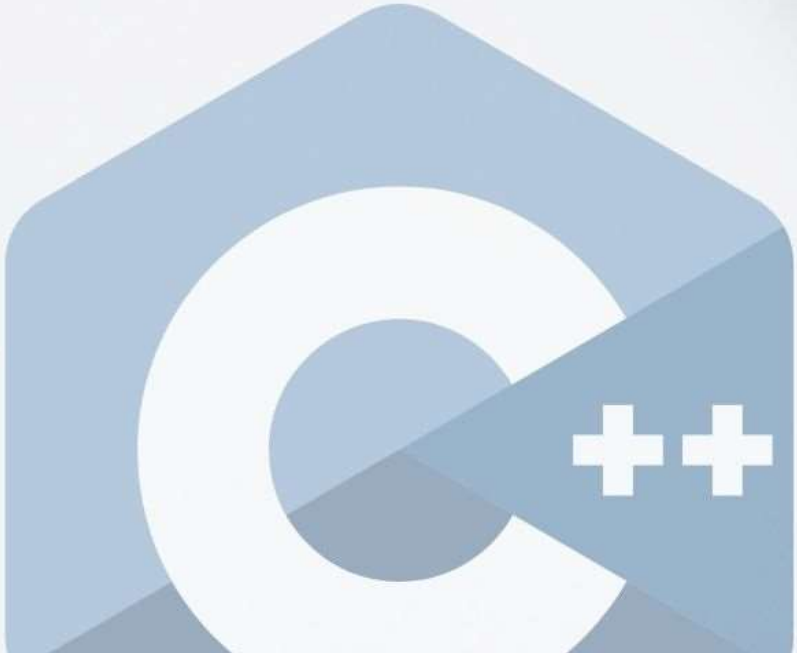
# DYNAMIC PROGRAMMING

KAPIL YADAV



# LEC - 1, FIBONACCI

Friday, 17 June 2022 10:07 AM

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

$f(1) = 1$, $f(0) = 0$. $\rightarrow TC = O(2^n)$

So, Memoization will be :-

```
fib( int i, vector<int>& dp)
{   if( i < 2)
        return i;
    if (dp[i] != -1) return dp[i];

    return dp[i] = f(i-1, dp) + f(i-2,dp);
}
```

TC = O(N), SC = O(N) + O(

Tabulation :-

1.) Write base cases.

dp[0] = 0, dp[1] = 1.

TC = O(N), SC = O(N)

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

optimization in Tabulation;

$O(N^2)$ $O(N)$ :

prev will be n, so return prev.

```
for(i=2; i<=N; i++)
{
    int curr = prev + prev2; prev2 = p
    prev = curr;
}
return prev;
```

TC = O(N)
SC = O(1)

**Memoization:**

```
1 int fib(int n) {
2         vector<int> v(n+1,-1);
3        return fibdp(n,v);
4     }
5
6   int  fibdp(int currentindex, vector<int> &v)
7   {
8      if(currentindex==0 || currentindex==1)
9             return currentindex;
10
11     int currentkey = currentindex;
12     if(v[currentkey]!=-1)
13         return v[currentkey];
14     int fib = fibdp(currentindex-1,v) +
   fibdp(currentindex-2,v) ;
15     v[currentkey]= fib;
```

## Tabulation:

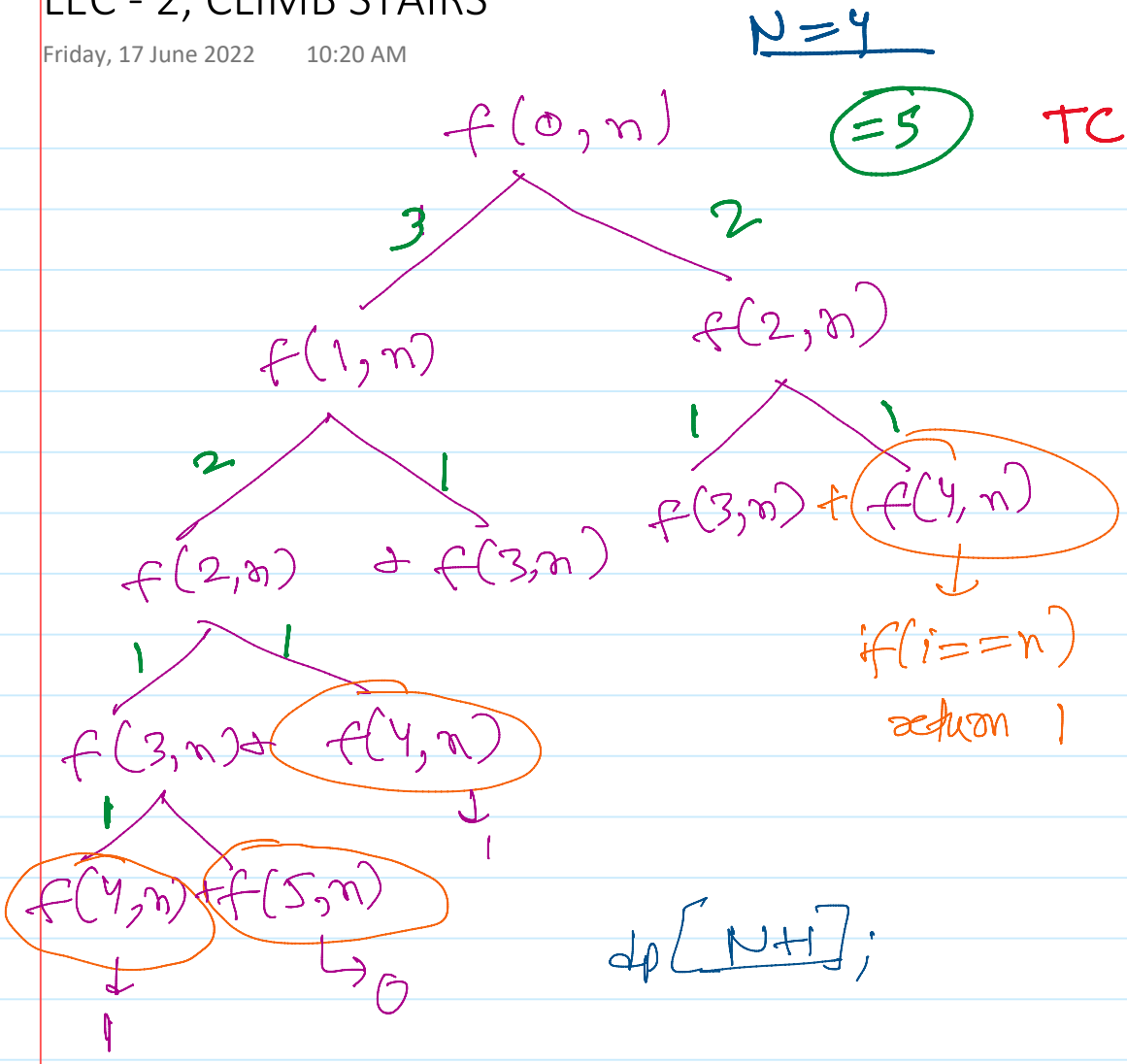
```
int fib(int n) {
        if(n==0 || n==1)
         return n;

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

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

## Space Optimization:

```
int fib(int n) {
       if(n==0 || n==1)
           return n;

        int prev = 1;
        int prev2 = 0;

       for( int i =2;i<=n;i++)
       {int curr= prev + prev2;
```

# LEC - 2, CLIMB STAIRS

Friday, 17 June 2022 10:20 AM




dp[N+1];

```
f( int currentIND, int target, vector<int>& dp)
{
    if( curr = target) return 1;
    if (curr > target) return 0;

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

    return dp[currIND] = f(currInd+1, target, dp) + f(currIND+2,
}
```

TC = O(N)
SC = O(N) + O(N)

**Memoization:**

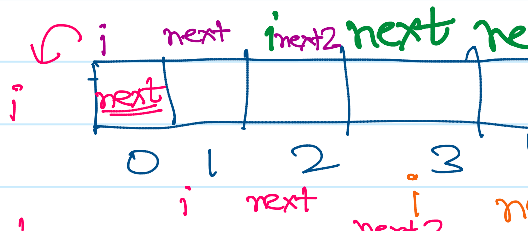
```cpp
int climbStairs(int n) {
        vector<int> v(n+1,-1);
        return noofways(0,n,v);
    }

    int noofways(int currentindex, int n, vector<i
    {
      if(currentindex==n)
          return 1;
      if(currentindex>n)
          return 0;
     int currentkey = currentindex;
        if(v[currentkey]!=-1)
            return v[currentkey];
         else
         {
         int onestep = noofways(currentindex+1,n,v
         int twosteps = noofways(currentindex+2,n,
        v[currentkey] = onestep+twosteps;
        return onestep+twosteps;
         }
    }
```

Tabulation :- take dp(n+2), because at 3, if we take 2 reach to 5.

