n+1, vector<int> (m+1, 0));
    for (int i = 1; i<=n ; i++) {
      for (int j=1; j<=m; j++) {
          if(s1[i-1] == s2[j-1])
          dp[i][j] = 1 + dp[i-1][j-1];
```

```
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[n][m];
}
```

Space Complexity : O(N+1)(M+1)

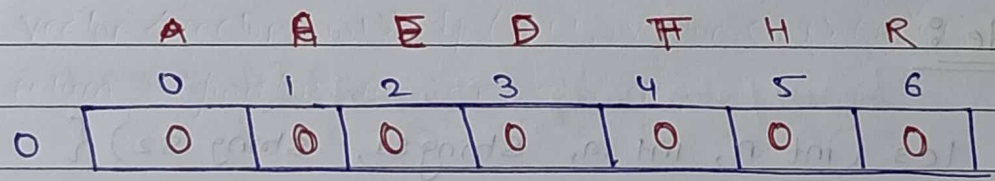
If we want to decrease the space Complexity it only depend on 3 value :

dp[i-1][j-1]

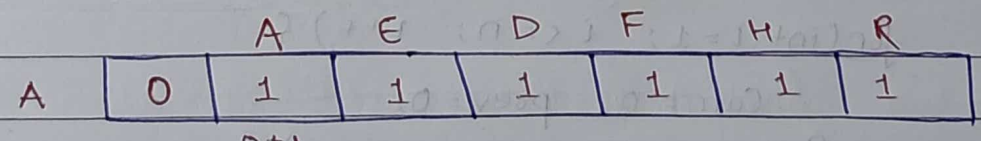
dp[i-1][j]

dp[i][j-1]

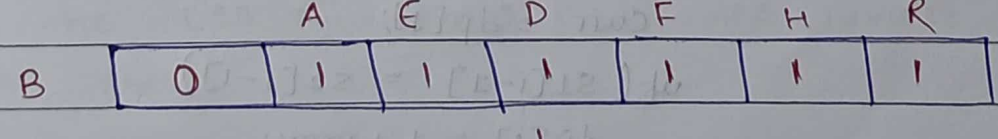
Instead of using 2D DP we will solve this only from 1D DP.



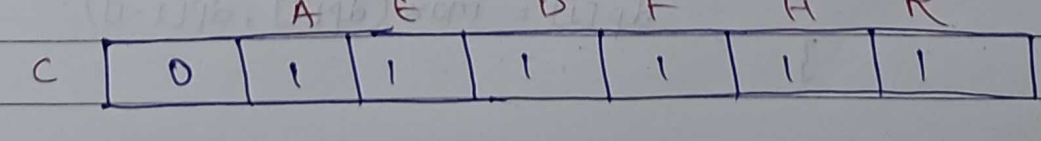
For i = 1



for i = 2



For i = 3



For i = 4

		A	E	D	F	H	R
D	0	1	1	2	2	2	2

for i = 5

		A	E	D	F	H	R
G	0	1	1	2	2	2	2

for i = 6

		A	E	D	F	H	R
H	0	1	1	2	2	3	3

return 3

Space Complexity : O(N).

Code :

```
int lcs (int n, int m, string s1, string s2) {
    vector <int> dp(m+1, 0);
     int curr , prev;
   for(int i = 1;  i <= n;  i++) {
        curr = 0,  prev = 0;
     for( int j = 1;  j <= m;  j++) {
          prev = curr;
          curr = dp[j];
        if( s1[i-1]  = s2[j-1])
           dp[j] = 1 + prev;
        else
        dp[j] = max (dp[j] , dp[j-1]);
    }
```

```
}
return dp[m];
```

* **Longest palindromic Subsequence :**

b b a b c b c a b

↳ We want longest palindromic subsequence.

palindrome :

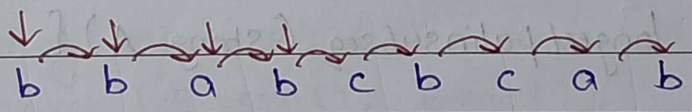
bb

bab

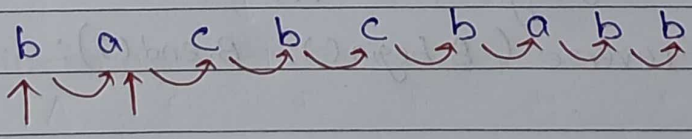
cbc

We can also delete some element if we want.

LCS :



revese :



b a b c b a b

Find the LCS of string and the reverse of the string.

The Concept is same as Longest Common Subsequence.

only we have to create a new string B, which is reverse of A.

Recursion (code):

```
int find(int n, int m, string &s1, string &s2) {
        if (n==0 || m==0)
            return 0;

        if (s1[n-1] == s2[m-1])
        return 1 + find(n-1, m-1, s1, s2);
        else
        return max(find(n, m-1, s1, s2), find(n-1, m, s1, s2));
}

int longestpalinsubseq (string A) {

        string B = A;
        reverse (B.begin(), B.end());

    return find(A.size(), B.size(), A, B);
}
```

Top Down Approach (Dynamic programming):

```
int find ( int n, int m, string &s1, string &s2,
           vector <vector <int>> &dp) {

    if (n == 0 || m == 0)
        return 0;

    if ( dp[n][m] != -1)
      return dp[n][m];

    if ( s1[n-1] == s2[m-1])
    return dp[n][m] = 1 + find (n-1, m-1, s1, s2, dp);

  return dp[n][m] = max (find(n, m-1, s1, s2, dp),
                         find(m, n-1, s1, s2, dp));
}
int longestpalinSubseq ( string A) {
        string B = A;
        reverse ( B.begin(); B.end());

vector <vector <int>> dp ( A.size()+1, vector <int> (B.size()+1,-

    return find (A.size(), B.size(), A, B, dp);

}
```

Bottom up Approach : (Dynamic programming)

```
int longestpalinsubseq (string A) {

    String B = A;
    reverse (B.begin(), B.end());

    int n = A.size();
    int m = B.size();

Vector <vector <int>> dp (A.size() + 1, vector <int> (B.size() +1, 0));

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            if (A[i-1] == B[j-1])
            dp[i][j] = 1 + dp[i-1][j-1];
            else
            dp[i][j] = max (dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[n][m];
}
```

Space Complexity : $O(N+1)(M+1)$

We can decrease the space Complexity and write the code gain.

optimal code (space complexity decrease)

```
int longestPalinSubseq ( string A) {

    String B = A;
    reverse ( B.begin(), B.end());
    int n = A.size();
    int m = B.size();

    Vector <int> dp ( m+1, 0);
    int curr, prev;

    for ( int i = 1;  i <= n; i++ ) {
        curr = 0, prev = 0;
        for ( int j = 1; j <= m; j++ ) {
            prev = curr;
            curr = dp[j];

            if ( A[i-1] = B[j-1] )
                dp[j] = 1 + prev;
            else
                dp[j] = max ( dp[j], dp[j-1] );
        }
    }
    return dp[m];
}
```

classmate
Date
Page

* Longest Repeating Subsequence :

Given : str

Task : length of longest repeating subsequence.

Example:

str : a x x z x y

0	1	2	3	4	5
a	x	x	z	x	y

1 - 2 2 - 4

2

str : a b c a b

length = 2

str : a x x z x y

Approach:

→ Copy str in str1

→ start traversing from end.

→ if matched || not matched

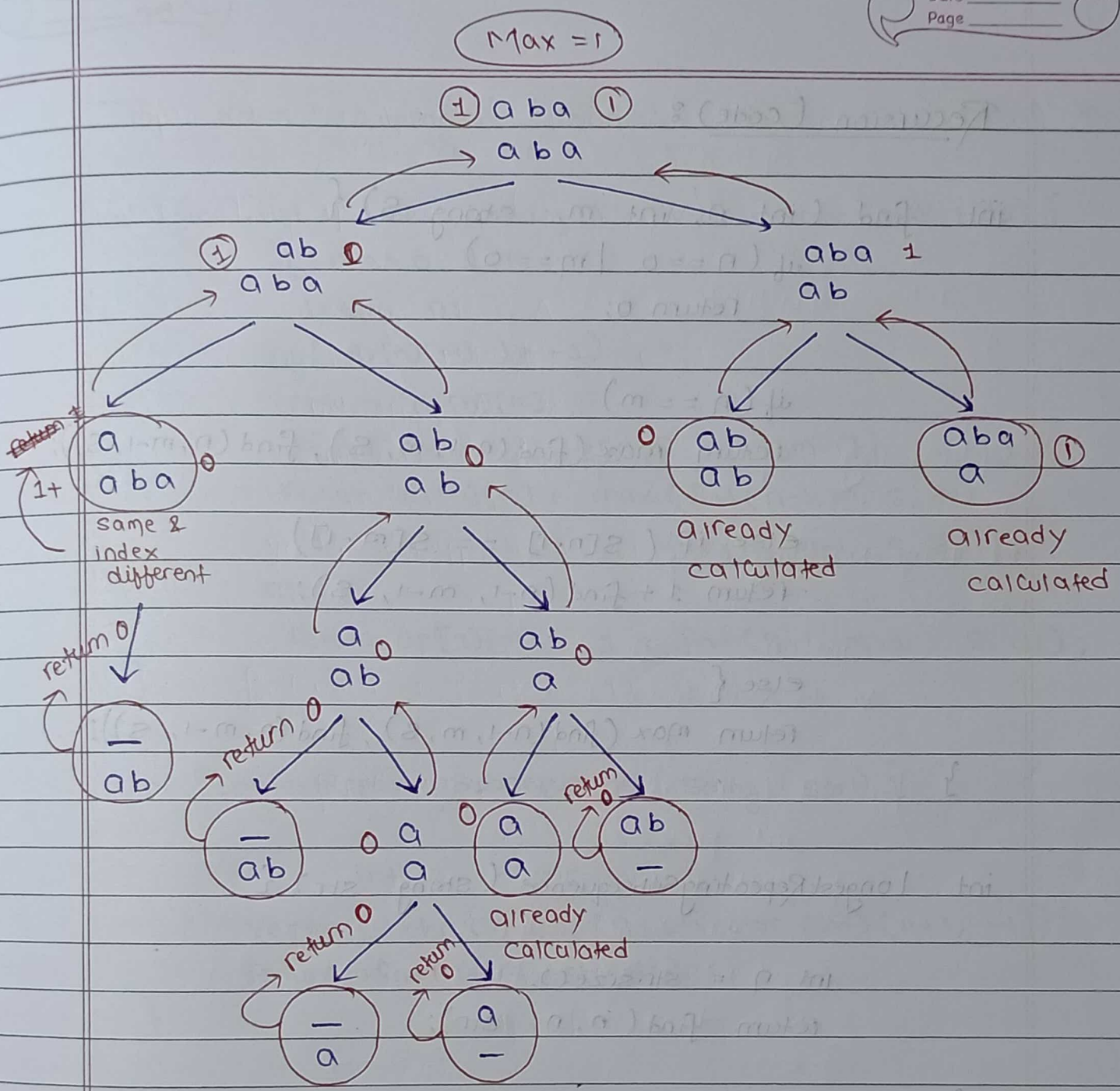
take two case

↳ remove 1 character from last from one string.

→ if matched && index different

↳ return 1 & remove from both.

Max = 1




```
if (n == m)
    return (find(n-1, m), find(n, m-1))
                                ↳ max.
else if ( s1[n-1] == s2[m-1])
    return  1 + find(n-1, m-1)

else
    return max (find(n-1, m), find(m, n-1))
```

Recursion (code) :

```
int find (int n, int m, string S) {
        if (n == 0 | m == 0)
            return 0;

        if (n == m)
        return max (find (n-1, m, S), find (n, m-1, S));

        else if ( S[n-1] == S[m-1])
        return 1 + find (n-1, m-1, S);

        else {
        return max (find (n-1, m, S), find (n, m-1, S));
}

int LongestRepeatingSubsequence (string str) {

        int n = str.size();
        return find ( n, n, str);

}
```

Top down Approach (Dynamic programming):

```
int find (int n, int m, string &s, vector<vector<int>> & dp) {
        if (n == 0 || m == 0)
          return 0;
        if ( dp[n][m] != -1)
         return dp[n][m];
        if ( n == m || s[n-1] != s[m-1])
        return dp[n][m] = max (find (n-1, m, s, dp),
                               find (n, m-1, s, dp));
       else
        return dp[n][m] = 1 + find (n-1, m-1, s, dp);
}

int LongestRepeatingSubsequence (string str) {

        int n = str.size();
    vector <vector <int>> dp (n+1, vector <int> (n+1, -1));
    return find (n, n, str, dp);
}
```

Bottom - up Approach (Dynamic programming):

```
int LongestRepeatingSubsequence (string str) {
        int n = str.size();
    vector <vector <int>> dp (n+1, vector<int> (n+1, 0));

    for (int i = 1;  i <= n;  i++) {
          for (int j=1;  j <= n; j++) {

        if ( i == j || str[i-1] != str[j-1])
         dp[i][j] = max (dp[i-1][j], dp[i][j-1]);
```

```
            else
                dp[i][j] = 1 + dp[i-1][j-1];
        }
    }
    return dp[n][n];
}
```

Optimising space Complexity (code):

```
int LongestRepeatingSubsequence ( string str) {

    int n = str.size();
    vector <int> dp (n+1, 0);

    for(int i=1; i<=n; i++){
        int prev = 0;
        for(int j=1; j<=n; j++){
            int temp = dp[j];
            if( i != j && str[i-1] == str[j-1]) {
                dp[j] = 1 + prev;
            } else
                dp[j] = max(dp[j], dp[j-1]);
            prev = temp;
        }
    }
    return dp[n];
}
```

→ When we start solving question on Dynamic programming, we analyze all the approach in written form then we will able to understand and write the code.

→ After solving certain number of problem, we will analyze the approach without paper and able to write the code.

Specially for the Conversion of

↳ Recursion to Top down Approach

↳ Top down Approach to Bottom up Approach

↳ Bottom up Approach to optimise the space Complexity.

* **Minimum number of deletions and insertions :**

Given : two String str1 & str2

Task : Total No. of operation performed.

↳ For Making str2 from str1.

Example: str1 = heap str2 = pea

Approach:

Find LCS : ea

Rest Element = 2 (from str1)

Deletion = 2

For str2 → size = 3

3 - Size of LCS

3 - 2 = 1

Insertion = 1.

deletion = str1.size - LCS

insertion = str2.size - LCS

This is the same problem as LCS.
only we have to perform some operation after infinding the LCS.

Recursion (code):

```
int find(int n, int m, string &s1, string &s2) {
        if (n==0 || m==0)
            return 0;

        if (s1[n-1] == s2[m-1])
        return 1 + find(n-1, m-1, s1, s2);

      else
      return max(find(n-1, m, s1, s2),
                 find(n, m-1, s1, s2));

