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)

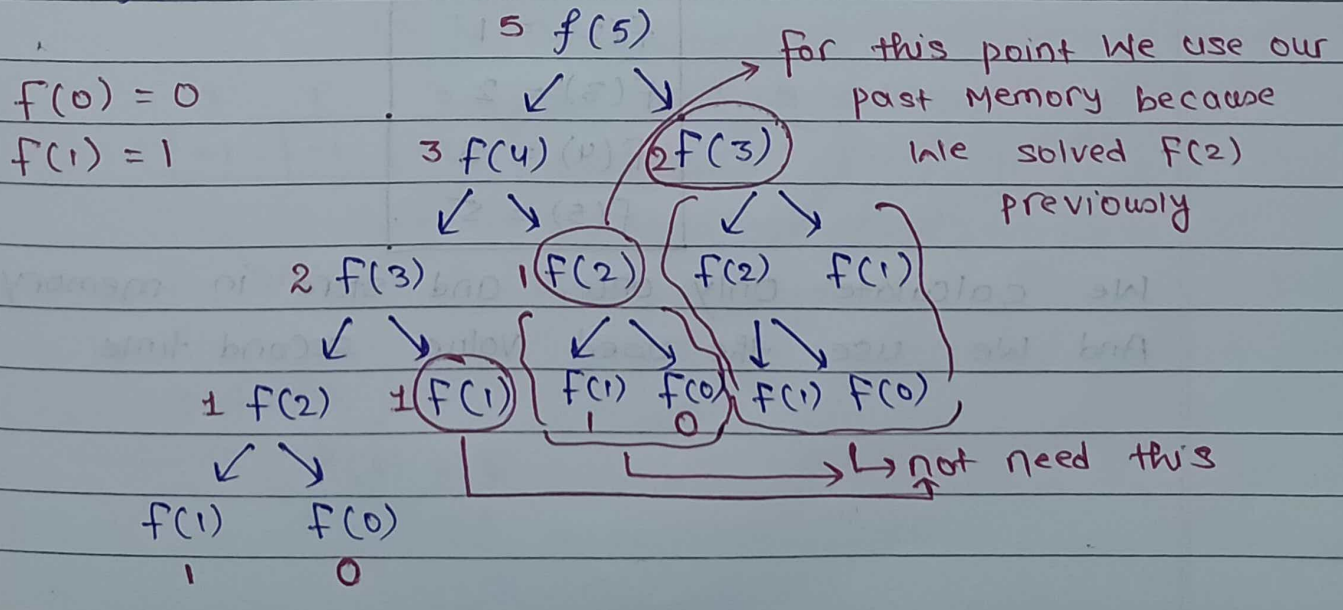
② Tabulation (Bottom up approach) (loop)

space optimization.

→ n^{th} fibonacci number :

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

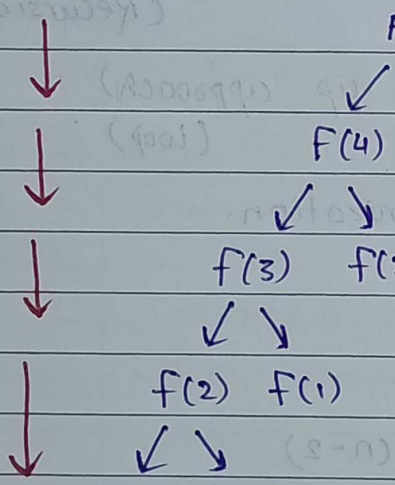
$f(5)$:



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

* 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	After DP
2^n	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