```cpp
int climbStairs(int n) {
```

TC = O(N
SC = O(N)

## Space Optimization :-




→ return next, because next stores the value for 0 index.

TC = O
SC = O

```
1 int climbStairs(int n) {
2         int next2=0;
3         int next1=1;
4
5         for(int CurrInd = n-1;CurrInd>=0;CurrIn
6         {  int steps= next1 + next2;
7             next2 =next1;
8             next1 = steps;
9         }
10        return next1;
11    }
```

# Lecture-3 Frog JUMP

17 June 2022 11:14

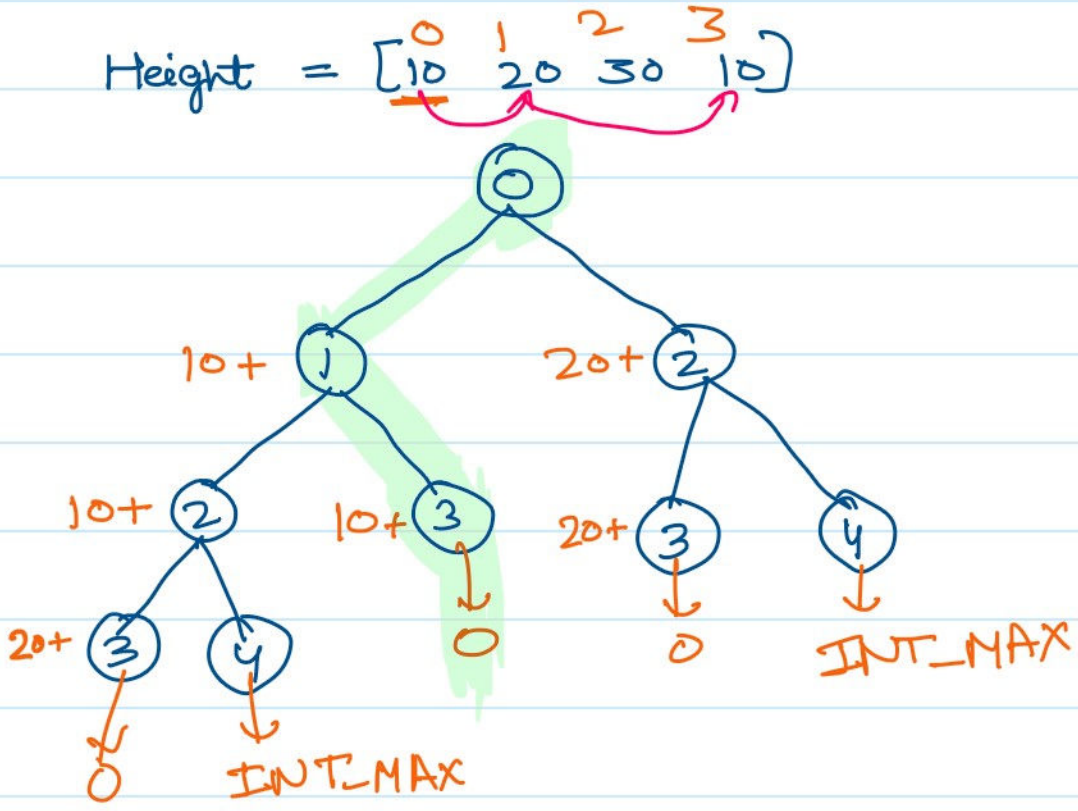
It is same as Min cost to climb the stairs (Leetcode), just the cost is added as
abs(heights[currindex]-height[currindex+1]) in case of jumping one step.
abs(heights[currindex]-height[currindex+2]) in case of jumping two steps

Recursive TC = $O(2^n)$

There is a frog on the 1st step of an N stairs long staircase. The frog wants to reach the Nth stair. HEIGHT[i] is the height of the (i+1)th stair.If Frog jumps from ith to jth stair, the energy lost in the jump is given by |HEIGHT[i-1] - HEIGHT[j-1] |.In the Frog is on ith staircase, he can jump either to (i+1)th stair or to (i+2)th stair. Your task is to find the minimum total energy used by the frog to reach from 1st stair to Nth stair.

For Example

```
If the given 'HEIGHT' array is [10,20,30,10], the answer 20 as the frog can jump
from 1st stair to 2nd stair (|20-10| = 10 energy lost) and then a jump from 2nd
stair to last stair (|10-20| = 10 energy lost). So, the total energy lost is 20.
```




minimum total energy = 20. 0→(1→3)

## Memoization:

```cpp
int recursive(int CurrInd, int targetInd, vector<int>
&heights,vector<int>& dp)
   { if(CurrInd>targetInd)
       return 10000000;
    if(CurrInd==targetInd)
        return 0;
    if(dp[CurrInd]!=-1)
        return dp[CurrInd];
    int stepone = abs(heights[CurrInd+1]-
heights[CurrInd]) +
recursive(CurrInd+1,targetInd,heights,dp);
    int steptwo = abs(heights[CurrInd+2]-
heights[CurrInd]) +
recursive(CurrInd+2,targetInd,heights,dp);
    return dp[CurrInd] = min(stepone,steptwo);
    }

int frogJump(int n, vector<int> &heights)
{      vector<int> dp(n,-1);
       return recursive(0,n-1,heights,dp);
}
```

TC=O(N)
SC=O(N)+O(N)

## Tabulation:

```cpp
int frogJump(int n, vector<int> &heights)
{      vector<int> dp(n+1,0);
       dp[n] = 100000;
       dp[n-1] = 0;
       for(int CurrInd =n-2;CurrInd>=0;CurrInd--)
       {  int stepone = abs(heights[CurrInd+1]-heights[CurrInd]) + dp[CurrInd+1];
          int steptwo = abs(heights[CurrInd+2]-heights[CurrInd]) + dp[CurrInd+2];
          dp[CurrInd] = min(stepone,steptwo);
       }
  return dp[0];
}
```

TC=O(N)
SC=O(N)

**Space Optimization:**

```cpp
int frogJump(int n, vector<int> &heights)
{       int next2 = 100000;
        int next = 0;
        for(int CurrInd =n-2;CurrInd>=0;CurrInd--)
        {  int stepone = abs(heights[CurrInd+1]-heights[CurrInd]) + next;
           int steptwo = abs(heights[CurrInd+2]-heights[CurrInd]) + next2;
         next2 =next;
         next = min(stepone,steptwo);
        }
  return next;
}
```

TC = O(N)
SC = O(1)

LinkedIn/kapilyadav22

# LECTURE-4 FROG JUMP 2 (ATCODER)

Friday, 17 June 2022 12:36 PM

## Problem Statement

There are $N$ stones, numbered $1, 2, \ldots, N$. For each $i$ ($1 \leq i \leq N$), the height of Stone $i$ is $h_i$.

There is a frog who is initially on Stone $1$. He will repeat the following action some number of times to reach Stone $N$:

- If the frog is currently on Stone $i$, jump to one of the following: Stone $i + 1, i + 2, \ldots, i + K$. Here, a cost of $|h_i - h_j|$ is incurred, where $j$ is the stone to land on.

Find the minimum possible total cost incurred before the frog reaches Stone $N$.

## Constraints

- All values in input are integers.
- $2 \leq N \leq 10^5$
- $1 \leq K \leq 100$
- $1 \leq h_i \leq 10^4$

- Same as previous Question, We just need to run our recursive function for K steps every time, so Time Complexity will be : O(K*N)

① variation of codestudio Jump frog problem.

## Memoization:

```
Frog Jump with distance K
//Take input cases from codestudio
#include<iostream>
#include<vector>
#include<climits>
using namespace std;

int recursive(int CurrInd, int targetInd,int k,vector<int>& heights,vector<int>& dp)
   { if(CurrInd>targetInd)
       return 10000000;
    if(CurrInd==targetInd)
        return 0;
    if(dp[CurrInd]!=-1)
        return dp[CurrInd];
        int mincost = INT_MAX;
        for(int i=1;i<=k && i<=targetInd;i++)
    { int stepk = abs(heights[CurrInd+i]-heights[CurrInd]) +
  recursive(CurrInd+i,targetInd,k,heights,dp);
      mincost= min(stepk,mincost);
    }
   dp[CurrInd] = mincost;
    return dp[CurrInd];
    }

int frogJump(int k,int n,vector<int>& heights)
{      vector<int> dp(n,-1);
       return recursive(0,n-1,k,heights,dp);
}

int main() {
    int t,n;
    cin>>t;
    while(t--) {
    cin>>n;
    vector<int> heights(n);
    for(int i=0;i<n;i++)
    cin>>heights[i];
    cout<<frogJump(2,n,heights)<<endl;
}
return 0;
}
```

TC = O(N) x O(K)

SC = O(N) + O(N)

**Tabulation:**