}
int minOperations(string str1, string str2) {

      int n = str1.size();
      int m = str2.size();

~~vector<vector<int>> dp(n+1, vector<int>(m+1, -1));~~
      int size_lcs = find(n, m, str1, str2, dp);

      int insertion = m - size_lcs;
      int deletion = n - size_lcs;
```

```
    return insertion + deletion;
}
```

Top Down Approach (Dynamic programming):

```
int find ( int n, int m, string &s1, string &s2, vector <vector <int>> &dp){
        if (n == 0 || m == 0)
            return 0;

        if (dp[n][m] != -1)
            return dp[n][m];

        if (s1[n-1] == s2[m-1])
        return dp[n][m] = 1 + find(n-1, m-1, s1, s2, dp);

        else
        return dp[n][m] = max (find(n-1, m, s1, s2, dp),
                               find(n, m-1, s1, s2, dp));
}
int minOperations ( string str1, string str2) {
        int n = str1.size();
        int m = str2.size();

    vector <vector < int>> dp (n+1, vector <int> (m+1, -1));
        int size_lcs = find (n, m, str1, str2, dp);

        int insertion = m - size_lcs;
        int deletion = n - size_lcs;

    return insertion + deletion;
}
```

* Bottom up Approach: (Dynamic programming)

```
int minOperations ( string str1, string str2) {

    int n = str1.size();
    int m = str2.size();

vector<vector<int>> dp (n+1, vector<int> (m+1, 0));
    for (int i= 1; i<=n; i++) {
        for (int j= 1; j<=m; j++) {
            if (str1[i-1] == str2[i-1])
                dp[i][j] = 1 + dp[i-1][j-1];
            else
            dp[i][j] = max (dp[i-1][j], dp[i][j-1]);
        }
    }

    int size_lcs = dp[n][m];

    int insertion = m - size_lcs;
    int deletion = n - size_lcs;

    return insertion + deletion;
}
```

* Optimise space Complexity :

```
int minOperations (string str1, string str2) {

        int n = str1.size();
        int m = str2.size();

    vector<int> dp(m+1, 0);
      int curr, prev;

    for(int i=1; i<=n; i++){
          curr = 0, prev = 0;
        for(int j=1; j<=m; j++){
            prev = curr;
              curr = dp[j];
            if (str1[i-1] == str2[j-1])
              dp[j] = 1 + prev;
            else
            dp[j] = max (dp[j], dp[j-1]);
        }
    }
    int size_lcs = dp[m];

    int insertion = m - size_lcs;
    int deletion = n - size_lcs;

    return insertion + deletion;
}
```

Lecture - 74

DP Longest Increasing Subsequence || LCS || LAP

* Longest Common Substring
  ↓
  Contigous

In this problem, sabse bada Common Substring Nikalna hai.

Subsequence Me hum beech beech Me chor kar bhi bana sakte the.

Lekin Contigous chaiye hota hai Substring Me.

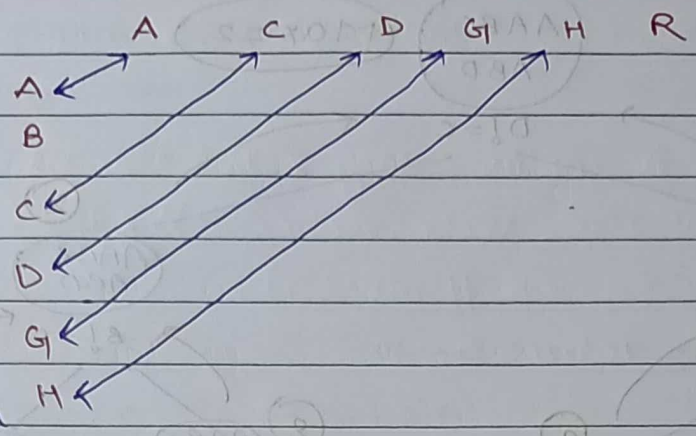
Ex: Str1 : A B C D G H

Str2 : A C D G H R

		A	C	D	G	H	R
	A	1	0	0	0	0	0
	B	0	0	0	0	0	0
	C	0	1+0	0	0	0	0
	D	0	0	1+1	0	0	0
	G	0	0	0	1+2	0	0
	H	0	0	0	0	1+3	0

↓

4




```
if ( S1[i-1] == S2[j-1])
    dp[i][j] = 1 + dp[i-1][j-1];

    ans = max (ans, dp[i][j])
```

Recursion Method :

We can start matching both string from last.

If both element does not Match, the We do 2 case.

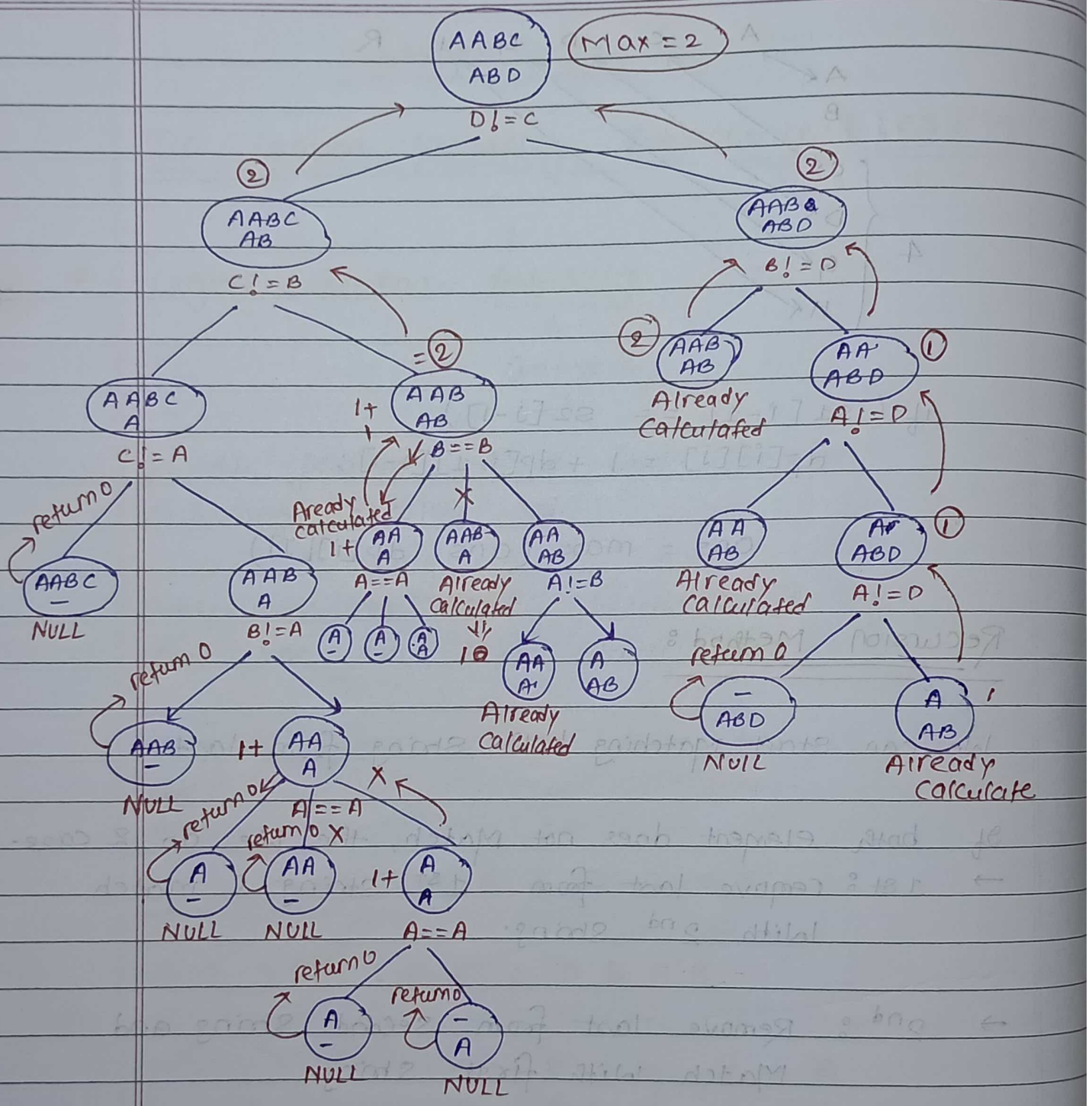
→ 1st : remove last from 1st string & match with 2nd string.

→ 2nd : Remove last from second string and Match with first string.

If both element match, then it divide into 3 parts
- ↳ remove last from both
- ↳ 2 case come from the cases when both element dont Match.

Example : Str1 : A A B C

Str2 : A B D




```
if (n == 0 || m == 0)
    return 0;

if (S1[n-1] == S2[m-1])
    len = 1 + find(n-1, m-1);

find(n-1, m)
find(n, m-1)
```

```
    ans = max (ans, len);
    return len;
```

Recursion (code):

```
int find (int n, int m, string &s1, string &s2, int &ans){
    if (n==0 || m==0)
        return 0;

    int len = 0;
    if (s1[n-1] == s2[m-1]) {
        len = 1 + find (n-1, m-1, s1, s2, ans);
        ans = max ( ans, len);
    }

    find (n-1, m, s1, s2, ans);
    find (n, m-1, s1, s2, ans);
    return len;
}
int longestCommonSubstr (string s1, string s2, int n, int m) {
        int ans = 0;
        find (n, m, s1, s2, ans);
        return ans;

}
```

Top down Approach (Dynamic programming):

```
int find (int n, int m, string &s1, string &s2,
          int &ans, vector<vector<int>> &dp) {
    if (n==0 || m==0)
        return 0;

    if (dp[n][m] != -1)
        return dp[n][m];

    int len = 0;
    if (s1[n-1] == s2[m-1]) {
        len = 1 + find (n-1, m-1, s1, s2, ans, dp);
        ans = max (ans, len);
    }

    find (n-1, m, s1, s2, ans, dp);
    find (n, m-1, s1, s2, ans, dp);

    return dp[n][m] = len;
}
int longestCommonSubstr (string s1, string s2, int n, int m){

    int ans = 0;
    vector<vector<int>> dp (n+1, vector<int>(m+1, -1));
    find (n, m, s1, s2, ans, dp);

    return ans;
}
```

Bottom up Approach :

```
int longestCommonSubstr (string s1, string s2, int n, int m) {

    int ans = 0;

vector<vector<int>> dp (n+1, vector<int>(m+1, 0));

    for (int i = 1; i<=n; i++) {
        for (int j= 1; j<=m; j++) {
            if (s1[i-1] == s2[j-1]) {
                dp[i][j] = 1 + dp[i-1][j-1];
                ans= max (ans, dp[i][j]);
            }
        }
    }
    return ans;
}
```

Example:

		A	C	D	G	H	R
		1	0	0	0	0	0
A	0						
B	0	0	0	0	0	0	0
C	0	0	0+1	0	0	0	0
D	0	0	0	1+1	0	0	0
G	0	0	0	0	2+1	0	0
H	0	0	0	0	0	3+1	0

⇓

4

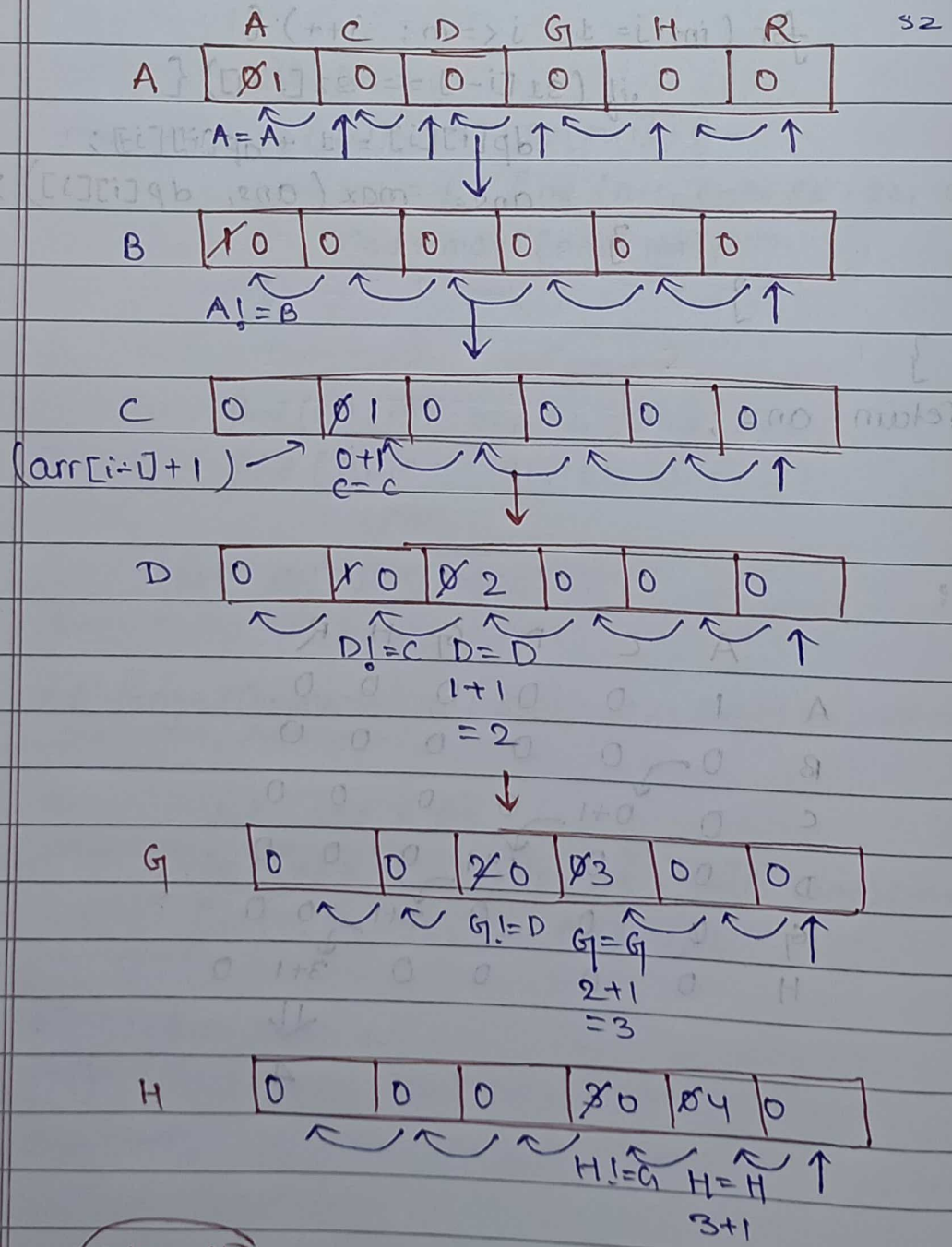
If We Want to optimise our space Complexity then,

the ans index or any index depend on only.

↳ one - D array

We can calculate all the values from only one - D Array.

We can start solving from last



Max = 4

Ans

Code (space Complexity optimused) :