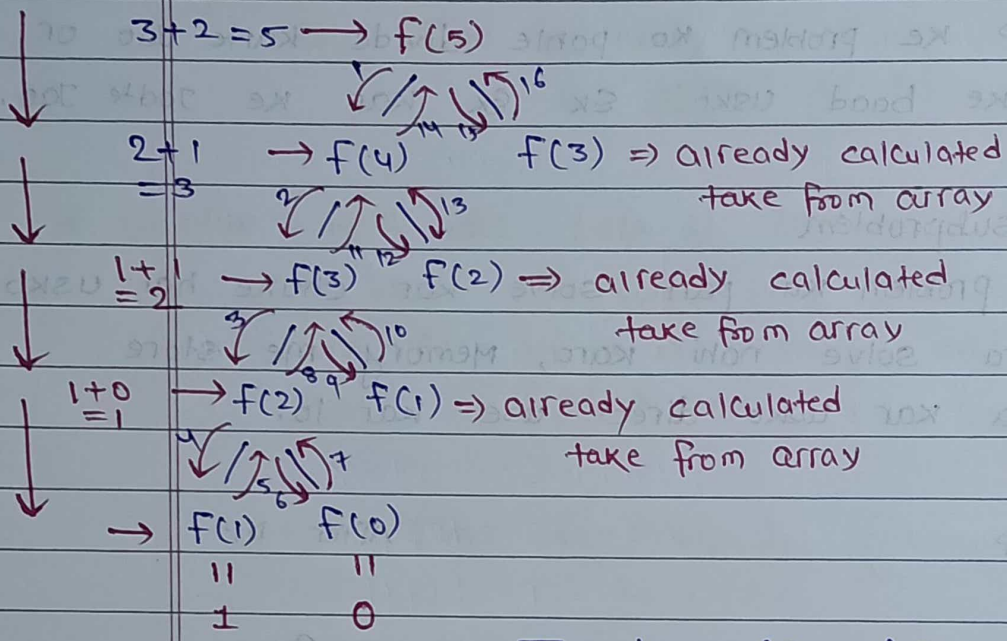
Initially all -1

Initially

F(0) = 0

F(1) = 1

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



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

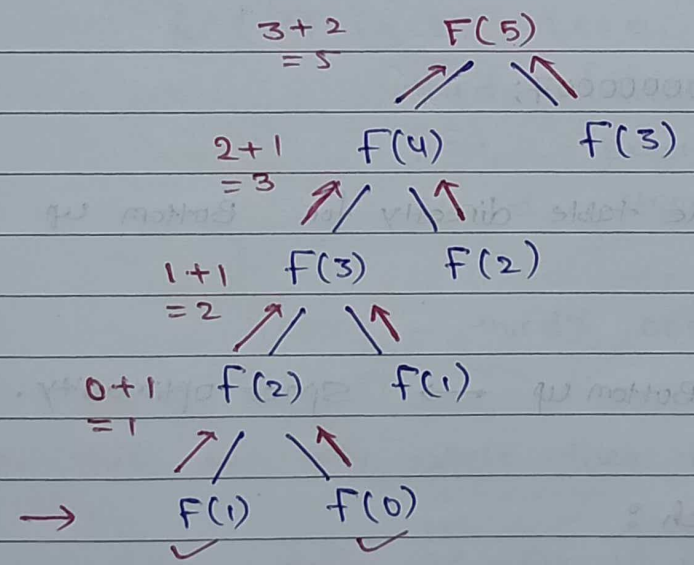
↓ dP ↓ Recursive call

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) → f(1) + f(0) = 1

↳ Next f(3) → f(2) + f(1) = 2

↳ Next F(4) → f(3) + f(2) = 3

↳ Next f(5) → 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;
```

→ In this we only store the two previous value.

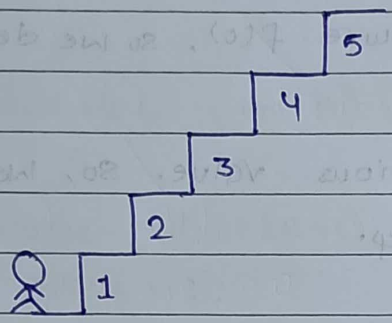
Space Complexity : O(1)

Time Complexity : O(N)

* Climbing stairs : (Leetcode)

Example :

N = 5



⇒ At 1 time take 1 step or 2 step

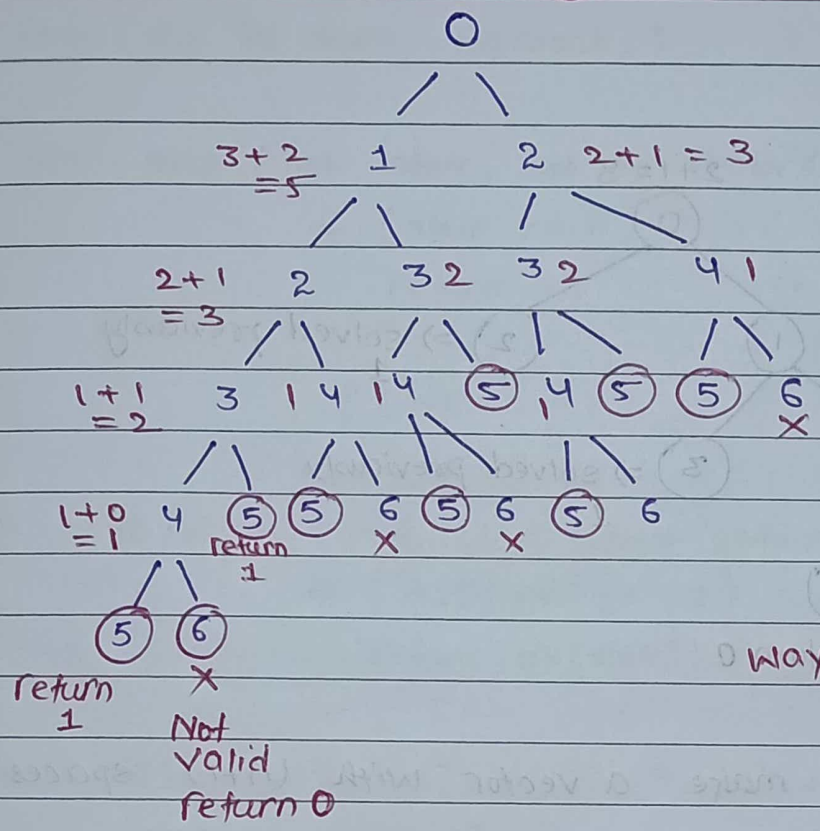
⇒ Find total No. of ways to reach on top?

→ 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.