```cpp
int frogJump(int k,int n,vector<int>& heights)
{       vector<int> dp(n+1,0);
         dp[n] = 100000;
        dp[n-1] = 0;
        for(int CurrInd =n-2;CurrInd>=0;CurrInd--)
        {  int mincost = INT_MAX;
         for(int i=1;i<=k && i<n;i++)
     { int stepk = abs(heights[CurrInd+i]-heights[CurrInd]) + dp[CurrInd+i];
       mincost= min(stepk,mincost);
     }
    dp[CurrInd] = mincost;
        }
   return dp[0];
}
```

TC = O(N) x O(K)
SC = O(N)

→ In tabulation, just change dp size by 1 or 2, to cover base cases.

→ Run loops, opposite from Memoization.

→ Replace Recursion function with dp with same index.

→ We can further optimise the space, by taking only K next elements, but in W.C ⇒ if K == N, the space will be O(N), so the WC will be same, hence we can't say using K elements optimizes the space.

# LECTURE-5 Maximum Sum of Non-adjacent elements (HOUSE ROBBER )

Friday, 10 June 2022 9:10 PM

## 198. House Robber

Medium 12898 276 Add to List Share

You are a professional robber planning to rob houses along a street. Each house has a certain amount of mon
constraint stopping you from robbing each of them is that adjacent houses have security systems connected a
**automatically contact the police if two adjacent houses were broken into on the same night.**

Given an integer array `nums` representing the amount of money of each house, return *the maximum amount*
*tonight* ***without alerting the police.***

**Example 1:**

```
Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.
```

**Example 2:**

```
Input: nums = [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
Total amount you can rob = 2 + 9 + 1 = 12.
```

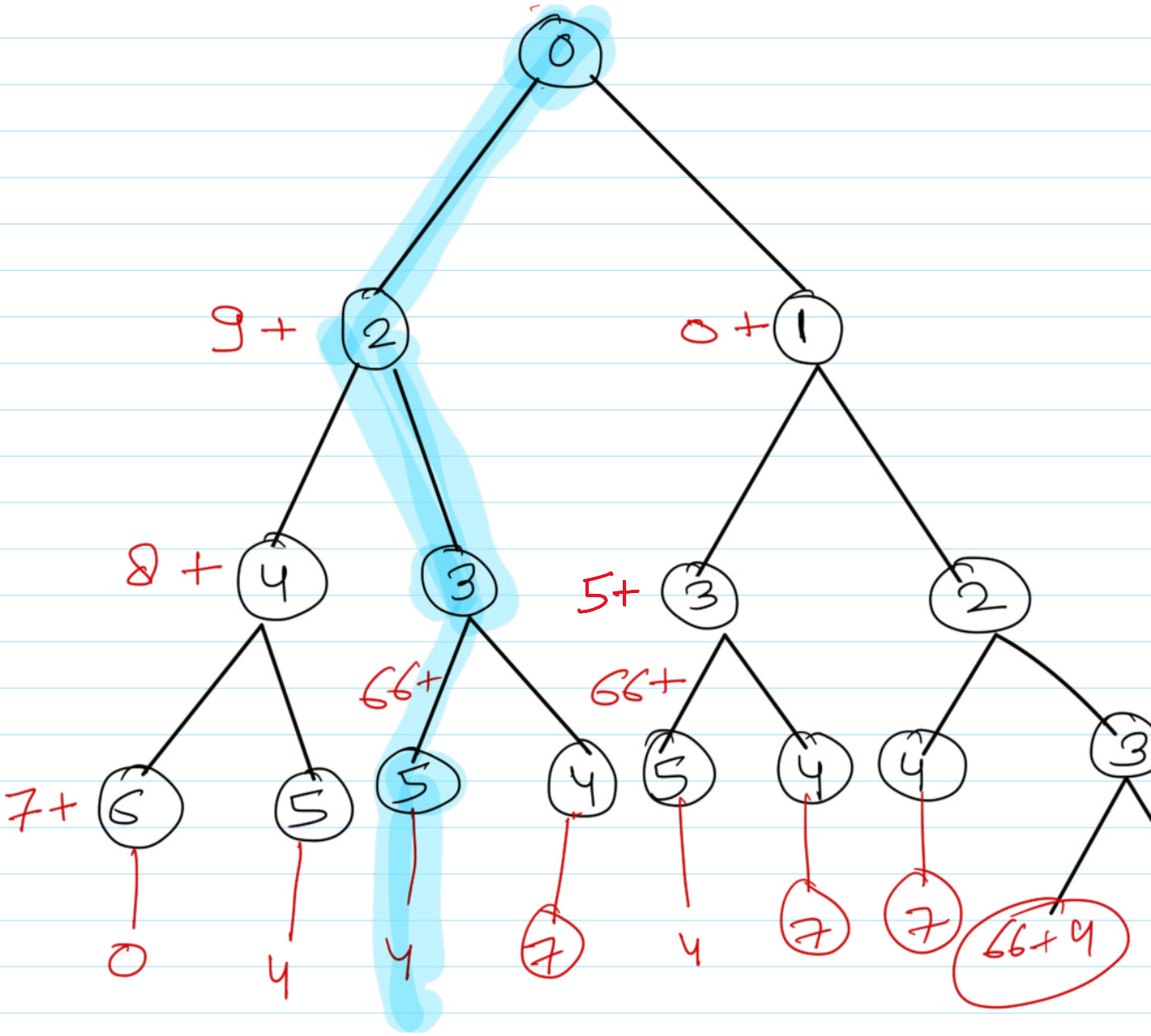
nums = 2 7 9 3 1

Here Robber can rob :- 2 + 9 + 1 = 12

→ 2 + 3 = 5
→ 2 + 1 = 3
→ 7 + 3 = 10
→ 7 + 1 = 8
→ 9 + 1 = 10

NOTE:- He cannot rob two adjacent house, but he can rob
adjacent house, it may be combination of even & odd inde

ex-2.) nums = 9 5 8 66 7 4


0
9+
2
0+
1
8+
4
3
5+
3
2
66+
66+
7+
6
5
5
4
5
4
4
3
0
4
4
7
4
7
7
66+4

GFG:- Recursion + Memoization.

TC=O(
SC=O(N
O(

```cpp
int rob(vector<int>& nums) {
        int n=nums.size();
        vector<int> v(n,-1);
        return nonadjacent(0,n-1,v,nums);
    }

    int nonadjacent(int currentindex, int target, vector<in
vector<int>& nums)
    {
        if(currentindex==target)
            return nums[currentindex];
        if(currentindex>target)
            return 0;

        int currentkey = currentindex;
        if(v[currentkey]!=-1)
            return v[currentkey];
        int rob = nums[currentindex] +
nonadjacent(currentindex+2,target, v,nums);
        int dontrob = nonadjacent(currentindex+1, target, v

       v[currentkey]= max(rob,dontrob);
        return v[currentkey];

    }
```

**TABULATION:**

```cpp
int rob(vector<int>& nums) {
        int n=nums.size();
        vector<int> dp(n+1,0);
        dp[n-1]=nums[n-1];
        dp[n] =0;
        for(int currIndex=n-2;currIndex>=0;currIndex--)
        {   int pick = nums[currIndex] + dp[currIndex+2];
         int notpick = dp[currIndex+1];
         dp[currIndex] = max(pick,notpick);
        }
       return dp[0];
    }
```

TC = O(N
SC = O(N

**SPACE OPTIMIZATION IN TABULATION:**

```cpp
//TABULATION
int rob(vector<int>& nums) {
        //space optimization in tabulation
        int n=nums.size();
        int next2 =0;
        int next = nums[n-1];
        for(int currentindex =n-2;currentindex>=0;currentindex
        int rob = nums[currentindex] + next2;
        int dontrob = next;
        int curramount= max(rob,dontrob);
            next2 = next;
            next = curramount;
        }
        return next;
    }
```

TC = O(N)
SC = O(

```c
int findMaxSum(int *arr, int n) {

        int incl=0;
        int excl=0;

        for(int i=0;i<n;i++)
        { int ni = arr[i] + excl;
          int ne = max(incl,excl);
            incl = ni;
            excl = ne;
        }
        return max(incl,excl);
    }
```

TC=O(N)
SC=O(1).

# LECTURE-6 HOUSE ROBBER -2

18 June 2022 10:57

## 213. House Robber II

Medium 5782 95 Add to List Share

You are a professional robber planning to rob houses along a street. Each house has a certain amount of mo
All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. M
adjacent houses have a security system connected, and **it will automatically contact the police if two adja
were broken into on the same night**.

Given an integer array `nums` representing the amount of money of each house, return *the maximum amount
you can rob tonight* ***without alerting the police***.

**Example 1:**