```
int longestCommonSubstr (string s1, string s2, int n, int m) {

    int ans = 0;
    vector <int> dp (m+1, 0);

    for (int i=1; i<=n; i++) {
        for (int j= m; j>=1 ; j--) {
            if (S1[i-1] == s2[j-1]) {
                dp[j] = dp[j-1] + 1;
                ans = max (ans, dp[j]);
            }
            else
                dp[j] = 0;
        }
    }
    return ans;
}
```

* Longest increasing subsequence:

Given: An array of integers.

Task: length of longest increasing subsequence

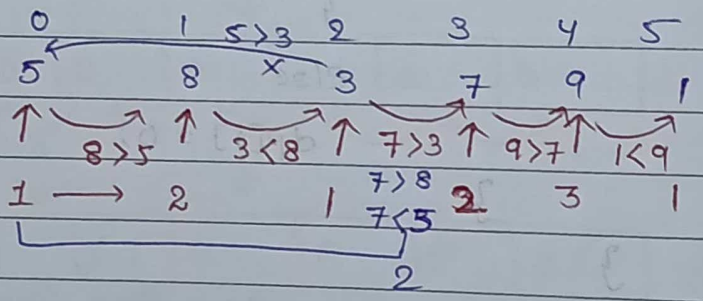
Example:

N = 6

{ 5, 8, 3, 7, 9, 1 }

0 1 2 3 4 5

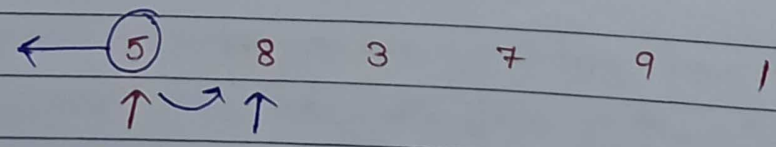
Approach:



Ans = 3

N = 6

Detailed Approach:

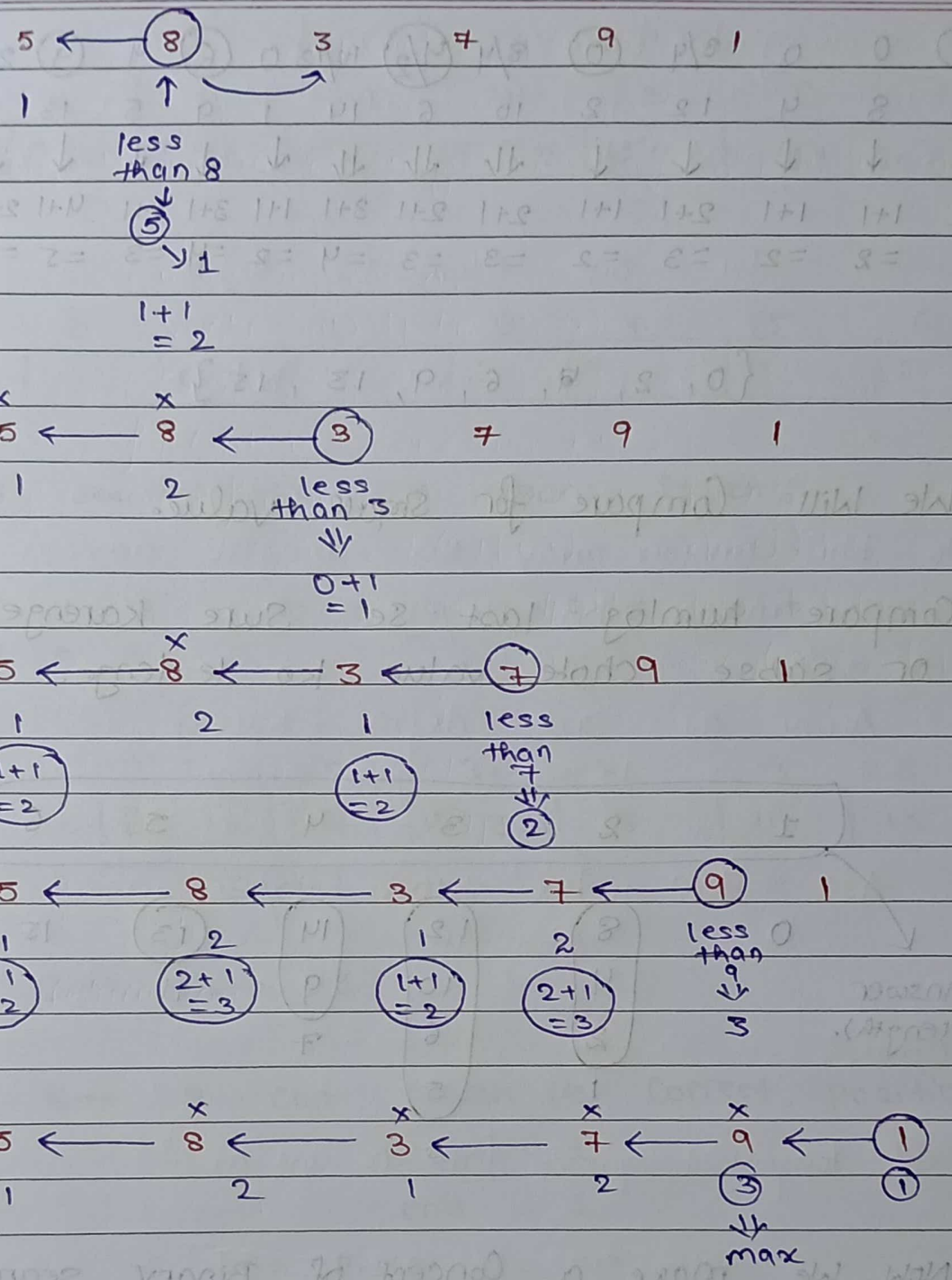


less than

5 = 0

height = 1

self



Ans = 3

Time Complexity = $O(N^2)$

space Complexity = $O(N)$

We have to convert time Complexity to $O(\log N)$

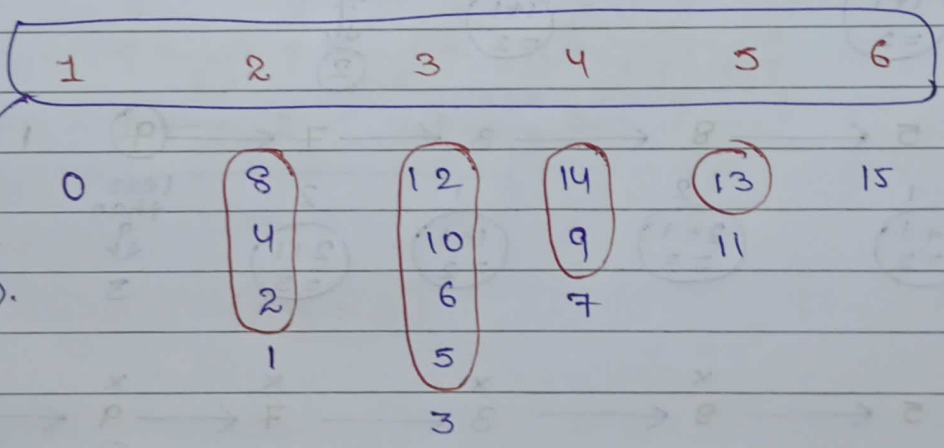
Example → 0 0 8/4 (0) 8/4 (4/2) 10/12 0 (6) 4 (9) 2 9 6 (13)

0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
↓	↓	↓	↓	↓	↓↓	↓↓	↓↓	↓	↓	↓	↓	↓	↓	↓	↓
1	1+1	1+1	2+1	1+1	2+1	2+1	3+1	1+1	3+1	2+1	4+1	2+1	4+1	3+1	5+1
	=2	=2	=3	=2	=3	=3	=4	=2	=4	=3	=5	=3	=5	=4	=6

{0, 2, 4, 6, 9, 13, 15}

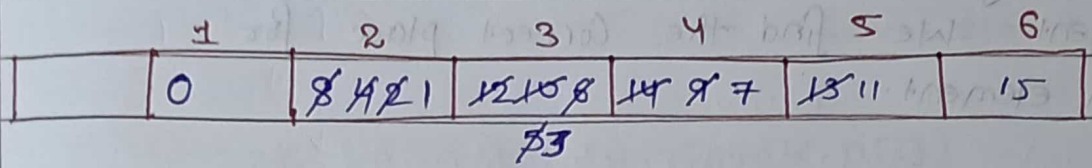
→ We Will Compare for small value.

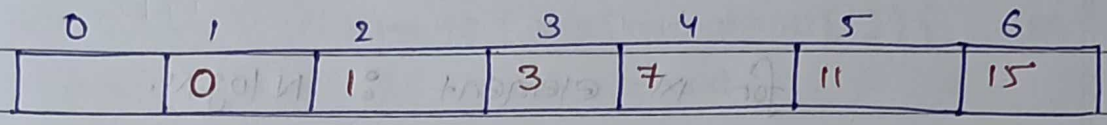
→ Compare humlog last se sure karenge or sabse chote value ko le lenge.



Now, We make a Concept of Binary search.

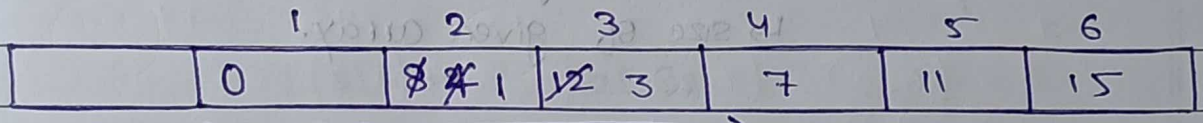
Jab humlog sabse small value ko dhundte hai to kyu nahi hum sirf ek hi value ko store kare.





Lets see how we use Binary search :

0 8 4 12 2 10 6 14 1 9 5 13 3 11 7 15



0 → fits at 1

8 → 8 can search its correct position

start = 1

end = 1

mid = 1

arr[mid] < 8

↳ 8 goes to mid+1

4 → start = 1

end = 2 (mid = 1)

arr[mid] < 4

↳ 4 goes to (mid+1, end) (2, 2)

12 → start = 1, end = 2, mid = 1

arr[mid] < 12 (mid+1, end)

mid = 2 arr[mid] < 12 → mid+1

By solving by binary search for each element We find the correct place for each element.

For one element : log N

for N element : N log N.

Jo bhi answer array ka length aayega wahi answer ho jayega.

Array ka size define karenge
↳ size of given array.

As: last index where you insert element.

Code:

```
int longestSubsequence (int n, int a[]) {

    int size = 0, start, end;
    int mid, index;

    vector<int> LIS (n);
    LIS [0] = a[0];

for (int i = 1; i<n; i++) {
    start = 0;
    end = size;
    index = size+1;
```

```
While ( start <= end )
    mid = start + (end - start) / 2;

    if (LIS[mid] < a[i])
        start = mid + 1;
    else if ( LIS[mid == a[i]) {
        index = mid;
        break;
    }
    else {
        index = mid;
        end = mid - 1;
    }
}
LIS[index] = a[i];
size = max(size, index);
}
return size + 1;
}
```

* **Longest Arithmetic progression**

Given : An Array
↳ having No Repeated Elements.

Task : Find longest AP

0	1	2	3	4	5
1	7	10	13	14	19

(difference, length)

0	1	2	3	4	5
1	7	10	13	14	19
	(6, 2) ↓ With 1	(3, 2) With 7	(3, 3) With 10	(1, 2) With 13	(5, 2) With 14
		(9, 2) With 1	(6, 3) With 7	(4, 2) With 10	(6, 4) With 13
			(12, 2) With 1	(7, 2) With 7	(9, 3) With 10
				(13, 2) With 1	(12, 2) With 7
					(18, 2) With 1

Ans = Maximum length.

Time Complexity : $O(N^3)$

If we make unordered map, then we can access it in O(1) time.

↳ then time Complexity = $O(N^2)$.

```
unordered_map <int, int> m[6];
```

But, after using unordered map it gives time limit Exceed.

Because unorded map does not take exactly O(1), it take some more than O(1).

Let we write code with unorded Map :

```
int lengthOfLongestAp (int A[], int n) {

    if (n <= 2)
        return n;

    unordered_map <int, int> m[n];
    int d;
    int ans = 2;
    for (int i = 1; i < n; i++) {
        for (int j = i-1; j >= 0; j--) {
            d = A[i] - A[j];
            if ( m[j].count(d)) {
                m[i][d] = max (m[i][d], 1 + m[j][d]);
                ans = max (ans, m[i][d]);
            }
            else {
                if ( !m[i].count(d)) {
                    m[i][d] = 2;
                }
```