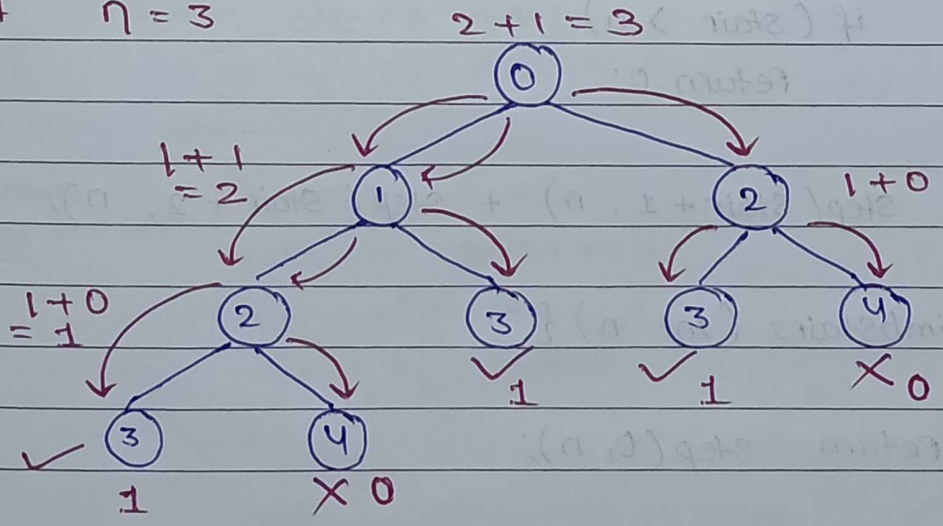


ways = 8

n == 5
return 1;

n > 5
return 0;

Let n = 3



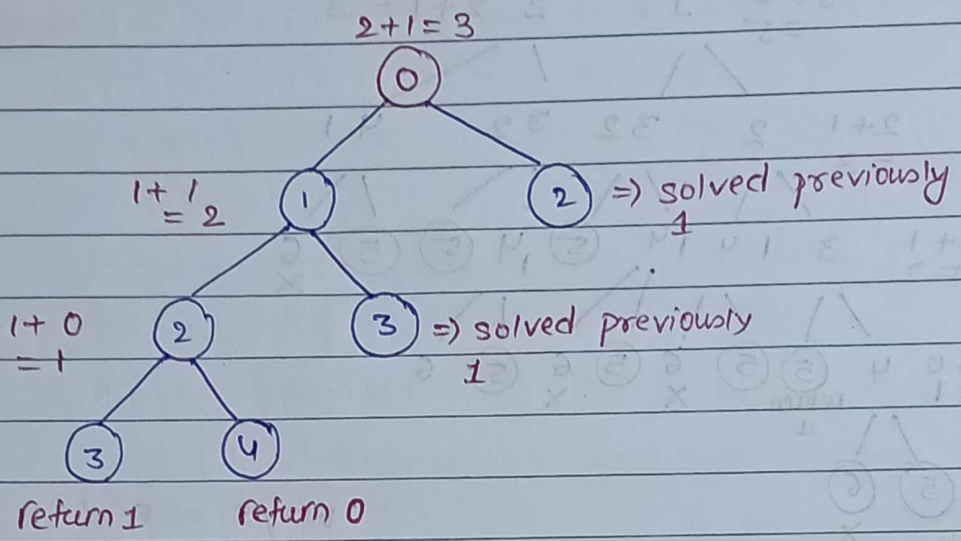
$f(n) = f(n+1) + f(n+2)$

n → return 1

>n → return 0

Optimise :

n = 3



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

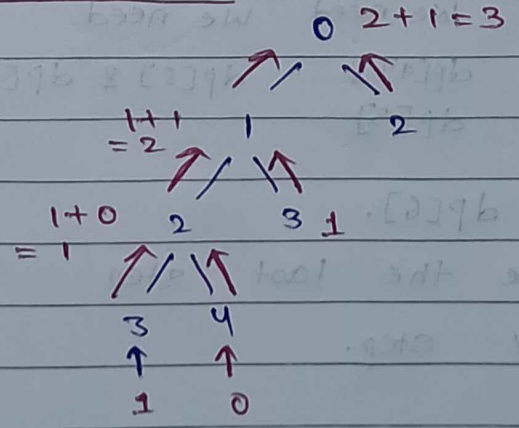
Code (By recursion) :

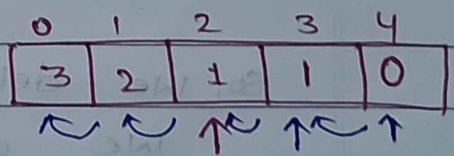
```
int    step (int stair, int n) {
            if (stair == n)
              return 1;
            if (stair > n)
              return 0;

  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

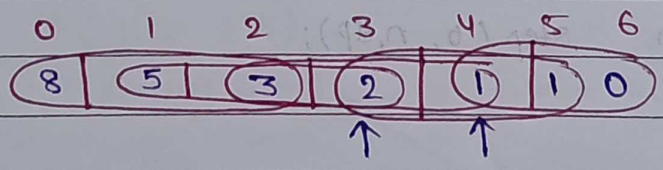
O(N) → space Complexity

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.

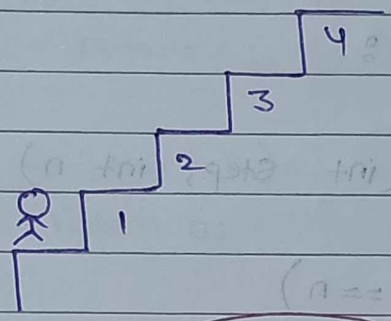
* Count Number Of Hops :

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

1 or 2 or 3

Example :

N = 4



Total = 7

possible :