```
Input: nums = [2,3,2]
Output: 3
Explanation: You cannot rob house 1 (money = 2) and then rob house 3 (money = 2), because th
adjacent houses.
```

- Same as House robber one, but since houses are in Circle, so if you rob first house, you cannot rob last hou
  and vice versa.

  - So, we will execute House robber one two times, for index (0,n-2) and (1,n-1).
  - There can be one edge case that, if there is only one element, it will fail, so
    If(size==1) return nums[0];

**MEMOIZATION:**

```cpp
int rob(vector<int>& nums) {
        int n = nums.size();
        vector<int> dp(n,-1);
        if(n==1)
            return nums[0];
        int excludinglast = maxamount(0,n-2,nums,dp);
        fill(dp.begin(), dp.end(), -1);
        int excludingfirst = maxamount(1,n-1,nums,dp);
        return max(excludinglast,excludingfirst);
    }
int maxamount(int CurrInd,int target,vector<int>& nums,vector<int>& d
{  if(CurrInd==target)
     return nums[CurrInd];
   if(CurrInd>target)
        return 0;
   if(dp[CurrInd]!=-1)
    return dp[CurrInd];

    int rob = nums[CurrInd] + maxamount(CurrInd+2,target,nums,dp);
    int notrob = maxamount(CurrInd+1,target,nums,dp);
    return dp[CurrInd] = max(rob,notrob);
}
```

$TC = O(2N) \cong O(N)$

$SC = O(N) + O(N)$

**TABULATION:**

```cpp
int rob(vector<int>& nums) {
        int n = nums.size();
        vector<int> dp1(n+1,0);
        vector<int> dp2(n+1,0);
        if(n==1)
            return nums[0];
        dp1[n-1]=nums[n-1];
        dp2[0]=nums[0];

        for(int CurrInd=n-2;CurrInd>=1;CurrInd--)
        {   int rob = nums[CurrInd] + dp1[CurrInd+2]
            int notrob = dp1[CurrInd+1];
            dp1[CurrInd] = max(rob,notrob);
        }
      for(int CurrInd=n-2;CurrInd>=0;CurrInd--)
        {   int rob = nums[CurrInd] + dp2[CurrInd+2]
            int notrob = dp2[CurrInd+1];
            dp2[CurrInd] = max(rob,notrob);
        }
        int excludinglast = dp1[1];
        int excludingfirst =dp2[0];
        return max(excludinglast,excludingfirst);
}
```

TC = O(N)
SC = O(N)

**SPACE OPTIMIZATION:**

```cpp
int houserobber1(int currindex,int target,vector<int>& nums) {
   //space optimization in tabulation
        int next2 =0;
        int next = nums[currindex];
        for(int currentindex = currindex-1; currentindex>=target;currentindex
            int rob = nums[currentindex] + next2;
            int dontrob = next;
            int curramount= max(rob,dontrob);
            next2 = next;
            next = curramount;
        }
        return next;
    }

    int rob(vector<int>& nums) {
        //it is most optimized
        int n=nums.size();
        if (n == 1) return nums[0];
        return max(houserobber1(n-1,1,nums),houserobber1(n-2,0,nums));
}
```

TC = O(2N) ≈ O(N)
SC = O(1)

# *LECTURE-7 NINJA'S Training

Friday, 17 June 2022 8:28 PM

## Ninja's Training

76 Difficulty: MEDIUM

Contributed By
Srejan Kumar Bera | Level 1

Avg. time to solve
30 min

Success Rate
50%

**Problem Statement**

Ninja is planing this 'N' days-long training schedule. Each day, he can perform any one of three activities. (Running, Fighting Practice or Learning New Moves). Each activity has son merit points on each day. As Ninja has to improve all his skills, he can't do the same activi two consecutive days. Can you help Ninja find out the maximum merit points Ninja can ea

You are given a 2D array of size N*3 'POINTS' with the points corresponding to each day a activity. Your task is to calculate the maximum number of merit points that Ninja can earn

**For Example**

```
If the given 'POINTS' array is [[1,2,5], [3 ,1 ,1] ,[3,3,3] ],the answer wi
be 11 as 5 + 3 + 3.
```

→ There are 3 activities and we cannot perform any activities consecutive.

Why NOT GREEDY?

| Acti 1 | Acti 2 | Acti 3 | |
|---|---|---|---|
| 10 | 50 | 1 | → Day 1 |
| 5 | 100 | 11 | → Day 2 |

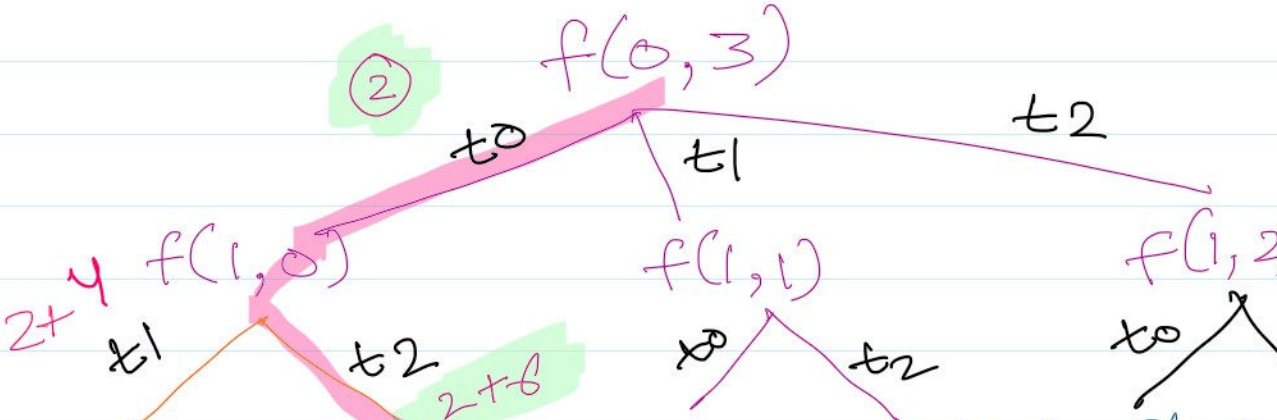
→ so if i pick 50 then, i have to pick max(5, 11) in day 2.

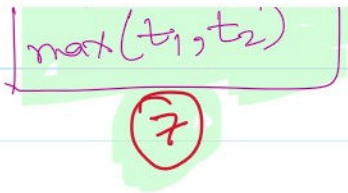
```
{ if(ind = n-1)
  {     int maxi = 0;
        for(int i = 0; i < 3; i++)
            if(i != lasttask)
                maxi = max(maxi, task[n-1][i]);
        return maxi;
  }
  int maxi = 0;
  for(int i = 0; i < 3; i++)
  { if(i != lasttask)
      points = task[day][i] + f(day+1, i   )
      maxi = max(maxi, points);
  } return maxi;
}
```

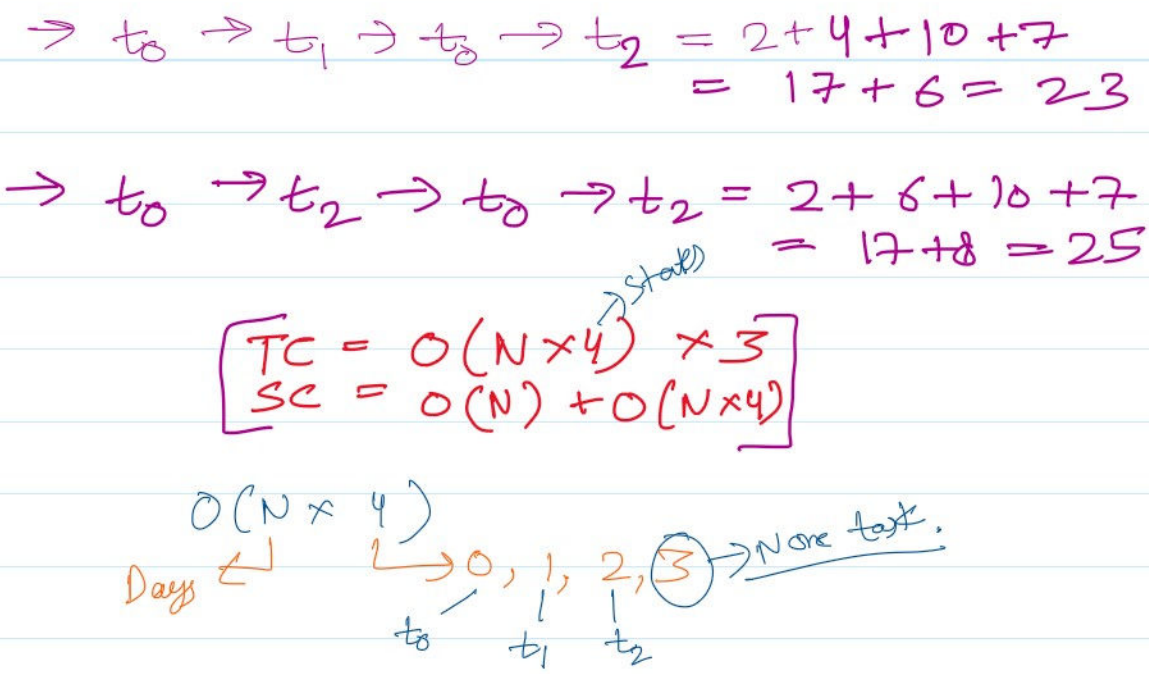
n = 4

| T0 | T1 | T2 | day |
|---|---|---|---|
| 2 | 1 | 3 | d0 |
| 3 | 4 | 6 | d1 |
| 10 | 1 | 6 | d2 |
| 8 | 3 | 7 | d3 |

0 - task1
1 - task2
2 - task3
3 - No task selected





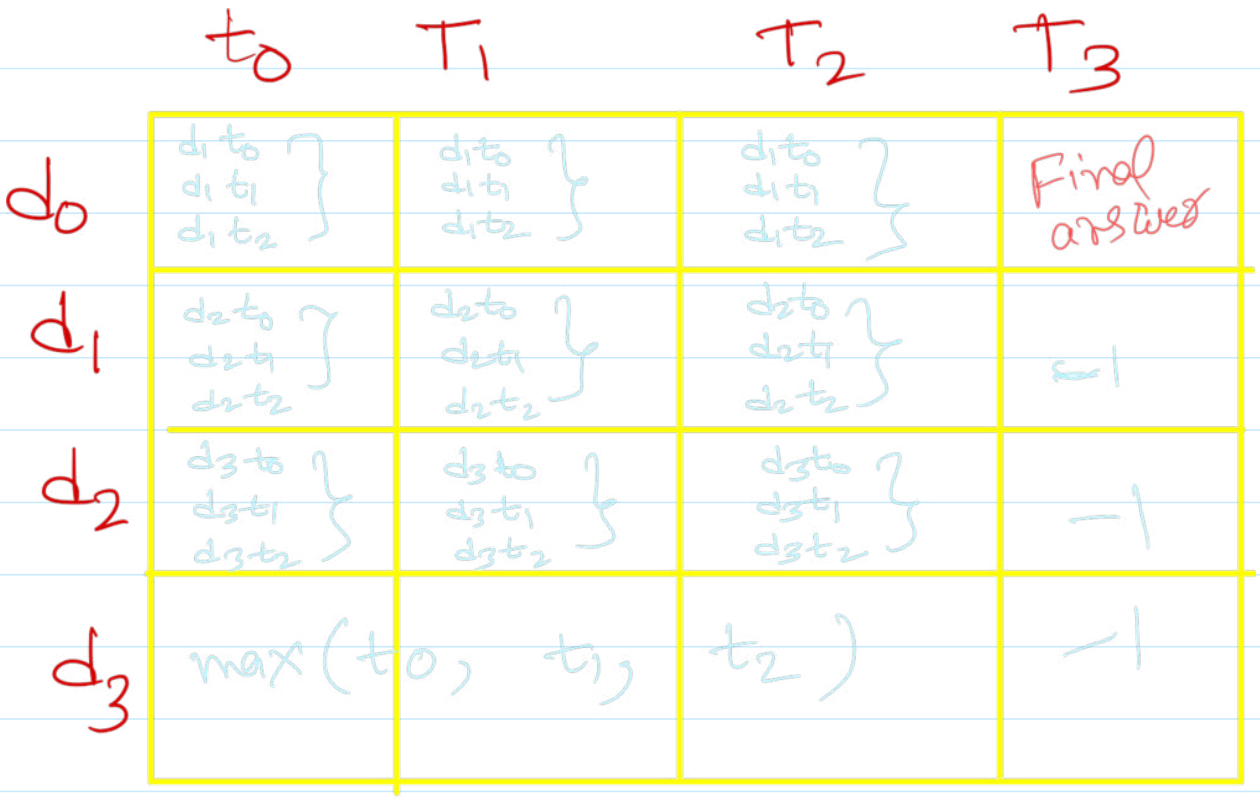

**Memoization:**