```
            }
        }
    }
    return ans;
}
```

This gives time limit Exceed.

→ So we have to use Something else for the Solving in place of unordered map.

Make an array of size : (Maximum difference)

⇓

last element - first element.

2D Array

1 7 10 13 14 19

19 - 1 = 18

Make a two D array of 18 × 18.

	1	7	10	13	14	19
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	2				
7						
8						
⋮						
18						

classmate
Date
Page

Let We can understand in detail :-

	1	7	10	13	14	19
0	0	0	0	0	0	0
1	0	0	0	0	~~0~~ 2	0
2	0	0	0	0	0	0
3	0	0	~~0~~ 2	~~0~~ 3	0	0
4	0	0	0	0	~~0~~ 2	0
5	0	0	0	0	0	~~0~~ 2
6	0	~~0~~ 2	0	~~0~~ 3	0	~~0~~ 4
7	0	0	0	0	~~0~~ 2	0
8	0	0	0	0	0	0
9	0	0	~~0~~ 2	0	0	~~0~~ 3
10	0	0	0	0	0	0
11	0	0	0	0	0	0
12	0	0	0	~~0~~ 2	0	~~0~~ 3
13	0	0	0	0	~~0~~ 2	0
14	0	0	0	0	0	0
15	0	0	0	0	0	0
16	0	0	0	0	0	0
17	0	0	0	0	0	0
18	0	0	0	0	0	~~0~~ 2

Max = 4

T.C = $O(N^2)$

S.C = $O(N^2)$

Lecture 75

Grid + Stock

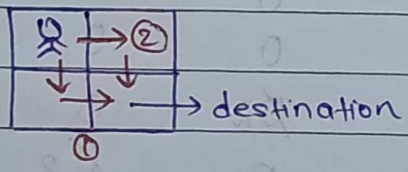
* Number of unique paths:

Grid : MxN

Ek baar me app ya to right move kar sakte ho ya down move kar sakte ho.

↳ Right

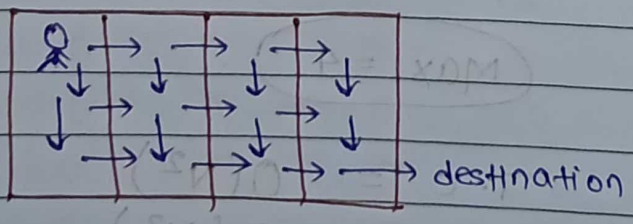
↳ Down



from source to destination, How many ways to reach?

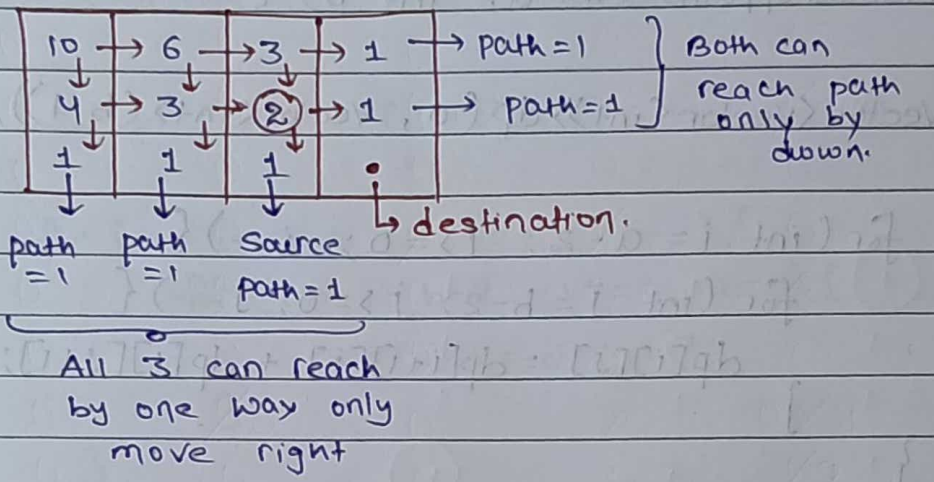
Ans = 2

Example : 2



Total : 10 ways.

* . How to approach ?



Sum of the down and right index.

→ Make the last row & last col = 1.

→ Start with blank index which is empty.

Code understanding :

Assign last row & col to 1 :

			1
			1
1	1	1	1

Add all the rest index with

```
a[i][j] = a[i+1][j] + a[i][j+1];
```

→ Answer.

10	6	3	1
4	3	2	1
1	1	1	1

Code :

```
int NumberOfpaths (int a, int b){

    Vector<vector<int>> dp(a, vector<int>(b,1));

    for(int i= a-2; i>=0; i--){
        for (int j = b-2; j>=0; j--){
            dp[i][j] = dp[i+1][j] + dp[i][j+1];
        }
    }
    return dp[0][0];
}
```

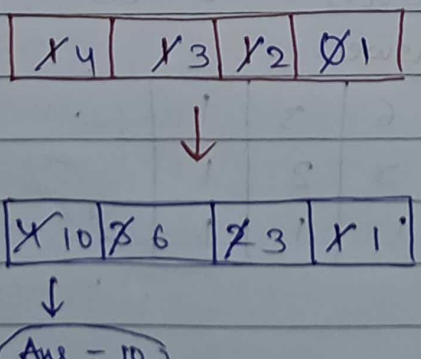
We can optimise this Code :

10	6	3	1
4	3	2	1
1	1	1	

→ This row is depend only on next row.

So, we can only solve this by 1 D array.

Approach :



optimised code :

```
int NumberOfPath (int a, int b) {

    Vector <int> dp ( b, 1);

    for (int i = a-2;  i >= 0;  i--) {
        for (int j = b-2; j >= 0; j--) {
            dp[j] += dp[j+1];
        }
    }
    return dp[0];
}
```

* Unique path in a Grid

Given : A Grid
- ↳ size : n x m
- ↳ Consists 0 and 1

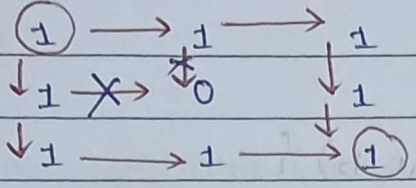
↳ 1 means you are allowed to enter.
↳ 0 means you are not allowed to enter.

Example :

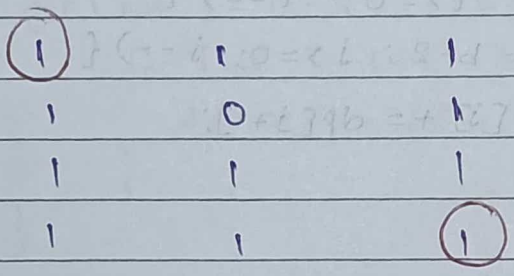
n = 3 m = 3

(1) ← start point	1	1
1	0	1
1	1	(1) → End point

Find number of possible path to reach end point from start point.



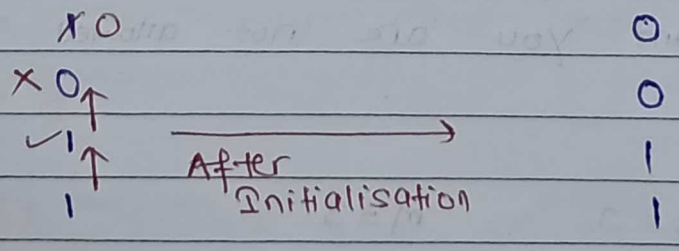
Ans = 2

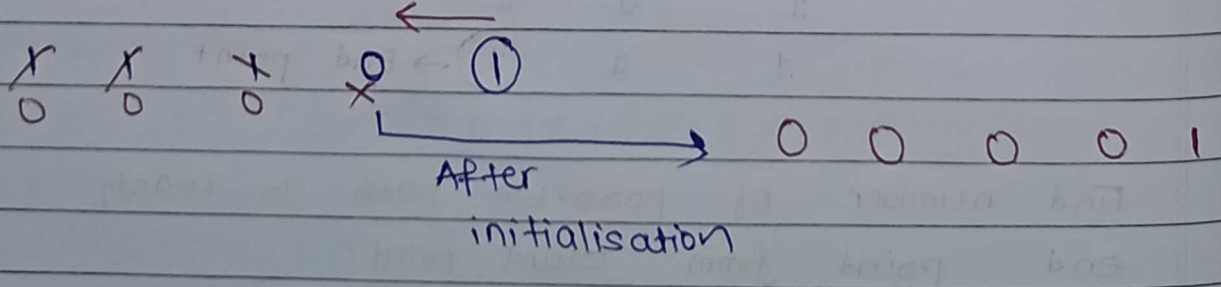


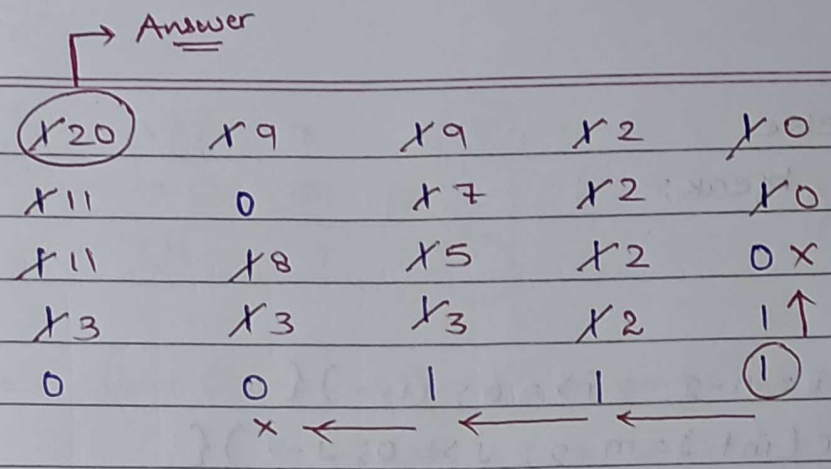
Initialise :

↳ Start from right most bottom
  ↳ check for left and up
    ↳ if any 0 found
      ↳ All upper or left
        . are initialised as 0.

Ex :







→ First we initialise last row & last colum.

→ then we check other element and perform.

arr[i][j] = arr[i][j+1] + arr[i+1][j]

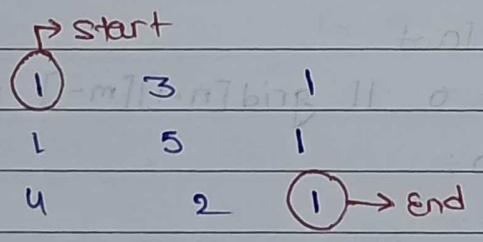
Code :

```
int uniquepaths (int n, int m, vector<vector<int>> &grid) {
    // for first and last
    if (grid[0][0] == 0 || grid[n-1][m-1] == 0)
        return 0;

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

for (int j = m-2; j >= 0; j--) {
        if (grid[n-1][j] == 1)
            dp[n-1][j] = 1;
        else
            break;
    }

for (int i = n-2; i >= 0; i--) {
        if (grid[i][m-1] == 1)
            dp[i][m-1] = 1;
```

```
            else
                break;
            }

        for (int i = n-2;  i >= 0;  i--) {
            for (int j = m-2;  j >= 0;  j--) {
                if (grid[i][j])
                dp[i][j] = (dp[i+1][j] + dp[i][j+1]) % 1000000007;
            }
        }
        return dp[0][0];
    }
```

* **Minimum Path Sum (LeetCode)**

Example:



Find the path with minimimum cost from start to end.

At each, you can go either Right or Down.

→ First update last row & colum from last

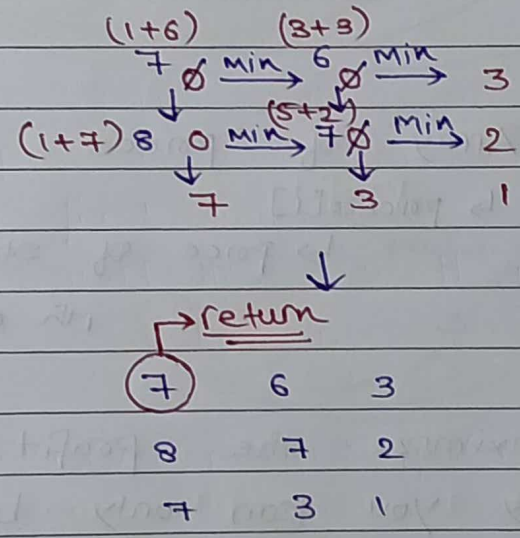
→ Then start for remaining and choose the Minm path.

1	3	1
1	5	1
4	2	1

update last row & last column

0	0	~~3~~ (2+1)
0	0	~~2~~ (1+1)
~~7~~ (4+3)	~~3~~ (1+2)	~~1~~

For remaining choose Min-m from right or down



7	6	3
8	7	2
7	3	1

Code:

```
int minpathsum ( vector<vector<int>> &grid ) {

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

for (int j= m-2;   j>=0; j--){
    grid[n-1][j] +=grid[n-1][j+1];
}
```

```
for (int i= n-2;  i >= 0;  i--){
        grid[i][m-1] += grid[i+1][m-1];
    }

for ( int i = n-2;  i >= 0;  i--) {
        for ( int j = m-2;  j >= 0;  j--){
        grid[i][j] += min( grid[i+1][j] , grid[i][j+1]);
        }
    }
return  grid[0][0];
}
```

* **Best time to Buy and Sell Stock**

Given : Array of prices
↳ prices[i]
↳ price of stock on ith day.

→ you want to maximize the profit.
one day you can only buy the stock or sell the stock.

you are allowed to skip the day.

↳ Return Maximum profit.

↳ If No profit made
↳ return 0.

Days	1	2	3	4	5	6
	7	1	5	3	6	4

profit (Condition)

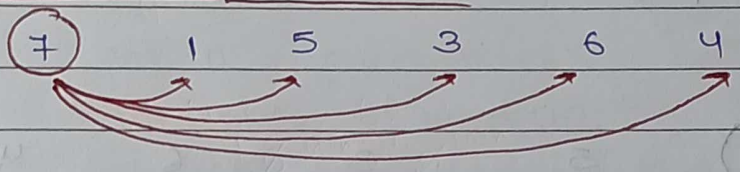
↳ Buy on less price

↳ sell on More price

→ Minimise the Buy day

↳ First We have to buy.

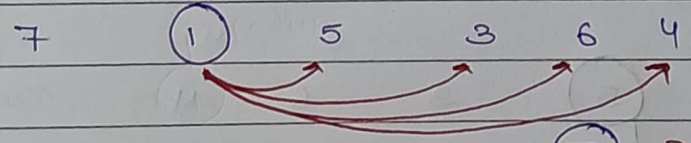
BRUTE FORCE APPROACH



profit = 0

profit -6 -2 -4 -1 -3

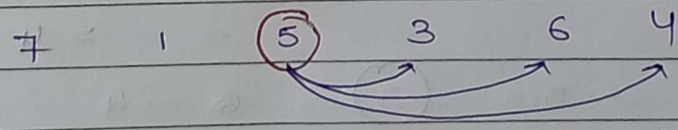
No profit



profit = 5

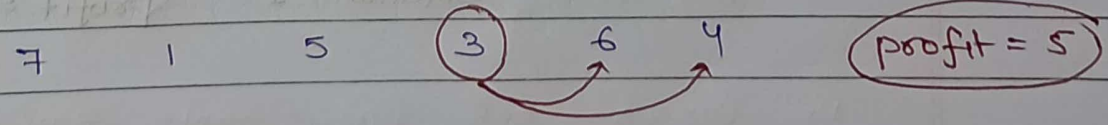
Profit : 4 2 5 3

↳ Max



profit = 5

profit 2 -1 1



profit = 5

profit 3 1

Time Complexity = N^2

Approach 2 :

7 1 5 3 6 4

Find smallest and largest and we get the Maximum profit.

largest always comes after smallest.

(7) 1 5 3 6 4

↑

smallest = 7 profit = 0

7 (1) 5 3 6 4

1 > 7

false profit = 0

smallest = 1

7 1 (5) 3 6 4

5 > 1

smallest = 1 profit = max(0, 5-1)

= 4

7 1 5 (3) 6 4

3 > 1

smallest = 1 profit = max(4, 3-1)

= 4

7 1 5 3 (6) 4

6 > 1

smallest = 1

profit = max(4, 6-1)

= 5

7 1 5 3 6 (4)

4 > 1

smallest = 1

profit = max (5, 4-1)

= 5

Time Complexity : O(N)

Code :