1 + 1 + 1 + 1

1 + 1 + 2

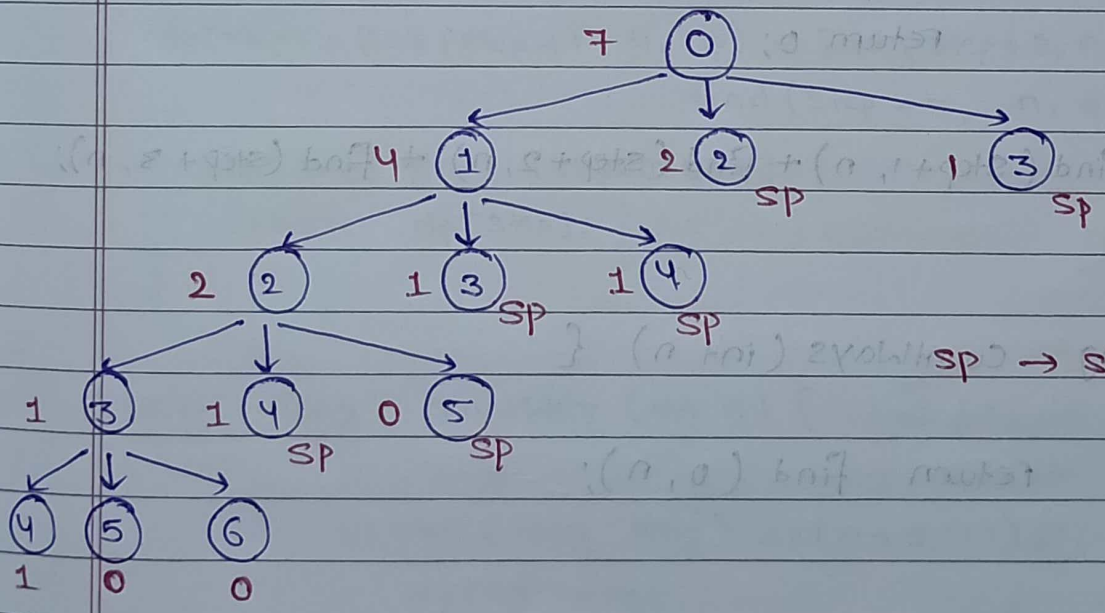
1 + 2 + 1

2 + 1 + 1

2 + 2

1 + 3

3 + 1



sp → solved previously.

For n → return 1

For n+1 → return 0

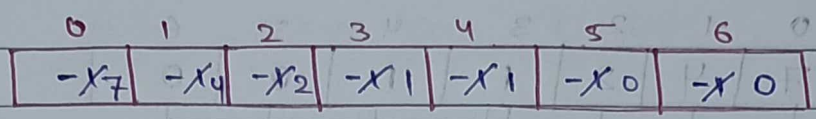
For n+2 → return 0

```
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);
}
```

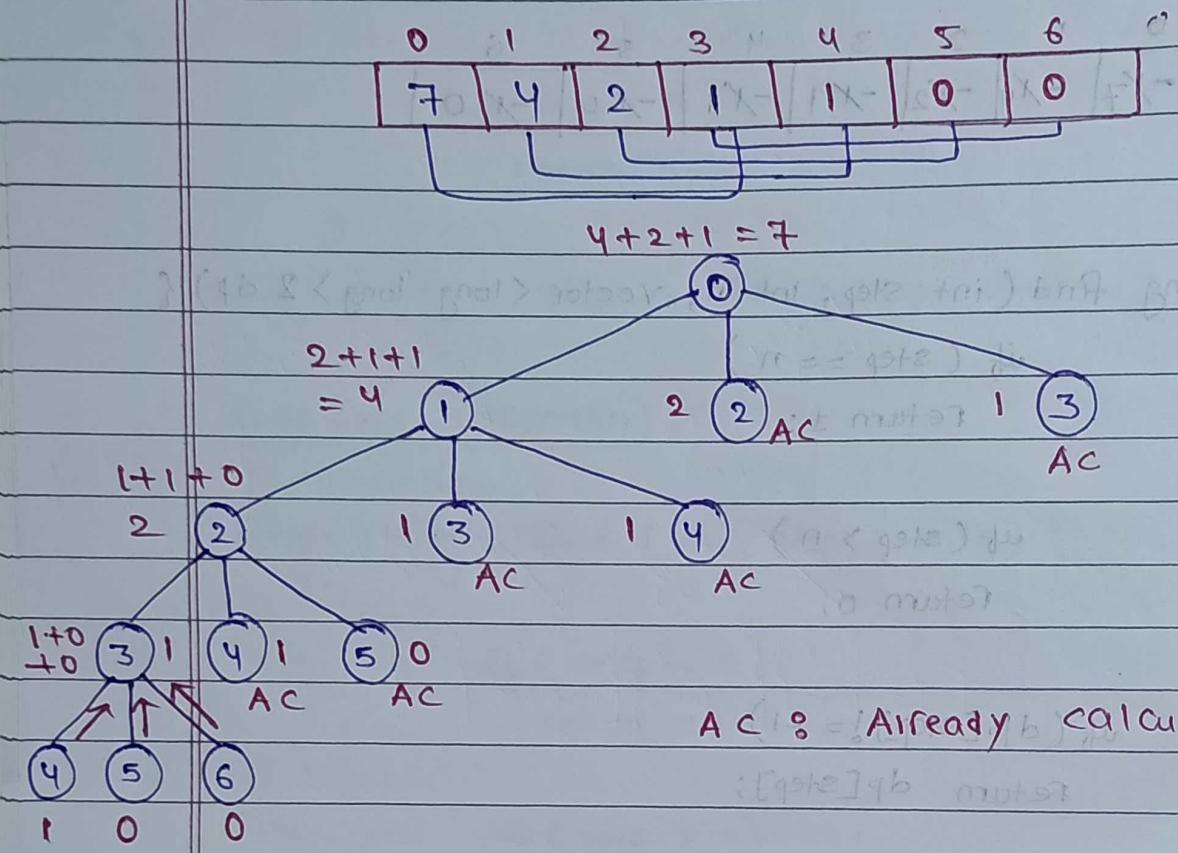
Code (by top down Approach) :



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:



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.