```cpp
int calcpoints(int currday, int lasttask,int lastday,vector<vector<
dp, vector<vector<int>> &points)
{   if(currday == lastday) {
    int maxi = 0;
    for(int i =0;i<=2;i++){
        if(i!=lasttask)
            maxi = max(maxi,points[currday][i]);
      }
    return maxi;}

    if(dp[currday][lasttask]!=-1)
        return dp[currday][lasttask];
     int maxi  =0;
     for(int i =0;i<=2;i++){
        if(i!=lasttask){
            int reward = points[currday][i] +
calcpoints(currday+1,i,lastday,dp,points);
```

$TC = O(N \times 4) \times 3$

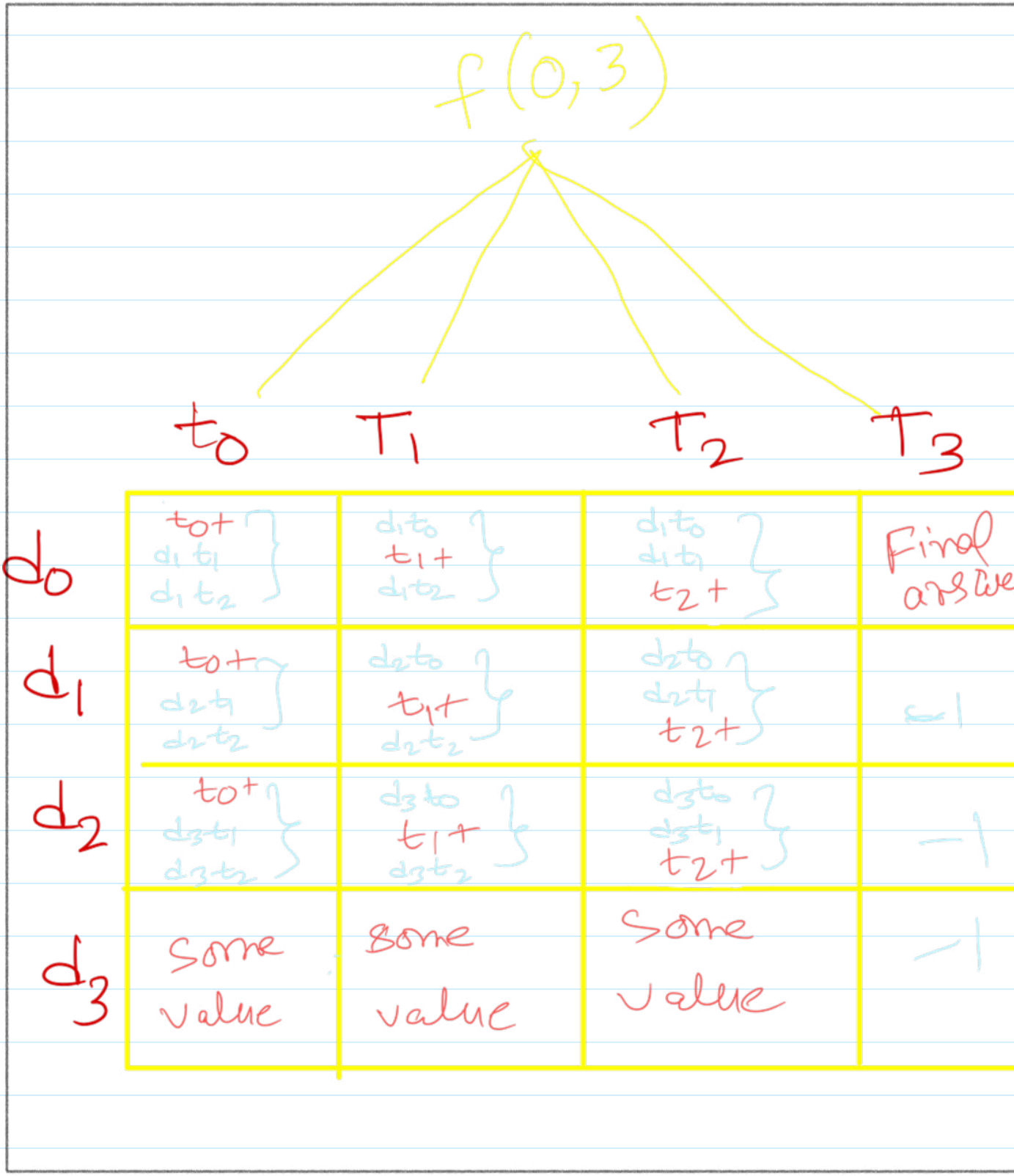
$SC = O(N \times 4)$

TC explained?




→ There are 4 states for each da roughly and at every state we are running a for loop for 3 times.

so $TC = O(N \times 4) \times 3$.


f(0,3)
t0
T1
T2
T3
d0
d1
d2
d3
Some value
Some value
Some value
Final ans
-1
-1