```
int maxprofit ( vector <int> & prices) {
        int profit = 0;
        int n = prices.size();
        int stock = prices[0];

    for (int i= 1; i<n; i++){
        profit = max ( profit , prices[i] - stock);
        stock = min (stock, prices[i]);
    }
    return profit;
}
```

* Best time to Buy and sell Stock II

An Integer Array with prices[i] of stock on ith day.

↳ Find maximum profit.

You can buy and sell the stock any number of times.

Condition :

↳ you can first buy the stock.

↳ ~~After~~ 2

↳ Before 2nd time buy the stock you have to sell the previous stock.

Example:

7 1 5 3 6 4

→ Unlimited time buy sell.

Condition :

↳ Buy → sell → Buy → sell ✓

↳ Buy → Buy → sell ✗

↳ Buy & sell on same day ✓

But profit 0.

7 1 5 3 6 4

1 → 5
5 → 6

profit = 5

1 5
3 6

profit = 7

Approach:

7 1 5 3 6 4

Buy at 7

7 1 5 3 6 4

1 < 7

cancel at 7 — Buy at 1

7 1 5 3 6 4

5 > 1

profit

sell at 5 → 5 - 1 = 4 profit

Then, Buy at 5.

7 1 5 3 6 4

3 < 5

cancel at 5 — Buy at 3

7 1 5 3 6 4

6 > 3

sell at 6 → 6 - 3 = 3

3 + 4 = 7

Buy at 6

7 1 5 3 6 4

4 < 6

cancel at 6 — Buy at 4

Total profit = 7

→ first Day मैं stock को buy कर lenge.

↳ अगर Next day stock का price ज्यादा है तो sell कर देंगे और profit को add कर देंगे।

↳ phir usi din stock ko usi price pe buy kar lenge.

↳ फिर से check करेंगे।

↳ अगर Next day stock का price कम है तो buy price का change कर के current day के price को assign कर देंगे।

* Approach 2: (More optimal)

(2) (7) 11 3 8

7 - 2

= 5 (+ve)

profit = 5

2 (7) (11) 3 8

11 - 7

= 4 (+ve)

profit = 5 + 4 = 9

2 7 (11) (3) 8

3 - 11

= -8 (-ve)

X

2 7 11 (3) (8)

8 - 3 = 5

(+ve)

profit = 9 + 5 = 14

Example 2:

(7) (1) 5 3 6 4

1 - 7

= -6 (-ve) X

7 (1) (5) 3 6 4

5 - 1

= 4 (+ve)

profit = 4

7 1 (5) (3) 6 4

3 - 5 = -2

-ve X

7 1 5 (3) (6) 4

6 - 3 = 3

+ve ✓

profit = 4 + 3 = 7

7 1 5 3 (6) (4)

4 - 6 = -2

-ve X

profit = 7

Code :

```
int maxprofit (vector <int> & prices) {

        int profit = 0;
        int n = prices.size();

    for (int i=1; i<n; i++) {
        if ( prices[i] > prices[i-1])
            profit += prices[i] - prices[i-1];
    }
    return profit;
```

Lecture 76

Stock + How to define row and column in dP

* **Subset sum problem**

Given : An array (of non negative integers)
Sum value

Task : Is there any subset exist whose sum is equal to given sum value.

Example : 3 34 4 12 5 2

Subset : Example

Subset of {1, 3} = {}, {1}, {3}, {1, 3}

$$\text{Subset} = 2^n$$

We can understand this with help of recursion tree.

Sum_Required = 9

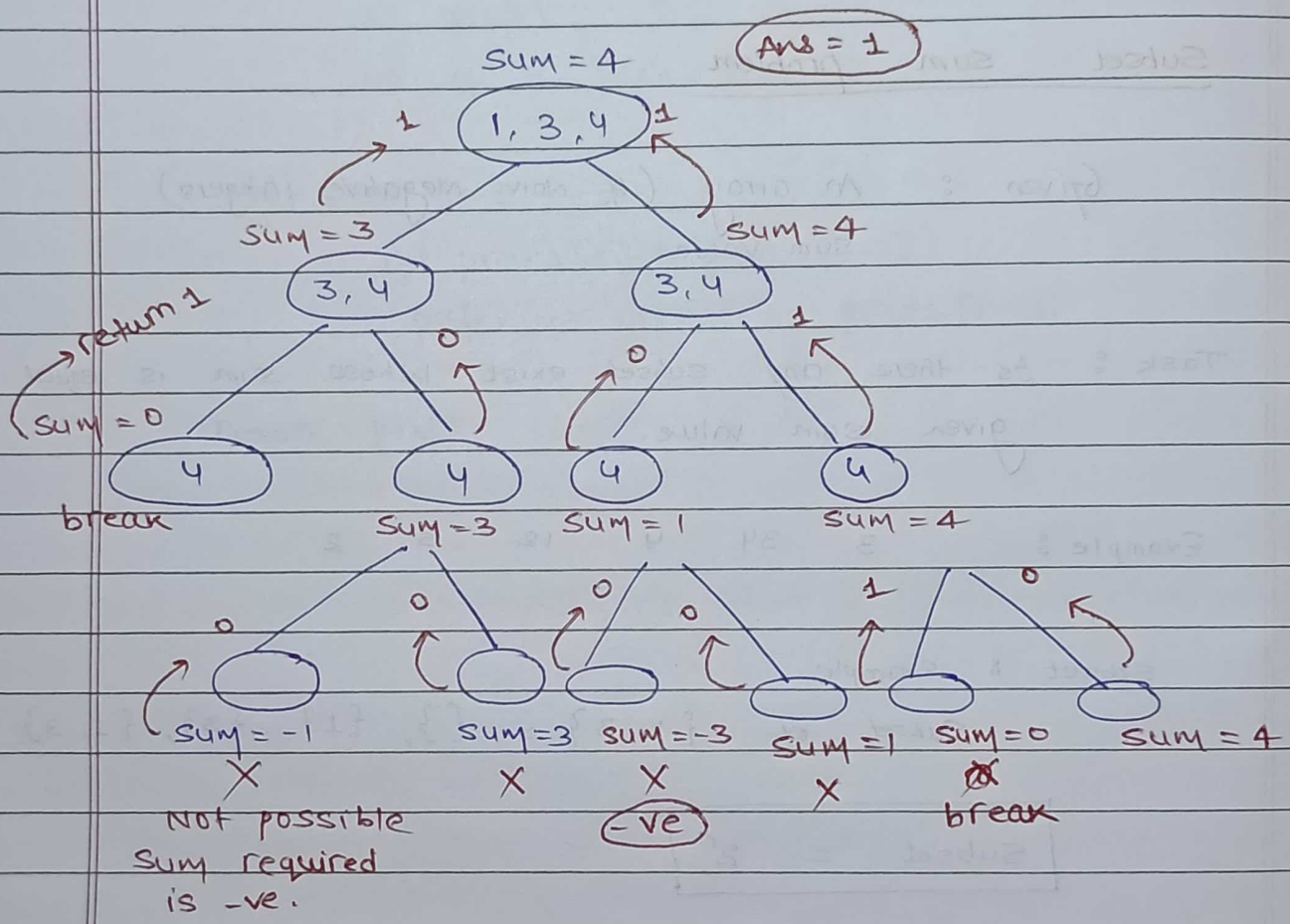
Here, we will decrease the sum_Required in each step and make sure the sum never be negative.

Check for all possible cases.

Understand with small Example:

arr = {1, 3, 4}

sum = 4



First we write the code of Recursion :

Recursion :

```
bool find (int n, int sum, vector<int> &arr) {

    if (sum == 0)
        return 1;

    if (n == 0)
        return 0;

    if (sum < arr[n-1])
        return find (n-1, sum, arr);

    else
        return find (n-1, sum - arr[n-1], arr) || find (n-1,
                                                  sum, arr);
}
bool isSubsetsum (vector<int> arr, int sum) {

        int n = arr.size();
    return find (n, sum, arr);
}
```

By dynamic programming: (Top Down Approach)

```
bool find (int n, int sum, vector <int> & arr, vector <vector<int>> & dp){
        if (sum == 0)
            return 1;

        if (n == 0)
            return 0;

        if (dp[n][sum] != -1)
            return dp[n][sum];

        if (sum < arr[n-1])
        return dp [n] [sum] = find (n-1, sum, arr, dp);

        else
        return dp[n][sum] = find (n-1, sum - arr[n-1],
                arr, dp) || find (n-1, sum, arr, dp);
}
bool isSubsetSum (vector <int> arr, int sum) {

        int n = arr.size();
    vector <vector <int>> dp (n+1, vector <int> (sum+1, -1));
    return find (n, sum, arr, dp);
}
```

Bottom up Approach:

```
if (sum == 0)
    return 1;
if (N == 0)
    return 0;
```

sum →

↓N	0	1	2	3	4	
0	1	0	0	0	0	→ N == 0 {return 0}
1	1	1	0	0	0	
2	1	1	0	1	1	
3	1	1	0	1	(1)	→ return this.

sum == 0
return 1

Example:

{1, 3, 4}

sum = 4

In this we can fill in reverse order

Col : sum → 0 1

row : N → 0 1

loop : { 1 → N
       { 1 → sum

↓

Because we fill from down.

Code: Bottom up Approach:

```
bool issubsetsum (vector <int> arr , int sum) {

        int n = arr.size();
    vector<vector<int>dp(n+1, vector<int> (sum+1, 0));

    for (int j= 0; j<= sum; j++)
            dp[0][j] = 0;

    for (int j=0;  j<= n;  j++)
            dp[j][0] = 1;

    for ( int i = 1;  i <=n;  i++) {
        for (int j=1;  j<= sum;  j++) {
            if ( j < arr[i-1]) {
                dp[i][j] = dp[i-1][j];
            }
        else{
        dp[i][j] = dp[i-1][j-arr[i-1]] || dp[i-1][j];
        }
    }
}
return dp[n][sum];
}
```

Next logic :

Recursion :

```
bool find (int index, int sum, vector<int>&arr, int n){
        if (sum == 0)
            return 1;
        if (index == N)
            return 0;
        if (sum < arr[index])
            return find (index+1, sum, arr, n);
        else
        return find (index+1, sum - arr[index], arr, n) ||
                find (index + 1, sum, arr, n);
}
    bool isSubsetSum ( vector<int>arr, int sum) {

            int n = arr.size();
        return find (0, sum, arr, n);
}
```

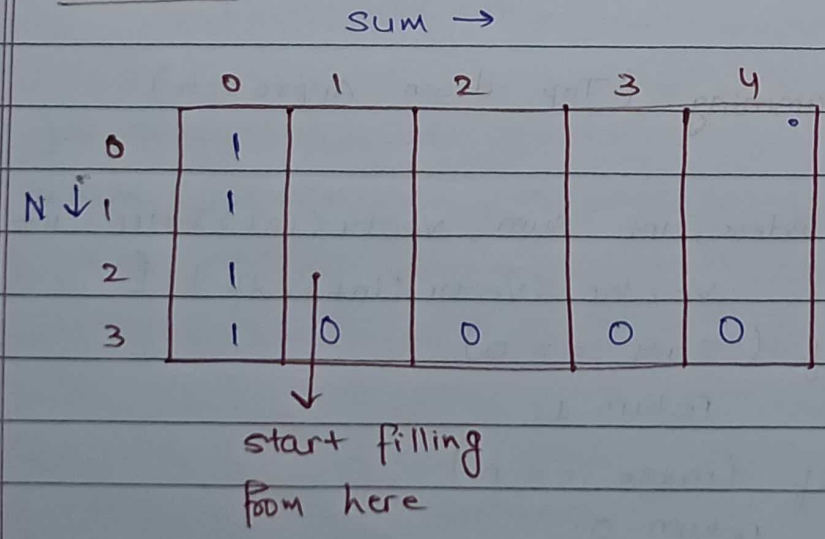
Dynamic programming (Top down Approach)

```
bool find (int index, int sum, vector<int>& arr, int n,
            vector <vector<int>> dp) {
        if (sum == 0)
            return 1;
        if (index == n)
            return 0;
        if (dp[index][sum] != -1)
            return dp[index][sum];
```

```
if ( sum < arr[index] )
return dp[index][sum] = find (index+1, sum, arr,
                              n, dp);

else
return dp[index][sum] = find (index+1, sum - arr[index],
        arr, n, dp) || find (index+1, sum, arr, n, dp);

}

bool issubsetsum (vector<int>arr, int sum) {

    int n = arr.size();
vector <vector <int>> dp (n+1, vector<int> (sum+1, -1));
return find (0, sum, arr, n, dp);

}
```

Bottom up Approach :

Example :



~~↳ solved~~

index : 0 to N

sum : sum to 0

↳ solved.

index : 0 to N-1

sum : sum to 1

⇓

reverse

index : N-1 to 0

sum : 1 to sum

Code:

```
bool isSubsetSum (vector<int>arr, int sum) {
        int n = arr.size();
    vector<vector<int>> dp (n+1, vector<int> (sum+1, 0));
     for (int j=0; j<=sum; j++)
       dp[n][j] = 0;

     for (int i=0; i<=n; i++)
        dp[i][0] = 1;

    for (int i = n-1; i>=0; i--) {
       for (int j = 1 ; j<=sum; j++) {
            if (j < arr[i])
              dp[i][j] = dp[i+1][j];
            else
            dp[i][j] = dp[i+1][j-arr[i]] || dp[i+1][j];
       }
    }

    return dp[0][sum];
}
```

* Conversion of 2D DP to 1D DP :

if (j < arr[i-1])

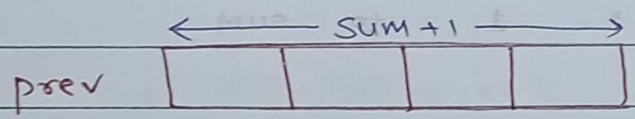
dp[i][j] = dp[i-1][j]

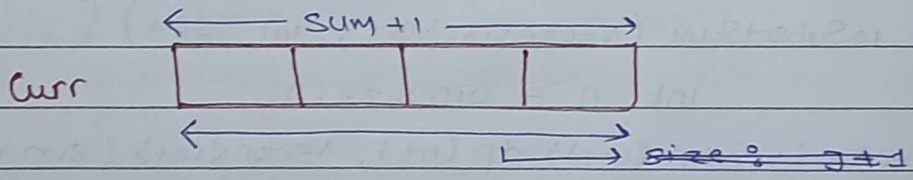
curr prev

else

dp[i][j] = dp[i-1][j-arr[i-1]] || dp[i-1][j]

curr prev prev





We use 2 row here only.

Vector <int> dp (sum + 1, 0); → only prev dp initialised.

Vector <int> curr (sum+1);

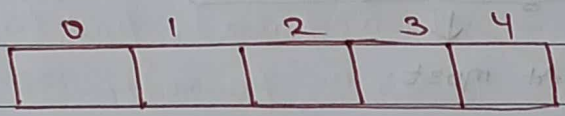
Code : (optimised)