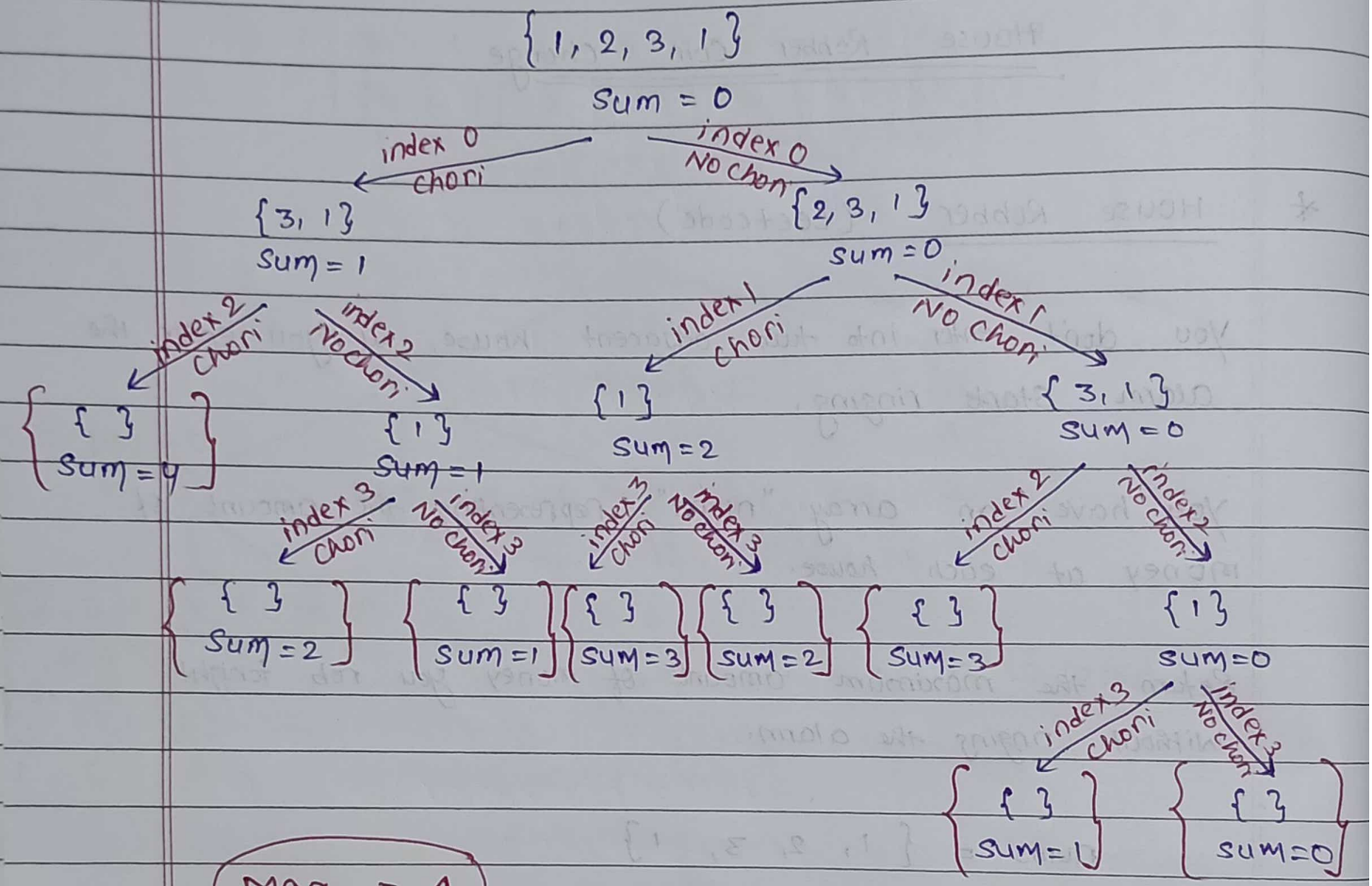
We start with all houses.