```cpp
int ninjaTraining(int n, vector<vector<int>> &points)
{  vector<vector<int>> dp(n,vector<int>(4,0));
        dp[n-1][0] = max(points[n-1][1],points[n-1][2]);
        dp[n-1][1] =max(points[n-1][0],points[n-1][2]);
        dp[n-1][2] =max(points[n-1][0],points[n-1][1]);
        dp[n-1][3] =max(dp[n-1][2],dp[n-1][0]);

 for(int currday = n-2;currday>=0;currday--)
 {  for(int lasttask = 3;lasttask>=0;lasttask--)
    { for(int i =0;i<=2;i++){
         if(i!=lasttask){
            int reward = points[currday][i] + dp[currday+1][i];
            dp[currday][lasttask] = max(dp[currday][lasttask], reward
         }
     }
}
  }
  return dp[0][3];
}
```

**Space Optimization:**

```cpp
int ninjaTraining(int n, vector<vector<int>> &poin
{  vector<int> prev(4,0);
        prev[0] = max(points[n-1][1],points[n-1][2]
        prev[1] =max(points[n-1][0],points[n-1][2])
        prev[2] =max(points[n-1][0],points[n-1][1])
        prev[3] =max(prev[2],prev[0]);

 for(int currday = n-2;currday>=0;currday--)
 {  vector<int> temp(4,0);
     for(int lasttask = 3;lasttask>=0;lasttask--)
    { for(int i =0;i<=2;i++){
        if(i!=lasttask){
           int reward = points[currday][i] + prev[
           temp[lasttask] = max(temp[lasttask], re
        }
     }
    }
    prev=temp;
  }
  return prev[3];
}
```

| | $T_0$ | $T_1$ | $T_2$ | |
|---|---|---|---|---|
| $d_0$ | $\max(arr[d_0][T_1] + dp[d_1][T_1], arr[d_0][T_2] + dp[d_1][T_2])$ | $\max(arr[d_0][T_0] + dp[d_1][T_0], arr[d_0][T_2] + dp[d_1][T_2])$ | $\max(arr[d_0][T_0] + dp[d_1][T_0], arr[d_0][T_1] + dp[d_1][T_1])$ | max |
| $d_1$ | $\max(arr[d_1][T_1] + dp[d_2][T_1], arr[d_1][T_2] + dp[d_2][T_2])$ | $\max(arr[d_1][T_0] + dp[d_2][T_0], arr[d_1][T_2] + dp[d_2][T_2])$ | $\max(arr[d_1][T_0] + dp[d_2][T_0], arr[d_1][T_1] + dp[d_2][T_1])$ | max |
| $d_2$ | $\max(arr[d_2][t_1] + dp[d_3][t_1], arr[d_2][t_2] + dp[d_3][t_2])$ | $\max(arr[d_2][t_0] + dp[d_3][t_0], arr[d_2][t_2], dp[d_3][t_2])$ | $\max(arr[d_2][t_0] + dp[d_3][t_0], arr[d_2][t_1], + dp[d_3][t_1])$ | max |
| $d_3$ | $\max(t_1, t_2)$ | $\max(t_0, t_2)$ | $\max(t_0, t_1)$ | ma |

$n = 4$

| T0 | T1 | T2 | day |
|---|---|---|---|
| 2 | 1 | 3 | d0 |
| 3 | 4 | 6 | d1 |
| 10 | 1 | 6 | d2 |
| 8 | 3 | 7 | d3 |

| | $T_0$ | $T_1$ | $T_2$ | |
|---|---|---|---|---|
| $d_0$ | 24 | 25 | 25 | 25 |

# *LECTURE -8 DP ON GRIDS

Sunday, 19 June 2022 11:28 AM

- Count Paths
- Count Paths with Obstacles
- Min Path Sum
- Max Path Sum
- Triangle
- 2 start Points

## 1. Unique Paths (leetcode)`

### 62. Unique Paths

Medium 9451 307 Add to List Share

There is a robot on an `m x n` grid. The robot is initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries t
to the **bottom-right corner** (i.e., `grid[m - 1][n - 1]`). The robot can only move either down or right at any point in tim

Given the two integers `m` and `n`, return *the number of possible unique paths that the robot can take to reach the bottom-ri
corner.*

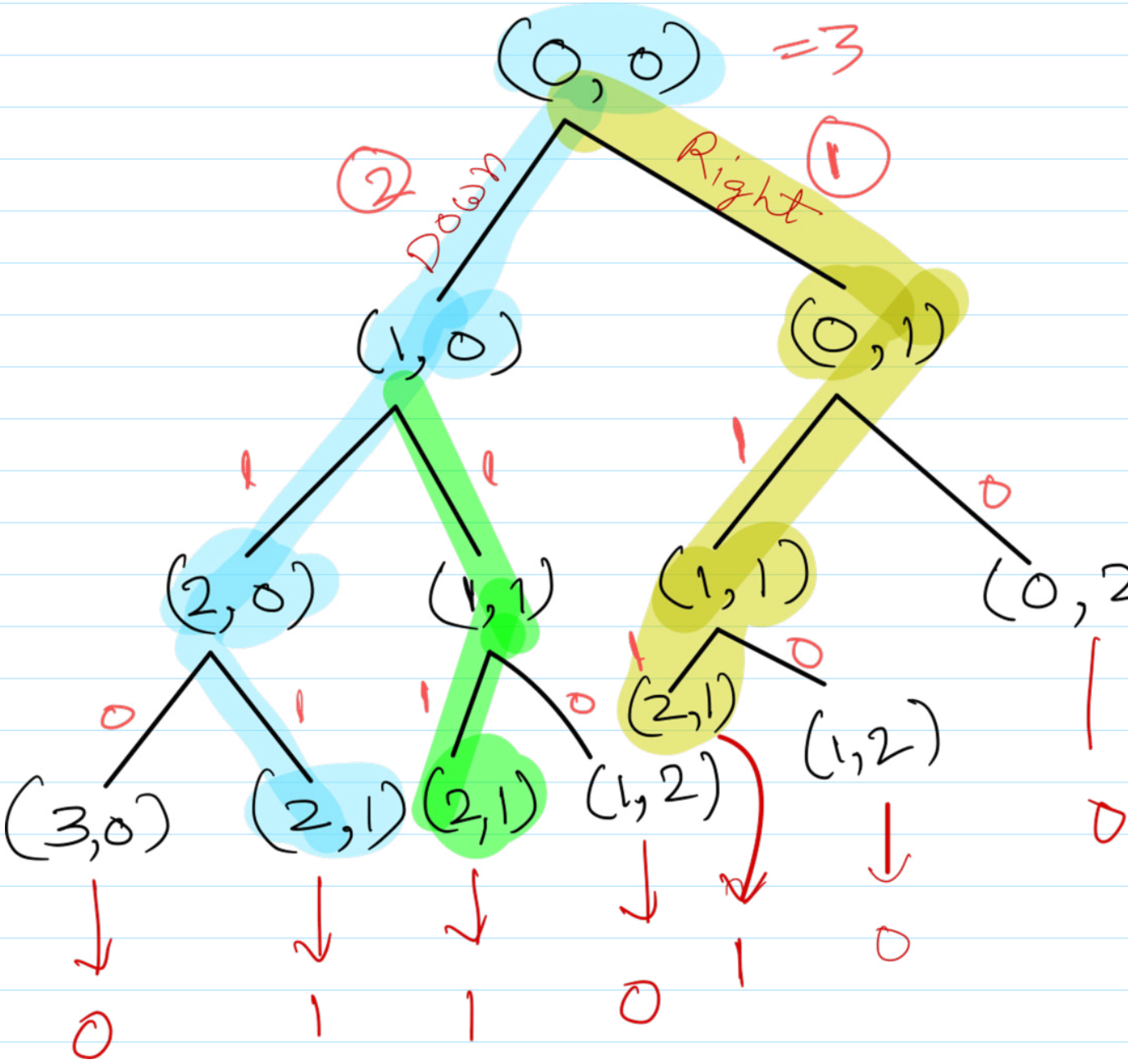
The test cases are generated so that the answer will be less than or equal to `2 * 10^9`.

**Example 1:**


```
Input: m = 3, n = 7
Output: 28
```

**Example 2:**

```
Input: m = 3, n = 2
Output: 3
Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:
1. Right -> Down -> Down
2. Down -> Down -> Right
3. Down -> Right -> Down
```




**Memoization:**

```
int totalpaths(int CurrRow, int CurrCol,int TargetRow, int
TargetColumn,vector<vector<int>>& dp)
    {
        if(CurrRow==TargetRow && CurrCol == TargetColumn)
            return 1;
        if(CurrRow>TargetRow || CurrCol>TargetColumn)
            return 0;
        if(dp[CurrRow][CurrCol]!=-1)
            return dp[CurrRow][CurrCol];
        int movedown = totalpaths(CurrRow+1, CurrCol,TargetRow,TargetColumn,dp);
        int moveright = totalpaths(CurrRow, CurrCol+1,TargetRow,TargetColumn,dp);
        return dp[CurrRow][CurrCol] = movedown+moveright;
```

**Tabulation:**

- For last row and last column, there will be only one way to reach the end, so fill The last row and column with values 1.