```
bool isSubsetSum (vector<int> arr, int sum) {

    int n = arr.size();
    vector <int> prev ( sum + 1, 0);
    vector <int> curr ( sum + 1);

    prev[0] = 1;
```

```
for (int i = 1;  i<=n;  i++) {
        for (int j= 1;  j<= sum ; j++) {
            if (j< arr[i-1])
                curr[j] = prev[j];
            else
            curr[j] = prev[j- arr[i-1]] || prev[j];
        }
        prev = curr;   Swap (curr, prev);
    }
    return curr[sum];
}
```

Here we use 2 row after optimisation.

→ Now we can optimise more to 1 row only.



We will fill from last, then we can done with only 1 row.

Code: (Highly optimised)

```
bool isSubsetSum ( vector<int> arr, int sum ) {
        int n = arr.size();
    vector <int> dp (sum + 1, 0);
    dp[0] = 1;
  for (int i = 0; i<n; i++)
    for (int j = sum; j>= arr[i] ; j--) {
        dp[j] = dp[j] || dp[j - arr[i]];
    }
  return dp[sum];
}
```

Best time to Buy and Sell Stock III

Given : An Array
↳ prices

prices[i] is the prices of a given stock on the ith day.

Task : Maximum profit

We can do atmost two transaction.

Example :

{ 3 3 5 0 0 3 1 4 }

Buy → sell → Buy → sell

↓

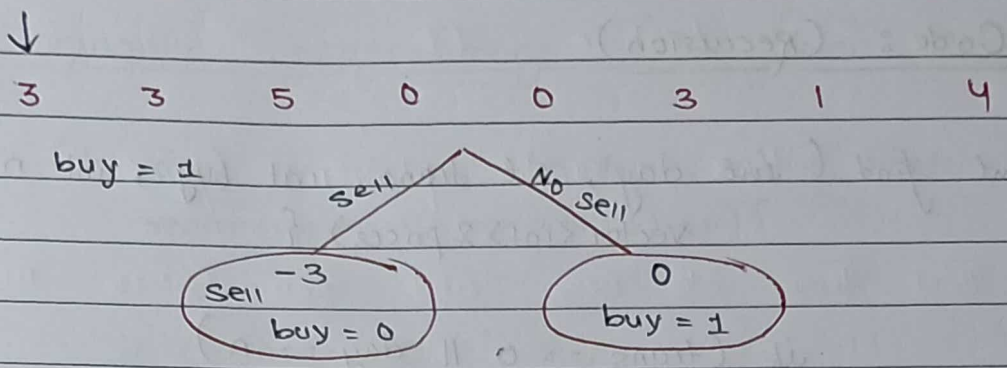
At most.

2 transaction

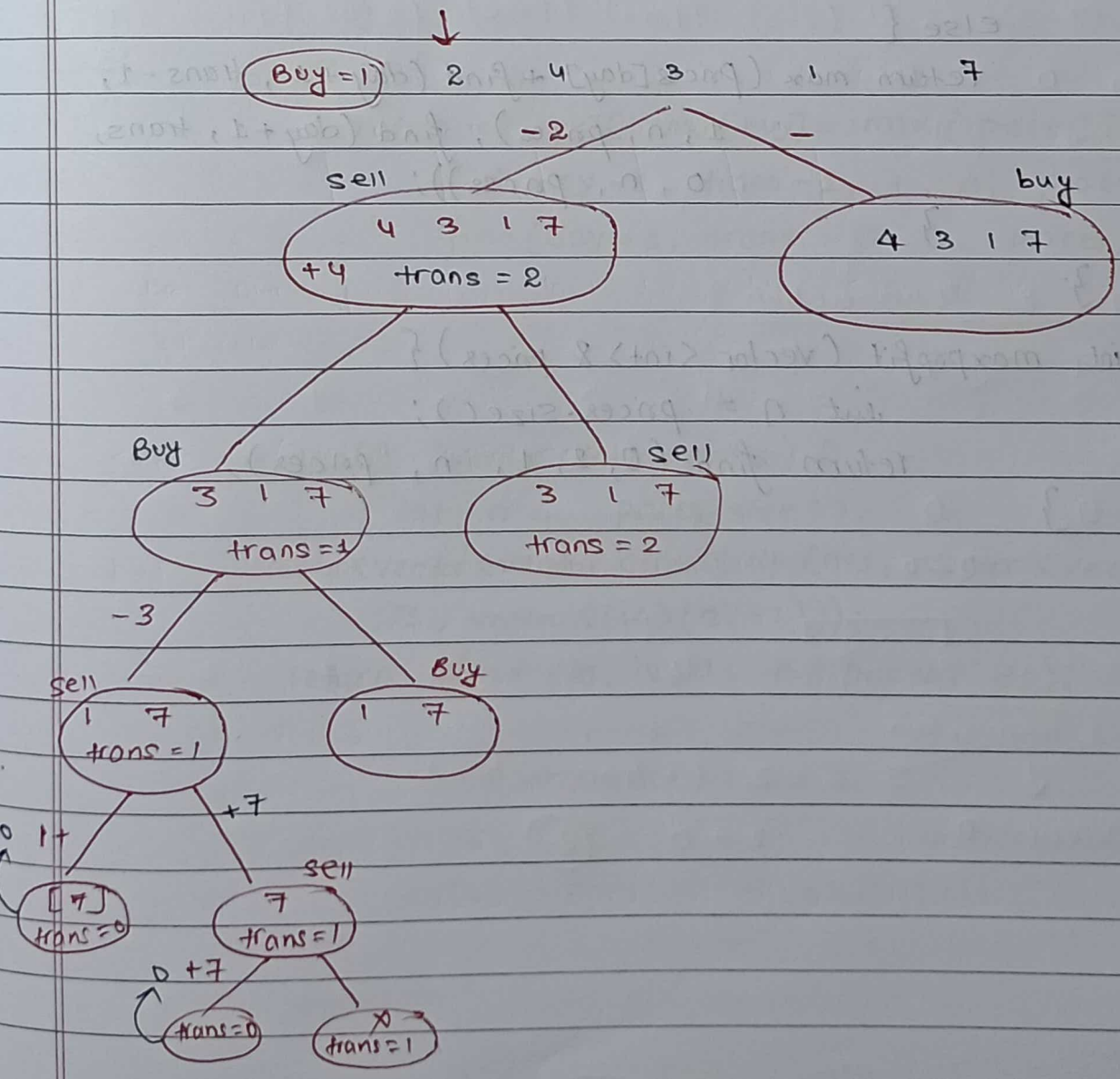
↓

{2 time}

buy = 1 → खरीदना है।
buy = 0 → बेचना है।




```
if (buy == 1)
max (-prices[i] + find(i+1, 0, Transaction),
                 find(i+1, 1, Transaction));
else
    (prices[i] + find(i+1, 1, Transaction-1),
                 find(i+1, 0, Transaction))
```



Code : (Recursion)

```
int find ( int day, int trans, int buy, int n,
            vector<int>& prices) {

    if (trans == 0 || day == n)
        return 0;

    if ( buy) {
    return max(-prices[day] + find(day+1, trans, 0,
            n, prices), find(day+1, trans, 1, n,
            prices));
    }
    else {
    return max(prices[day] + find(day+1, trans-1,
            1, n, prices), find(day+1, trans,
            0, n, prices));
    }
}

int maxprofit (vector<int>& prices){
        int n = prices.size();
        return find(0, 2, 1, n, prices);
}
```

Code : Dynamic programming (Top down Approach)

```
int find (int day, int trans, int buy, int n, vector<int>& prices,
          vector<vector<vector<int>>>& dp) {

    if (trans == 0 || day == n)
        return 0;
    if (dp[day][trans][buy] != -1)
        return dp[day][trans][buy];
    if (buy) {
        return dp[day][trans][buy] = max(-prices[day] +
            find(day+1, trans, 0, n, prices, dp), find(
                day+1, trans, 1, n, prices, dp));
    }
    else {
        return dp[day][trans][buy] = max(prices[day] +
            find(day+1, trans-1, 1, n, prices, dp),
            find(day+1, trans, 0, n, prices, dp));
    }
}

int maxprofit (vector<int>& prices) {
        int n = prices.size();
    vector<vector<vector<int>>> dp(n+1, vector<vector<int>>(
        3, vector<int>(2, -1)));
    return find(0, 2, 1, n, prices, dp);
}
```

Lecture - 77

Stock and player Game

* Best time to Buy and sell stock III

3 3 5 0 0 3 1 4

→ only 2 transaction possible

Example :

3	3	5	0	0	3	1	4
(3-3)	(3-3)	(5-3)	(prev)	(prev)	(3-0)	(prev)	(4-0)
profit on 1 transaction 0	0	2	2	2	3	3	4

profit on 2 transaction :

3 3 5 0 0 3 1 4

→ 0 0

0+0

3 3 5 0 0 3 1 4

1 transaction

↓ 5 - 3 = 2 (1 transaction)

Value = 0 2 + 0 = (2)

3 3 5 0 0 3 1 4

2 transaction

(0-5) = -ve
(No transaction)

profit = 2

Total profit = 2

3 3 5 0 0 3 1 4

1 transaction

0 - 0 = 0 (1 transaction)

profit = 2

Total profit = 2+0 = 2

0 0 -2 3
3 3 5 0 0 3 1 4
(3-0)

1 transaction

3 - 0 = 3 (1 transaction)

profit = 2

Total profit = 3+2 = 5

-2 -2 -4 1 1 -2
3 3 5 0 0 3 1 4
Max

1 transaction

1 - 0 = 1 (1 transaction

profit = 3

Total profit = 4

1 1 -1 Ans = 5 4 1 3
4
3 3 5 0 0 3 1 4
Max

1 transaction

4 - 0 = 4 (1 transaction)

profit = 2

Total profit = 6

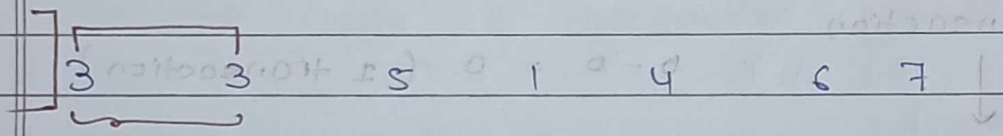
↳ Ans

Example 2:

3 3 5 1 4 6 7

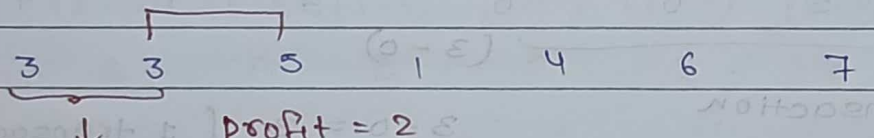
1 trans: 0 0 2 2 3 5 6

For 2nd transaction:



0 profit = 0

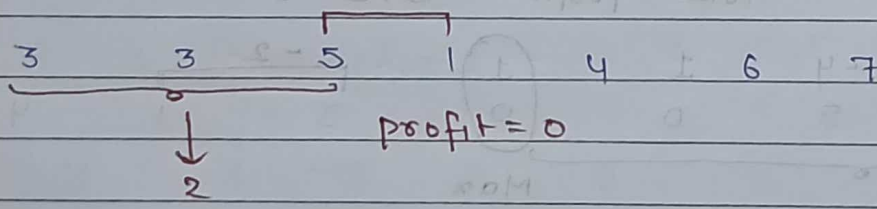
Total profit = 0 + 0 = 0



profit = 2

0

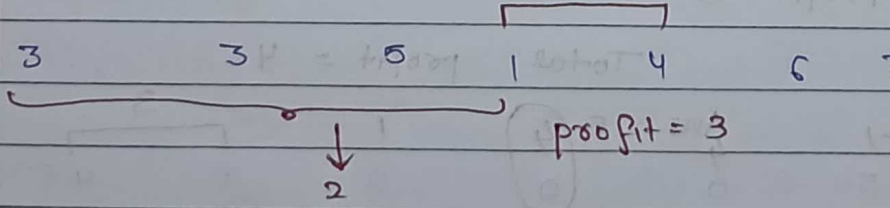
Total profit = 2



profit = 0

2

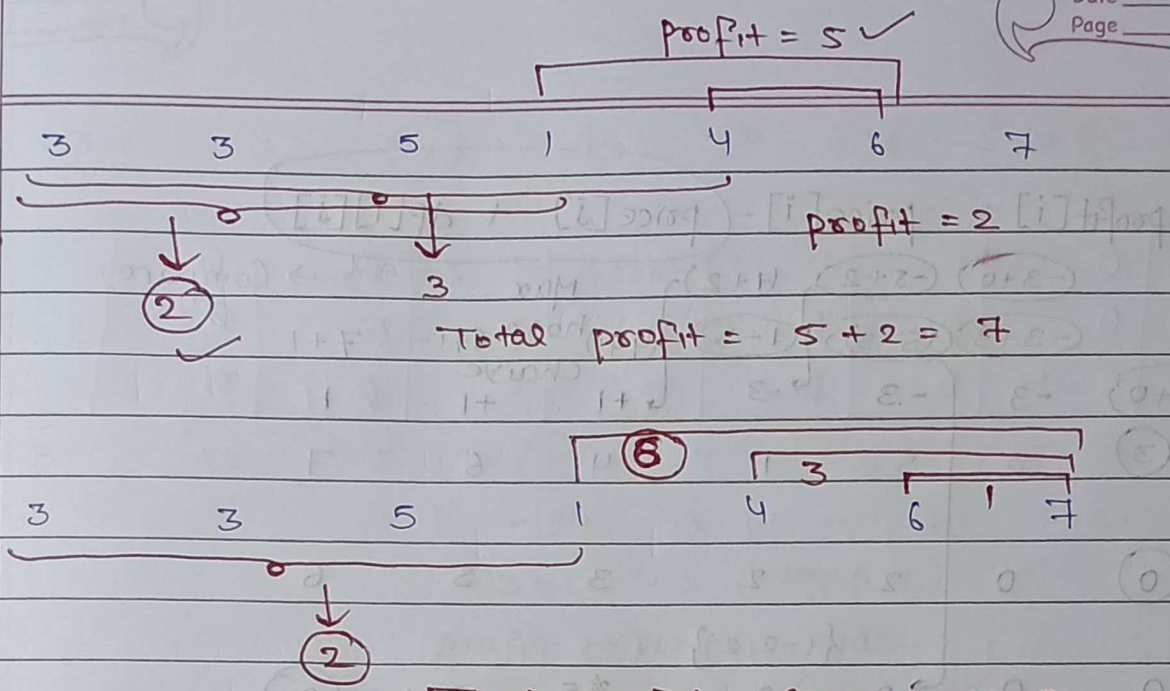
Total profit = 2



profit = 3

2

Total profit = 5



{ Maximum profit = 8 }

Time Complexity : $N^2 \times 2$

$\Downarrow\Downarrow$

$2N$

↳ How to achieve this.

For the new transaction, we compare with all the previous day. We have to minimise that comparison.

Humlog phir se minimum choose karte jayenge 2 transaction ke liye.

				j		i↓		
	3		3	5	1	4	6	7
1	0		0	2	2	3	5	6
2	0		0					

profit[i] = price[i] - (price[j] + dp[i][j]) → Compare.

(-3+0) (-5+2) (4+2) Max hona chaiye

(-3+3) (5-3) (1-3) (4+1) 6+1 7+1

(-3+0)	-3	-3	-3	+1	+1	1	
(3)	3	5	1	(8) 4	6	7	
1 trans	(0)	0	2	2	3	5	6
			{(-2,2)}				
2 trans	0 ↑	0	2 → 2		5	7	(8) → Ans.

Because No profit at Day 1.

We can understand this in detail.

	3	3	5	1	4	6	7
1 trans:	0	0	2	2	3	5	6
2 trans:							
	(3)	3	5	1	4	6	7

-3+0
= -3

profit = 0

3	(3)	5	1	4	6	7

-3

-3 + 3 = 0 (profit)

-3 + 0 = -3 (आगे जाएगा)

3 3 (5) 1 4 6 7

-3

-3+5 = 2 (profit)

-5+2 = -3 (आगे jayega)

3 3 5 (1) 4 6 7

-3

-3+1 = -2 (Negative)

profit → previous Wala

profit = 2

-1+2 = 1 (proceed)

3 3 5 1 (4) 6 7

1

1+4 = 5 (profit)

-4+5 = 1 (आगे जाएगा)

3 3 5 1 4 (6) 7

1

1+6 = 7 (profit)

-6+7 = 1 (आगे जाएगा)

3 3 5 1 4 6 (7)

1

7+1 = 8 (profit)

-7+8 = 1

{ Maximum profit = 8 }

	3	3	5	1	4	6	7
0	0	0	0	0	0	0	0
1	0	0	~~0~~ 2	~~0~~ 2	~~0~~ 3	~~0~~ 5	~~0~~ 6
2	0	0	~~0~~ 2	~~0~~ 2	~~0~~ 5	~~0~~ 7	~~0~~ 8

↓ Answer.

Code: (Bottom up Approach)