→ 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;
```

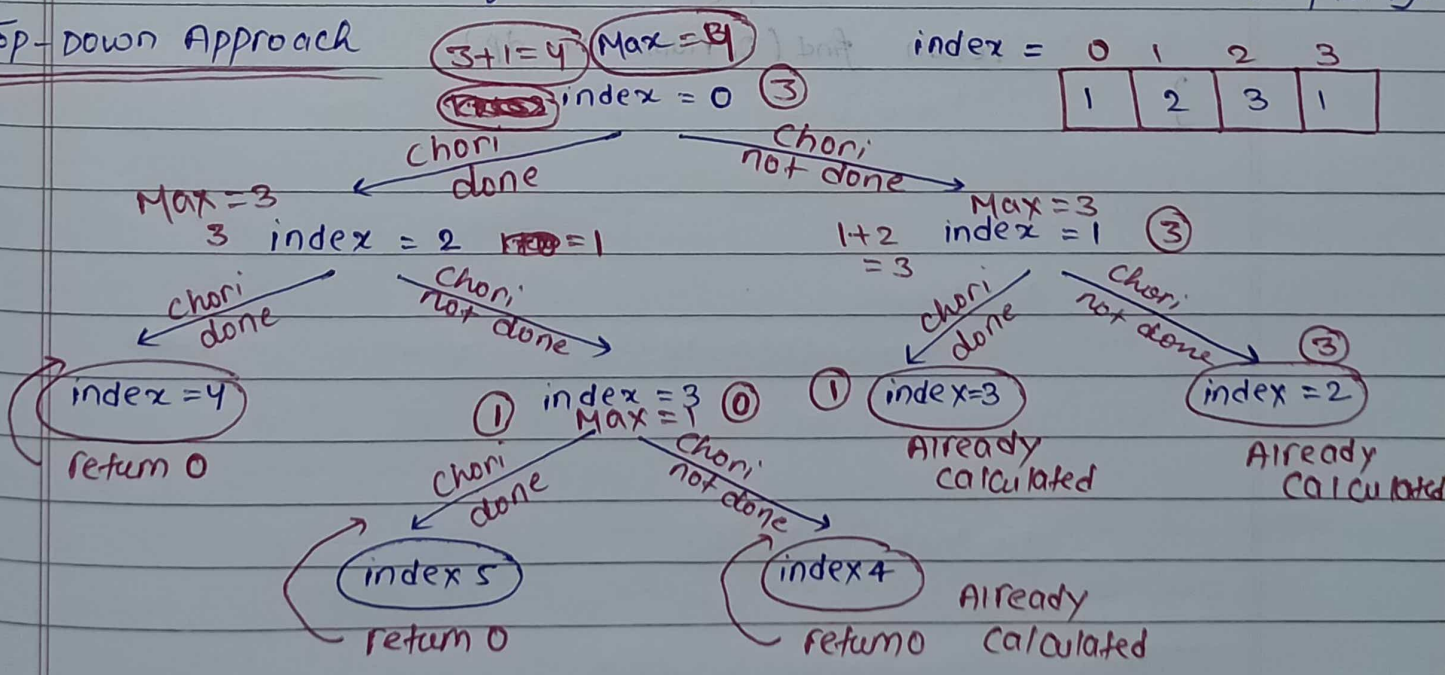
Code :

```
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:

House :

0	1	2	3
1	2	3	1

Dp :

0	1	2	3	4	5
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)

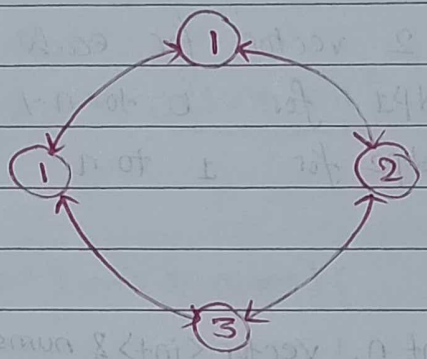
Code :

```
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:

	0	1	2	3
nums :	1	2	3	1

	0	1	2	3	4	5
dp1 :	4	3	3	0	0	-1

	0	1	2	3	4	5
dp2 :	-1	3	3	1	0	0

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);
}
```

* 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 :