```cpp
int uniquePaths(int m, int n) {
     vector<vector<int>> dp(m, vector<int> (n,0));

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

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

      for(int CurrRow=m-2;CurrRow>=0;CurrRow--)
      { for(int CurrCol=n-2;CurrCol>=0;CurrCol--)
          { int movedown = dp[CurrRow+1][CurrCol];
            int moveright = dp[CurrRow][CurrCol+1];
            dp[CurrRow][CurrCol] = movedown+moveright;
          }
      }
       return dp[0][0];
    }
```

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

**OR**

```cpp
int uniquePaths(int m, int n) {
        //striver kind solution
     vector<vector<int>> dp(m,vector<int> (n,0));

      for(int CurrRow=m-1;CurrRow>=0;CurrRow--)
      { for(int CurrCol=n-1;CurrCol>=0;CurrCol--)
          { if(CurrRow==m-1 && CurrCol==n-1)
            dp[CurrRow][CurrCol] =1;
        else{   int movedown=0,moveright=0;
            if(CurrRow<m-1)
                movedown = dp[CurrRow+1][CurrCol];
            if(CurrCol<n-1)
                moveright = dp[CurrRow][CurrCol+1];
             dp[CurrRow][CurrCol] = movedown+moveright;
          }
          }
      }
       return dp[0][0];
    }
```

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

**SPACE OPTIMIZATION:**

- If there is a Prev row & Prev Column, We can Optimize it.

Down & right.

→ To handle the edges, run right for CurrRow < m-1, & Down for CurrCol < n

and Down paths is in next vector.

Right path is in curr vector

So, curr[CurrCol] = Down + right.

Down = next[CurrCol]

Right = curr[CurrCol+1]

→ after a row ends, store curr in next.

next = curr.

→ At the end, return next[0].

```
int uniquePaths(int m, int n) {
        vector<int> next(n,0);
        for(int CurrRow=m-1;CurrRow>=0;CurrRow--)
        {   vector<int> curr(n,0);
            for(int CurrCol=n-1;CurrCol>=0;CurrCol--)
            { if(CurrRow==m-1 && CurrCol==n-1)
                  curr[CurrCol] =1;
          else{   int movedown=0,moveright=0;
              if(CurrRow<m-1)
                  movedown = next[CurrCol];
              if(CurrCol<n-1)
                  moveright = curr[CurrCol+1];
              curr[CurrCol] = movedown + moveright;
             }
            }
              next= curr;
        }
         return next[0];
    }
```

```cpp
int uniquePaths(int m, int n) {
        vector<int> next(n,1);

        for(int CurrRow=m-2;CurrRow>=0;CurrRow--)
        {   vector<int> curr(n,0);
               curr[n-1] = 1;
            for(int CurrCol=n-2;CurrCol>=0;CurrCol--)
            {    int movedown=0,moveright=0;
                  movedown = next[CurrCol];
                    moveright = curr[CurrCol+1];
             curr[CurrCol] = movedown + moveright;
            }
             next= curr;
        }
         return next[0];
    }
```

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



There is one more way to do this question.

**Optimal Way ;** TC: O(m-1) or o(n-1) SC = O(1)

```cpp
int uniquePaths(int m, int n) {
        //it will done using combinatorics
        // ncr, where n is the number of steps require, r is the arrangement of these steps.
        // TC = O(m-1) or (n-1)
      int Number_of_steps = n+m-2 ; // (m-1) + (n-1)
      int r = m-1; // or n-1;
        double totalpaths = 1;
        for(int i = 1;i<=r;i++)
        {
```

# LECTURE - 9 Unique Paths II (leetcode) | Count Path with Obstacles

19 June 2022 17:03

## 63. Unique Paths II

Medium 5361 394 Add to List Share

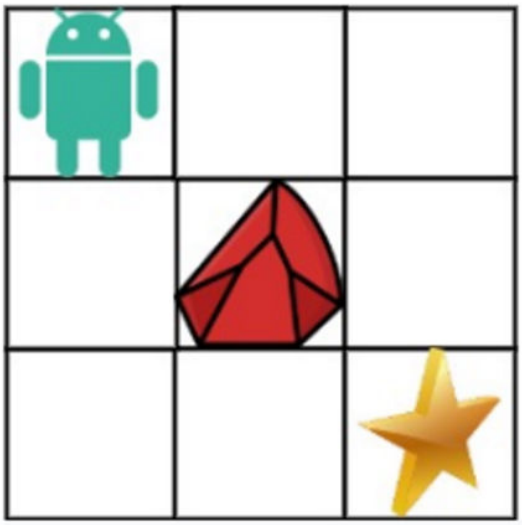
You are given an `m x n` integer array `grid`. There is a robot initially located at the **top-left corner** (i.e., `grid[0` robot tries to move to the **bottom-right corner** (i.e., `grid[m-1][n-1]`). The robot can only move either down point in time.

An obstacle and space are marked as `1` or `0` respectively in `grid`. A path that the robot takes cannot include that is an obstacle.

Return *the number of possible unique paths that the robot can take to reach the bottom-right corner.*

The testcases are generated so that the answer will be less than or equal to `2 * 10^9`.

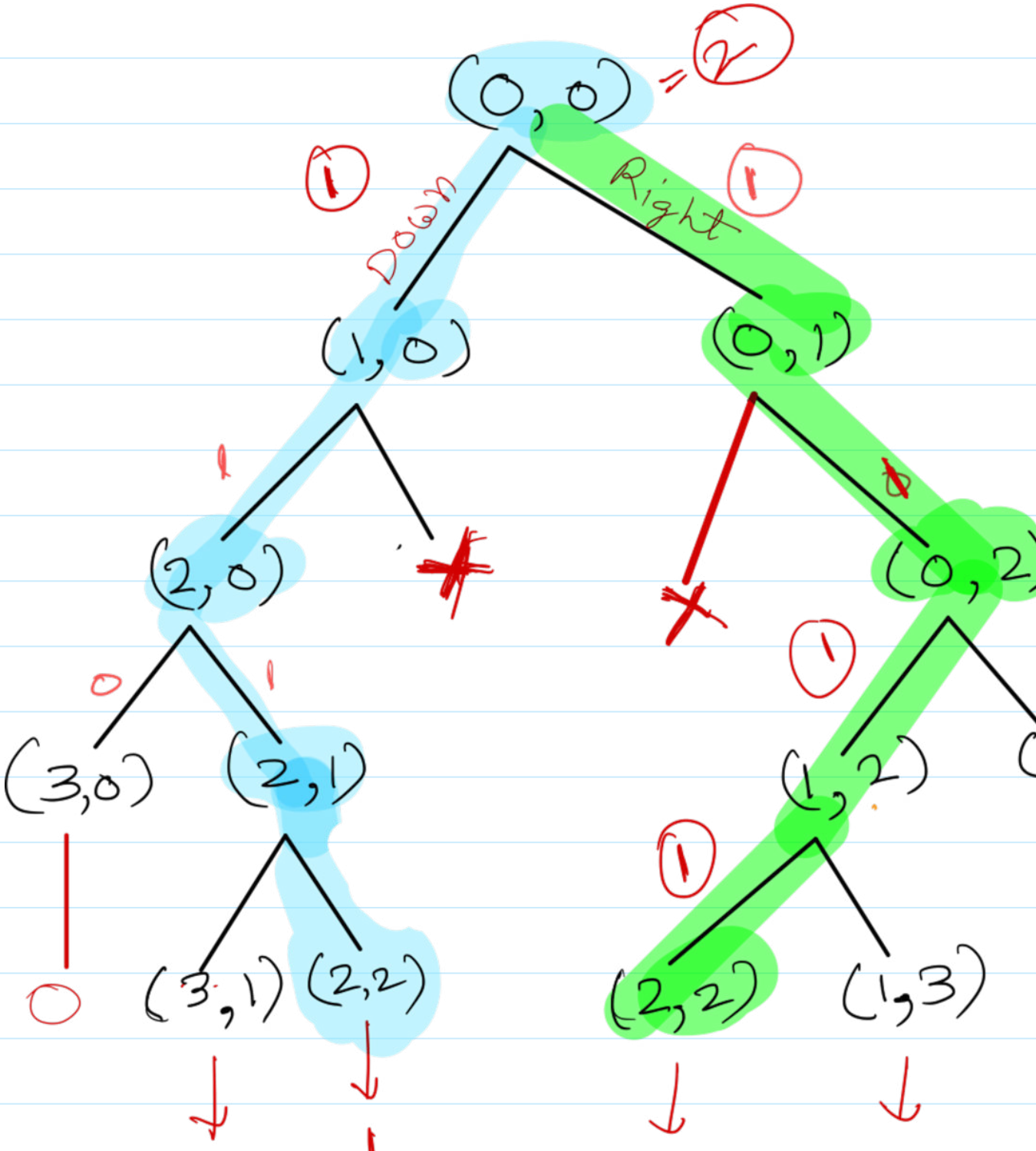
**Example 1:**