```
int maxprofit (vector<int>& prices) {

    int n = prices.size();
    vector < vector <int>> dp (3, vector <int> (n+1, 0));

    int total;
    for (int i = 1; i <= 2; i++) {
        total = - prices[0];

    for (int j = 2; j <= n; j++) {
        dp[i][j] = max (dp[i][j-1], prices[j-1] + total);
        total = max (total, - prices[j-1] + dp[i-1][j]);
    }
  }
  return dp[2][n];
}
```

* Best time to Buy and sell stock IV

In this problem, we have to find Maximum profit after doing K transaction.

K is given in question.

Code:

```
int maxProfit ( int K, vector<int>& prices) {
        int n = prices.size();
    Vector<vector<int>> dp(K+1, vector<int>(n+1, 0));

      int total = 0;

    for ( int i = 1;  i <= K;  i++) {

            total = -prices[0];
      for(int j = 2;  j <= n;  j++) {
        dp[i][j] = max (dp[i][j-1], prices[j-1] + total);
        total = max (total, -prices[j-1] + dp[i-1][j]);
  }
}
  return dp[K][n];
}
```

* player With Max Score :

Given : An array (Non-Negative)
↳ Size : N

2 players → plays Game.

player 1 start the Game.
↳ can choose one element from end and the element will be added to the scroe. Then player 2 can choose.

Task : check that player 1 will win or not.

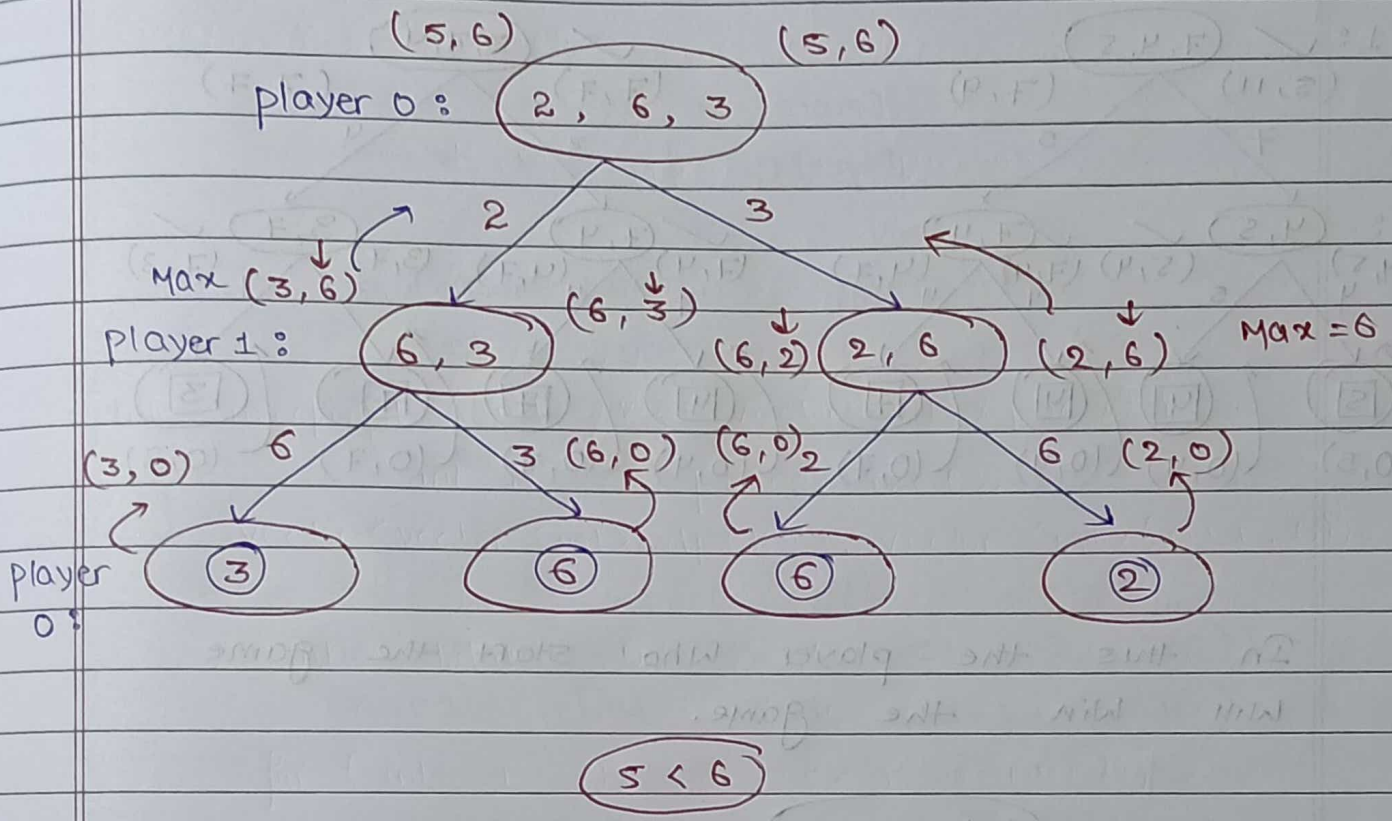
Example:

2 ~~4~~ ~~3~~ ~~1~~ ~~5~~

player 1 (start)	player 2
(1) Max (2, 5) = 5	(2) Max (2, 4) = 4
(3) Max (2, 3) = 3	(4) Max (2, 6) = 6
(5) Max (2) = 2	
Total = 10	Total = 10

Match Tie
return 0.

But the approach is wrong. We can understand this by recursion tree.



5 < 6

There is No possibility to win player 1 if both play optimally.

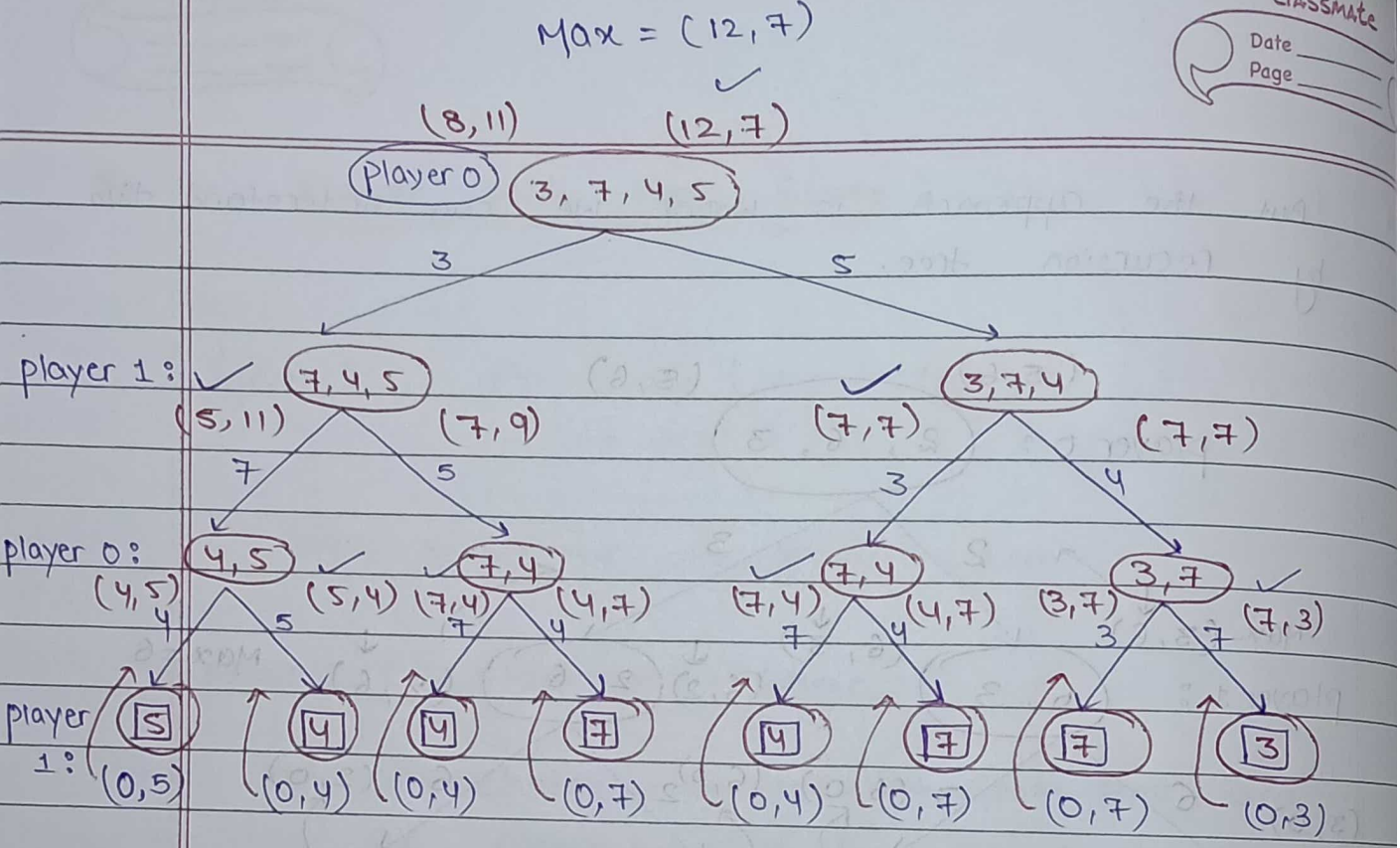
Example 2 :

{ 3 7 4 5 }

We will understand this problem with this Example.

player 0 & player 1 are 2 player

player 0 can start the game.



In this the player who start the game will win the game.

return 1

Recursion code:

```
pair<int, int> find(int start, int end, int player,
                    int arr[]){
    if (start == end){
        if (player == 0)
            return {arr[start], 0};
        else
            return {0, arr[start]};
    }

pair<int, int> score1, score2;
```

```
    if (player == 0) {
        score1 = find (start + 1, end, 1, arr);
        score2 = find (start, end - 1, 1, arr);

        score1.first += arr[start];
        score2.first += arr[end];

        if (score1.first > score2.first)
            return score1;
        else
            return score2;
    }
    else {
        score1 = find (start + 1, end, 0, arr);
        score2 = find (start, end - 1, 0, arr);

        score1.second += arr[start];
        score2.second += arr[end];

        if (score1.second > score2.second)
            return score1;
        else
            return score2;
    }
}
bool is1Winner (int N, int arr[]) {
    pair <int, int> score;
    score = find (0, N-1, 0, arr);
    return score.first > score.second;
}
```

Dynamic programming : (Top down Approach)

```
pair<int, int> find (int start, int end, int player,
        int arr[], vector<vector<vector<pair<int, int>>>> dp){

    if (start == end){
        if (player == 0)
            return {arr[start], 0};
        else
            return {0, arr[end]};
    }

    if (dp[start][end][player].first != -1)
        return dp[start][end][player];

    pair<int, int> score1, score2;

    if (player == 0)
        score1 = find (start+1, end, 1, arr, dp);
        score2 = find (start, end-1, 1, arr, dp);

        score1.first += arr[start];
        score2.first += arr[end];

        if (score1.first > score2.first)
            return dp[start][end][player] = score1;
        else
            return dp[start][end][player] = score2;
    }
    else {
        score1 = find (start+1, end, 0, arr, dp);
        score2 = find (start, end-1, 0, arr, dp);
```

```
            score1.second += arr[start];
            score2.second += arr[end];

        if (score1.second > score2.second)
            return dp[start][end][player] = score1;
        else
            return dp[start][end][player] = score2;
    }
}
bool is1Winner (int N, int arr[]) {

    pair<int, int> score;
vector<vector<vector<pair<int, int>>>> dp(N, vector<vector<pair<int,
    int>>>(N, vector<pair<int, int>>(2, {-1, -1})));

    score = find(0, N-1, 0, arr, dp);

    return score.first > score.second;
}
```

Lecture 78

Dp Advance Question

* Egg Dropping puzzle :

Given : → you have N eggs
→ K - Floored building from 1 to K.

There exist a floor f where $0 <= f <= k$ such that any egg dropped a floor higher than f will break.

Any egg dropped at or below floor f will not break.

Rules :

→ An egg that survives a fall can be used again.
→ A broken egg must be discarded.
→ The effect of a fall is same for all egg.
→ If egg doesn't break at a certain floor, it will not break at any floor below.
→ If the eggs break at certain floor, it will break at any floor above.

Return minimum number of moves that you need to determine with certainty what the value of f is.

Example :

N = Egg, K = floor

N = 2 , K = 10

→ Agr Egg tut gya to dubara use nahi hoga.

Nahi

→ Agar Egg ^ tuta to same egg ko hm phir se use kar sakte hai.

Case : N = 1 , K = 10

→ Ek-Ek kar sare floor pe check karenge jab tak anda na fut jaye ya last take na pahunch jaye.