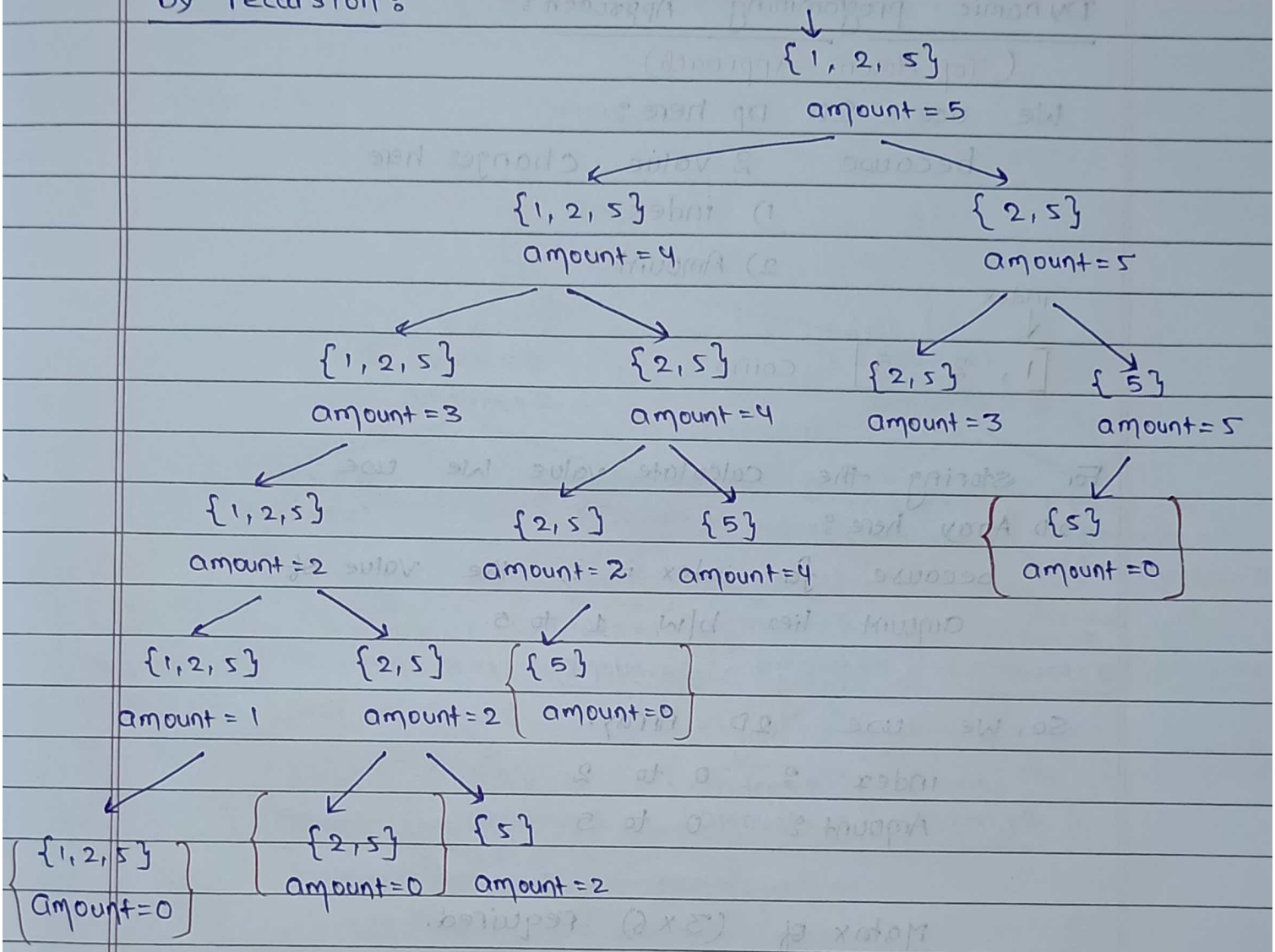
Coins : {1, 2, 5}
Amount : 5

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);
}
```

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)

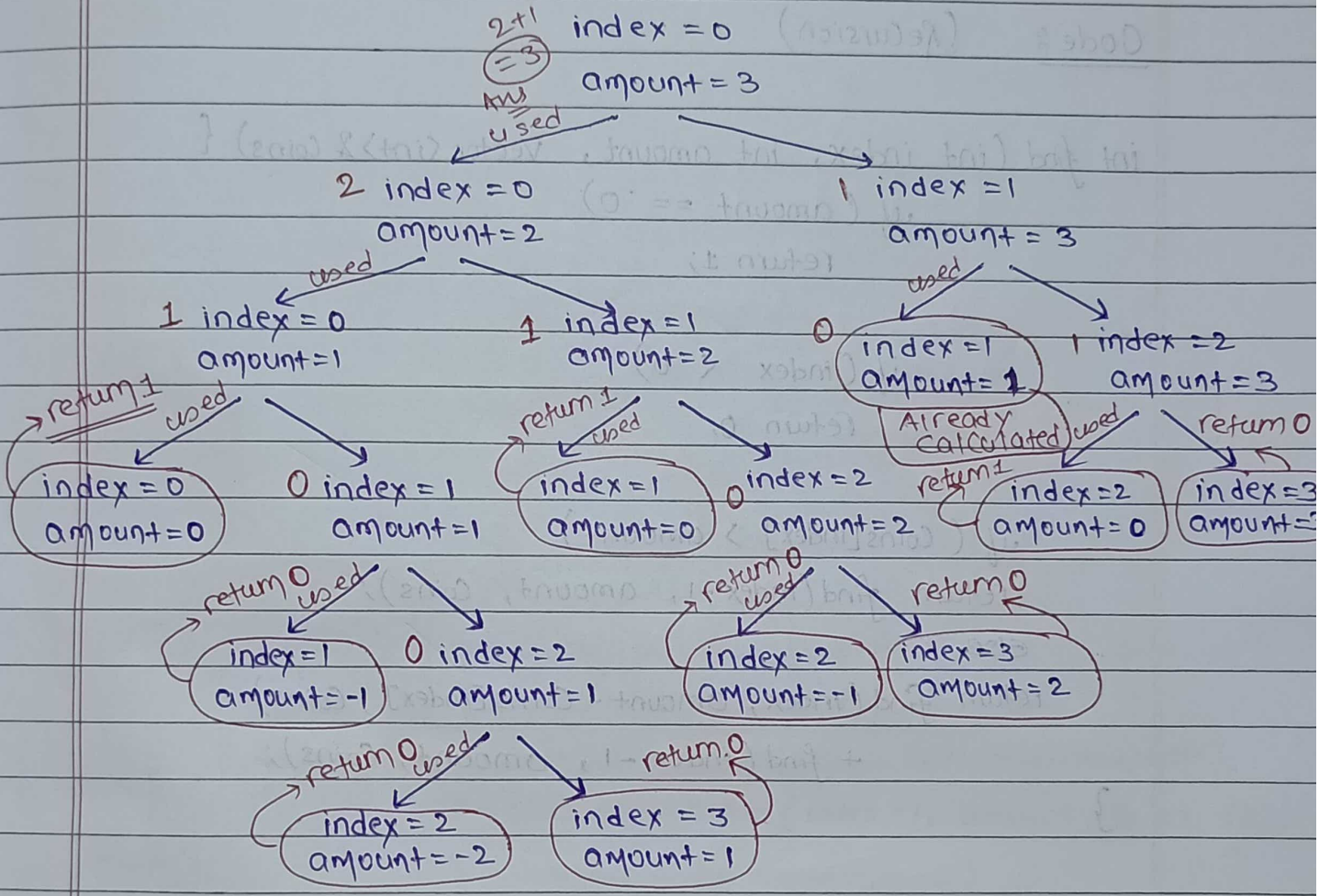
We understand with an Example:

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)

(j)

(i)	0	1	2	3	4	5	(amount)
(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	

```
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 :

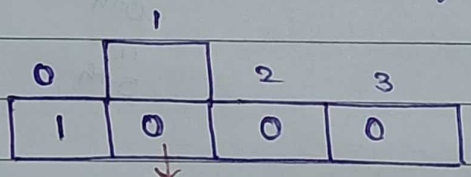
We have to fill the 2D array and we need the last value as the answer.