```
if (N == 1)
    return K;
```

case : N = 2, K = 10

1 : Nahi tuta

0 : tut gya

अगर Anda tut गया तो :

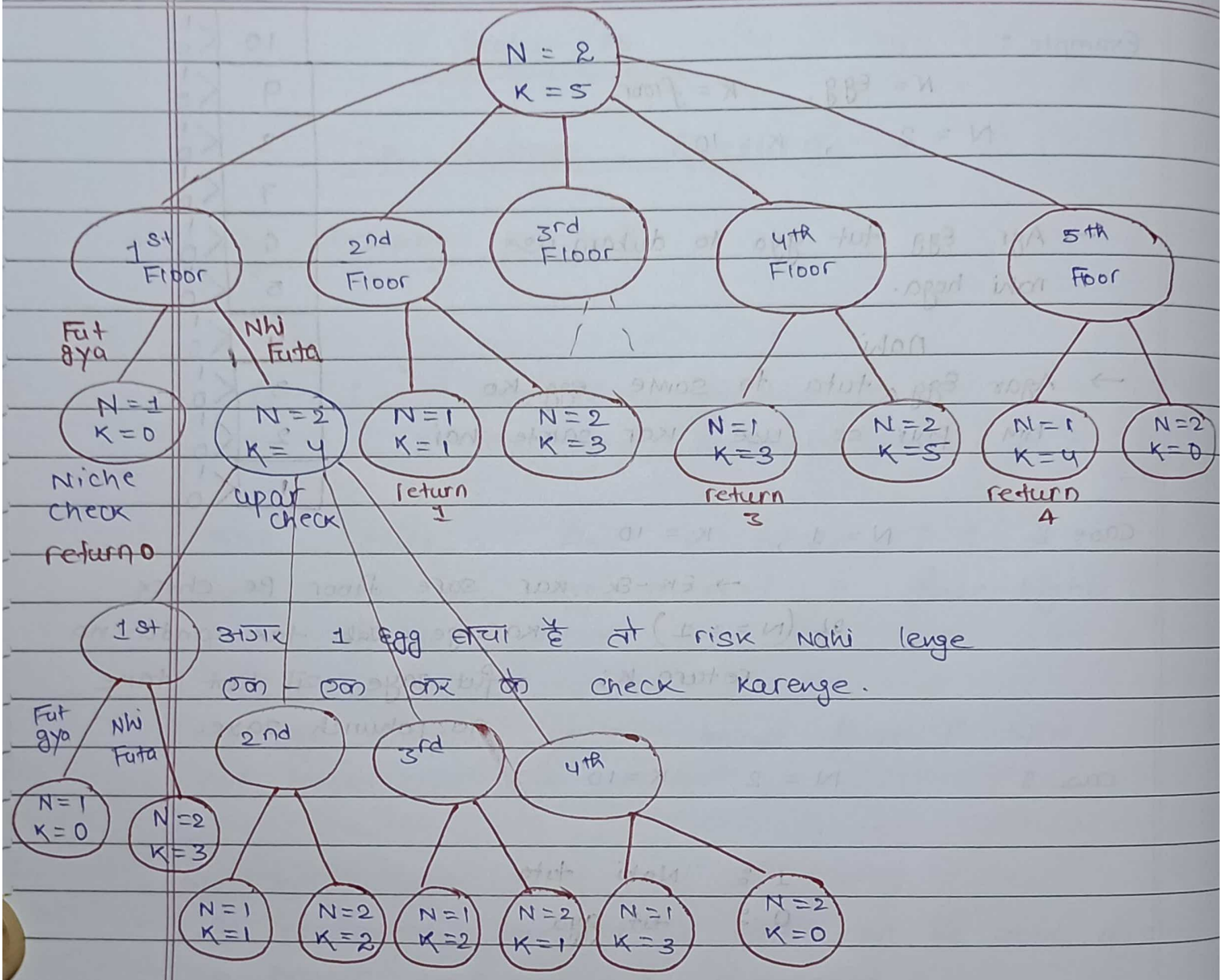
1 + find (N-1, K-1)

अगर Anda Nahi tuta तो :

1 + find (N, above floor)

2 Egg, (1-5) buildings

N = 2, K = 5



अगर 1 egg बचा है तो risk Nahi lenge एक - एक कर के check karenge.

```
if (N == 1)
    return K;

if (K == 0)
    return 0;
```

→ अगर tut gya तो नीचे जा कर check karo

→ अगर नहीं tuta तो upar जा कर check करो।

Recursion (code) :

```
int find (int eggs, int floors){~~vector<vector<int>> dp~~
    if (eggs == 1)
        return floors;

    if (floors == 0)
        return 0;

    ~~if (dp[eggs][floors] != -1)~~
    ~~return dp[eggs][floors];~~

    int ans = INT_MAX;
    int temp;
    for(int i = 1; i <= floors; i++) {
        temp = 1 + max (find (eggs-1, i-1, ~~dp~~), find (eggs,
                    floors - i, ~~dp~~));
        ans = min (ans, temp);
    }
    return ans;
}

int eggDrop (int n, int k) {

    return find (n, k);
}
```

Top Down Approach (Dynamic programming):

```
int find (int eggs, int floors, vector <vector <int>> & dp) {

    if (eggs == 1)
        return floors;

    if (floors == 0)
        return 0;

    if ( dp[eggs][floors] != -1)
        return dp[eggs][floors];

    int ans = INT_MAX;
    int temp;

    for (int i = 1; i <= floors; i++) {
        temp = 1 + max (find (eggs - 1, i-1, dp),
                        find (eggs, floors - i, dp));
        ans = min (ans, temp);
    }
    return dp[eggs][floors] = ans;
}
int eggDrop (int n, int k) {

    vector <vector <int>> dp (n+1, vector <int> (k+1, -1));
    return find ( n, k, dp);
}
```

Bottom up Approach (Dynamic programming):

```
int eggDrop (int n, int k) {

    vector<vector<int>> dp (n+1, vector<int> (k+1, 0));

    for (int j = 0; j<=k; j++) {
        dp[1][j] = j;
    }

    for (int i = 2; i<=n; i++) {
        for (int j =1; j<=k; j++) {

            int ans = INT_MAX;
            int temp;
            for (int a = 1; a<=j; a++) {
                temp = 1 + max (dp[i-1][a-1], dp[i][j-a]);
                ans = min (ans, temp);
            }
            dp[i][j] = ans;
        }
    }
    return dp[n][k];
}
```

Longest Increasing Path in a Matrix

Matrix with n rows and m columns.

Find the length of longest increasing path in Matrix

Increasing path means the value in specified path increases.

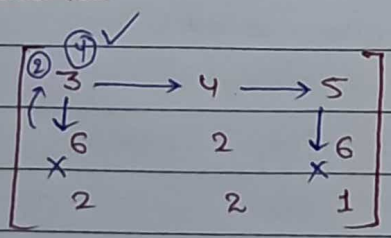
Example:

if a path of length k has value $a_1, a_2, a_3 \dots a_k$, then for every i from {2-K} must hold $a_i > a_{i-1}$.

From each cell, you can move in four direction.

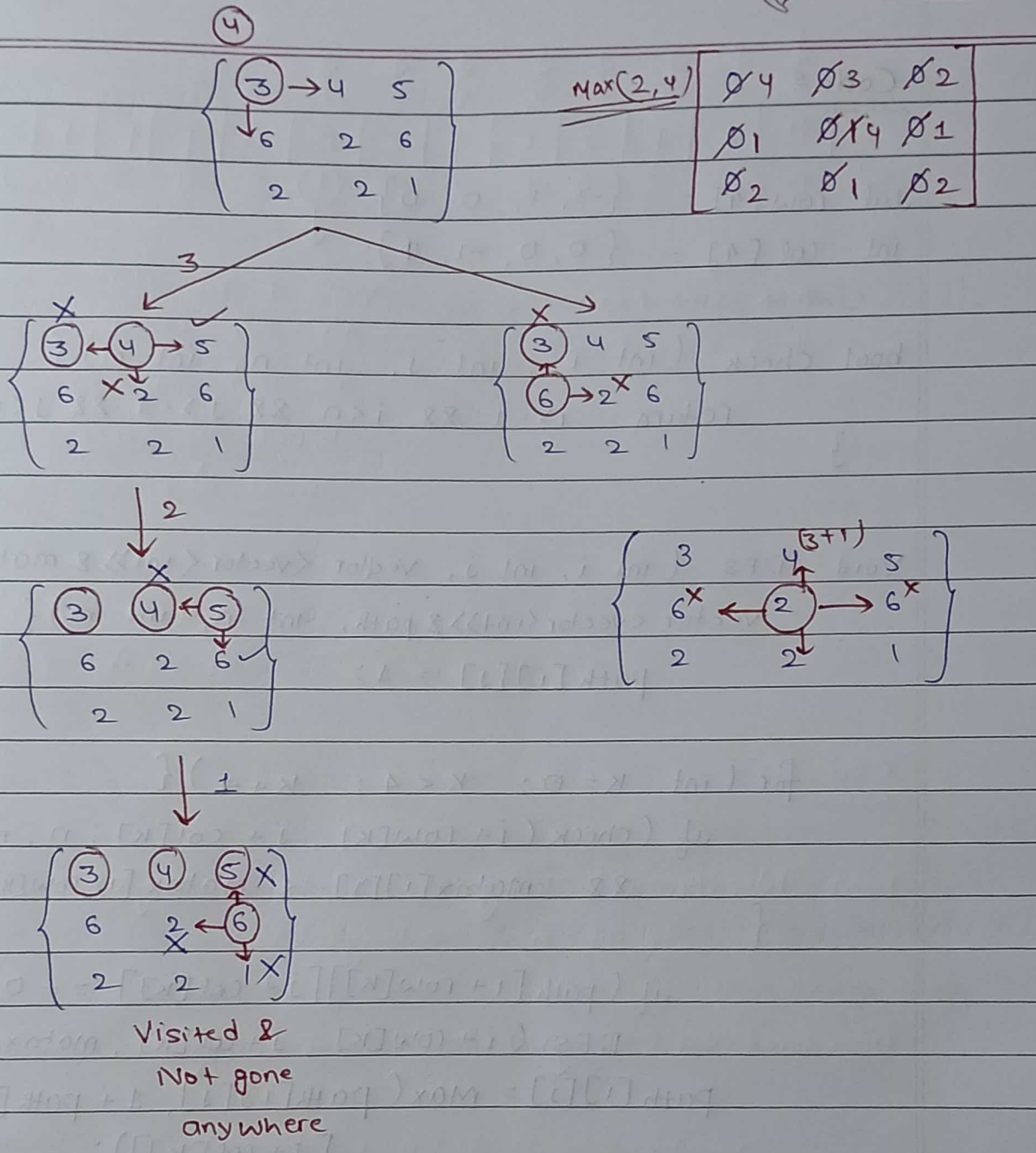
Not allowed to move diagonally or move outside the Boundary.

Example :-



path

0	0	0
0	0	0
0	0	0



We can make an extra array for storing the value of longest path in Matrix.

We can choose the answer which is maximum from all the array. (matrix).

Code:

```
int row[4] = {-1, 1, 0, 0};
int col[4] = {0, 0, -1, 0};

bool check (int i, int j, int n, int m) {
        return i>-1 && i<n && j>-1 && j<m;
    }

Void DFS (int i, int j, vector <vector<int>> & matrix,
      vector <vector<int>> & path, int n, int m) {
            path[i][j] = 1;

    for (int k=0; k<4; k++) {
        if (check (i+row[k], j+col[k], n, m)
            && matrix[i][j] < matrix[i+row[k]][j+col[k]]
        {
            if (path[i+row[k]][j+col[k]] == 0)
                DFS (i+row[k], j+col[k], matrix, path, n, m)
            path[i][j] = max(path[i][j], 1+path[i+row[k]]
                                [j+col[k]]);
        }
    }
}

nt longestIncreasingPath (vector<vector<int>> & matrix,
                          int n, int m) {

    Vector <vector<int>> path (n, vector<int>(m, 0));
        int total = 0;
```

```
for(int i = 0; i<n; i++){
    for(int j=0; j<m; j++){
        if(path[i][j] == 0)
        DFS(i, j, matrix, path, n, m);
        total = max(total, path[i][j]);
    }
}
return total;
}
```

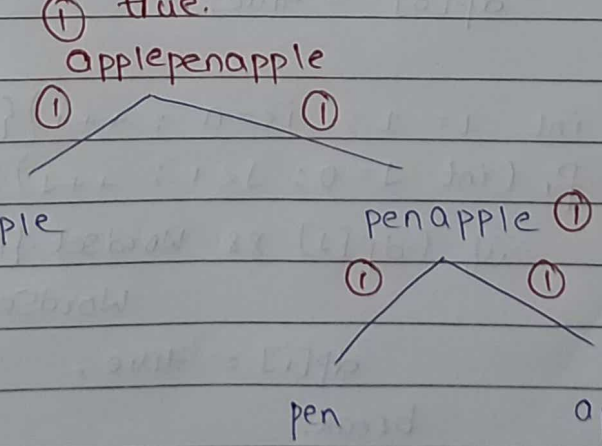
* **Word Break:** (leetcode)

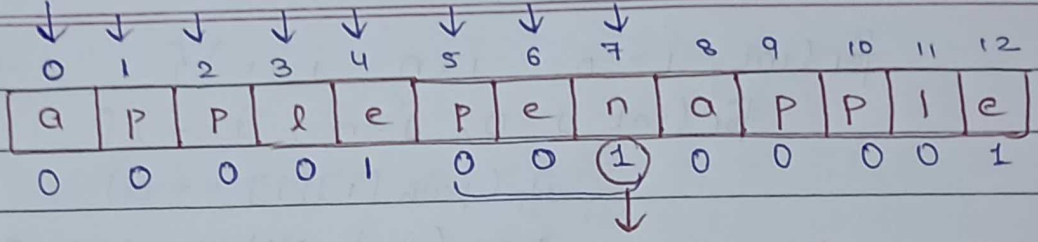
Given: → A string s
→ Dictionary of strings WordDict

Task: return true if s can be segmented into a space separated sequence of one or more dictionary word.

Example:

dict: { apple, pen }





0 → Exist (No)
1 → Exist (yes)

Yaha 1 tab hoga jab us Word ke pahle wala Word bhi Exist karta ho।

pen → (5-7)
7 pe 1 tab ho jab 4 pe 1 ho।

Code:

```
bool WordBreak ( string s, Vector <string> & WordDict ) {
        int n = s.length ();

unordered_set <string> Wordset ( WordDict.begin(), WordDict.end());

    vector <bool> dp ( n+1, false );

        dp[0] = true;

for ( int i = 1 ; i <= n ; ++i ) {
    for (int j = 0 ; j < i ; ++j ) {
        if (dp[j] && Wordset.find ( s.substr (j, i-j)) !=
                                    WordSet.end ()) {
            dp[i] = true ;
            break;
        }
    }
}
```

```
        }
    return dp[n];
}
```