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

1. previous row
2. Current row

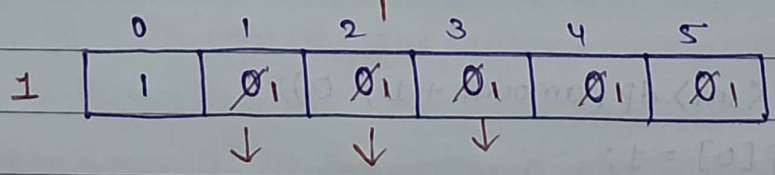
We want to optimize more :

On 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



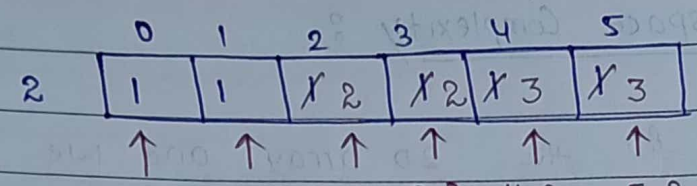
1 rupee ke Coin se 1 rupya bnnahai kitna possible ways hai

(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

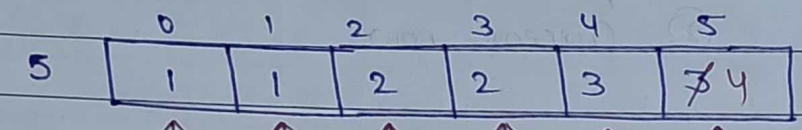
1-2 = -1 does not exist

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 ✗

1-5 = -4 ✗

2-5 = -3 ✗

3-5 = -2 ✗

4-5 = -1 ✗

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 ~~don't~~ 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}

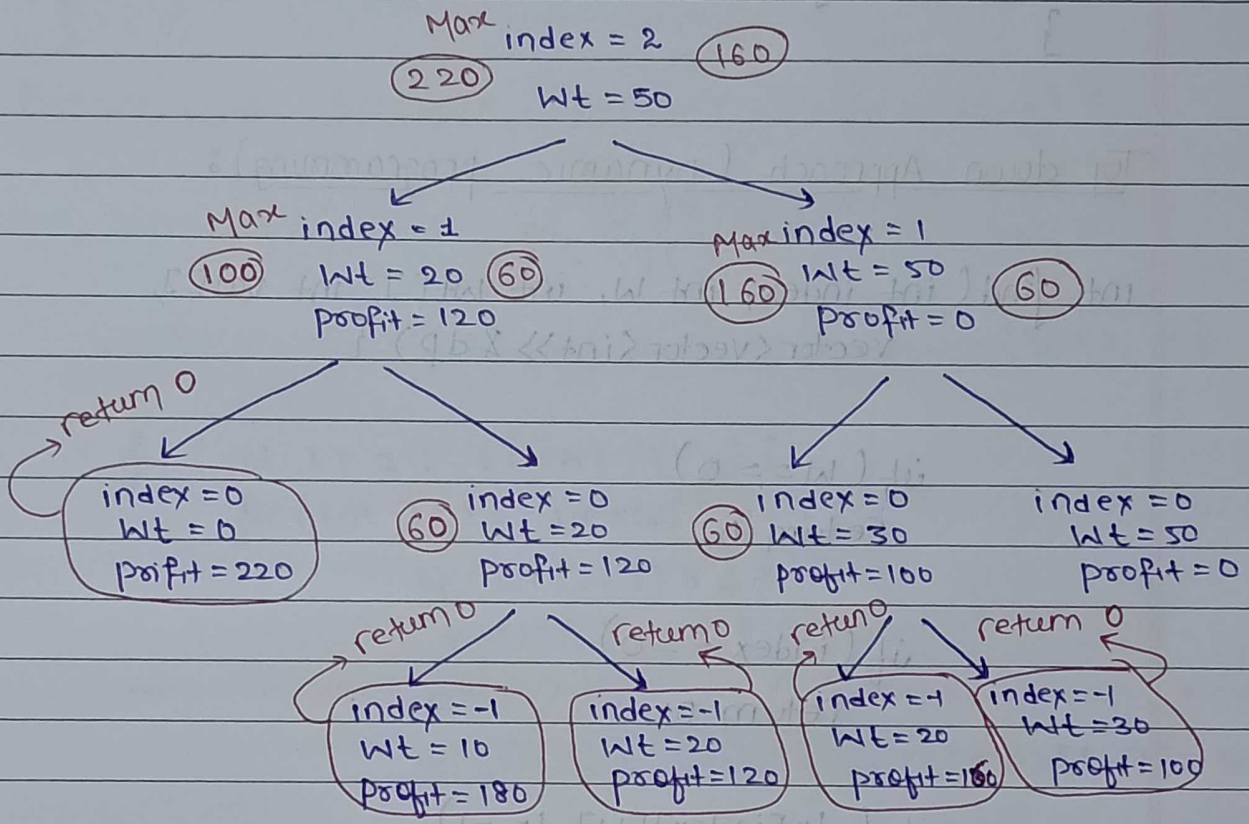
In this we solve by recursion tree.

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];
}
```