```
Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2
Explanation: There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right
```

**Example 2:**

Ex → [[0,0,0],[0,1,0],[0,0,0]]



**MEMOIZATION:**

```
int totalpaths(int CurrRow, int CurrCol,vector<vector<int>>& obstacleGrid,vector<vec
dp)
    {  int targetRow = obstacleGrid.size()-1;
       int targetCol = obstacleGrid[0].size()-1;

     if(CurrRow == targetRow && CurrCol == targetCol && obstacleGrid[CurrRow][CurrCo
         return 1;
     if(CurrRow> targetRow || CurrCol>targetCol ||obstacleGrid[CurrRow][CurrCol]==1)
         return 0;

     if(dp[CurrRow][CurrCol]!=-1)
         return dp[CurrRow][CurrCol];
     int moveright = totalpaths(CurrRow,CurrCol+1,obstacleGrid,dp);
     int movedown = totalpaths(CurrRow+1,CurrCol,obstacleGrid,dp);
     return dp[CurrRow][CurrCol] = moveright + movedown;
    }


    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();
        vector<vector<int>> dp(m,vector<int> (n,-1));
        return totalpaths(0,0,obstacleGrid,dp);
    }
```

**TC:** O(M*N)
**SC :** O(M*N) + O(M-1) +O(N-1) recursive space. (N-1)+(M-1) is the path length

**TABULATION:**

```
int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();
        vector<vector<double>> dp(m,vector<double> (n,0));

    for(int CurrRow=m-1;CurrRow>=0;CurrRow--)
         { for(int CurrCol=n-1;CurrCol>=0;CurrCol--)
          {if(CurrRow ==m-1 && CurrCol==n-1 && obstacleGrid[CurrRow][CurrC
              dp[CurrRow][CurrCol]=1;
          else if(obstacleGrid[CurrRow][CurrCol]==1)
              dp[CurrRow][CurrCol]==0;
         else { double movedown =0;
           double moveright =0;
```

**SPACE OPTIMIZATION:**

```
int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();
        vector<double> next(n,0);

    for(int CurrRow=m-1;CurrRow>=0;CurrRow--)
         { vector<double> curr(n,0);
        for(int CurrCol=n-1;CurrCol>=0;CurrCol--)
          { if(CurrRow ==m-1 && CurrCol==n-1 && obstacleGrid[CurrRow]
  [CurrCol]==0)
              curr[CurrCol]=1;
          else if(obstacleGrid[CurrRow][CurrCol]==1)
              curr[CurrCol]=0;
         else { double movedown =0;
           double moveright =0;
           if(CurrRow<m-1)
             movedown = next[CurrCol];
           if(CurrCol<n-1)
              moveright = curr[CurrCol+1];
          curr[CurrCol] = movedown+moveright;
          }
          }
          next = curr;
      }
       return (int)next[0];
    }
```

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

LinkedIn/kapilyadav22

# LECTURE-10 MINIMUM PATH SUM IN GRID

19 June 2022 17:37

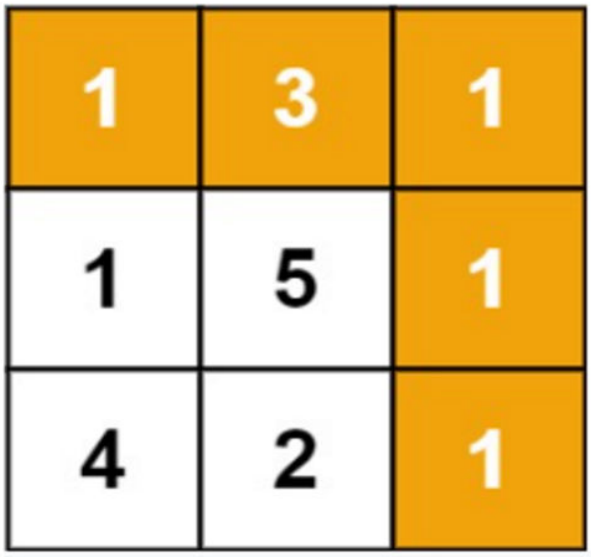
## 64. Minimum Path Sum

Medium 7583 106 Add to List Share

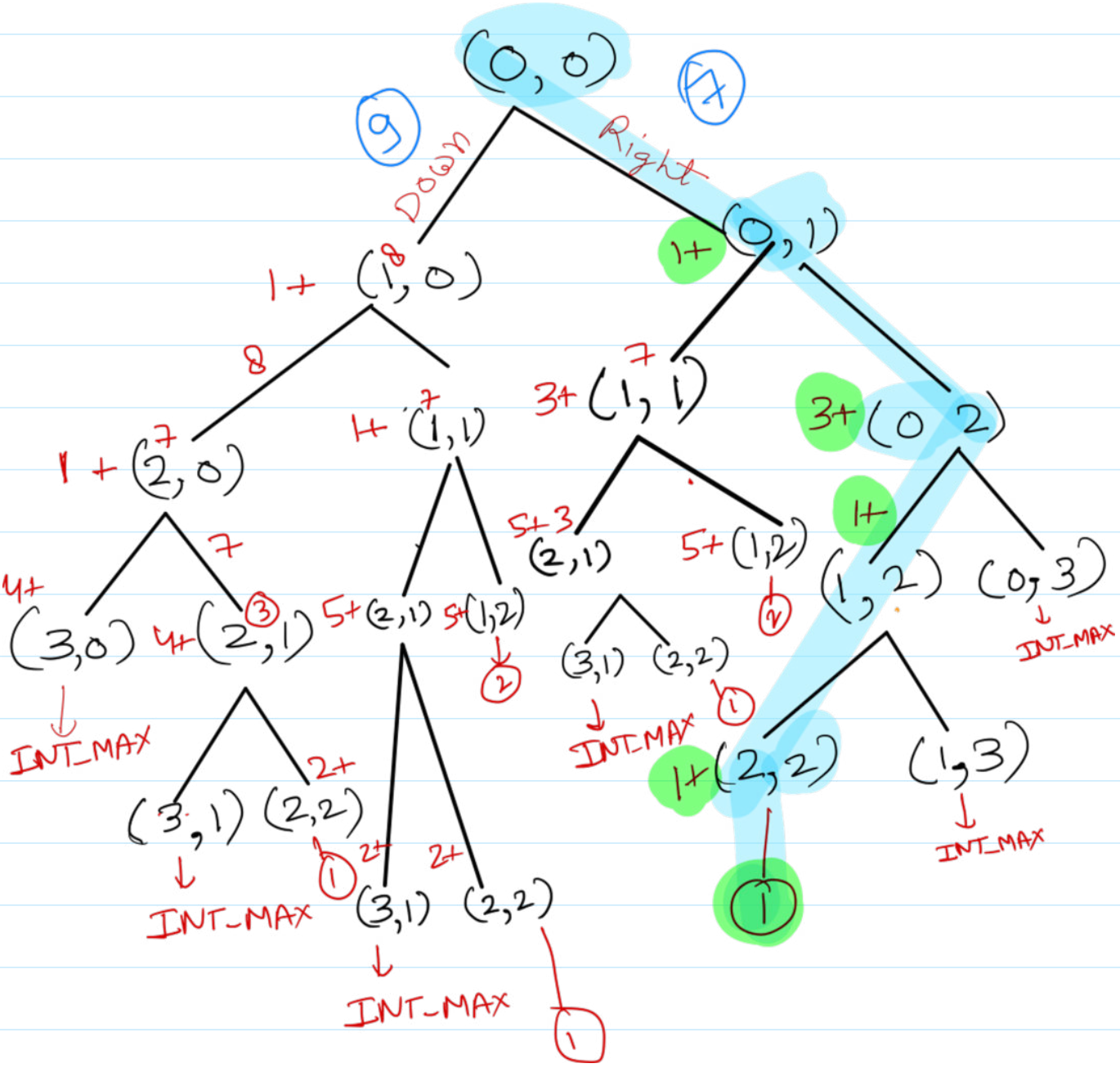
Given a `m x n` `grid` filled with non-negative numbers, find a path from top left to bottom right minimizes the sum of all numbers along its path.

**Note:** You can only move either down or right at any point in time.

**Example 1:**




```
Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
```




**MEMOIZATION\:**

```
int findpath(int CurrRow, int CurrCol, int TargetRow, int TargetCol,
vector<vector<int>>& grid, vector<vector<int>> &dp)
   {  if(CurrRow==TargetRow && CurrCol == TargetCol)
        return grid[CurrRow][CurrCol];
      if(CurrRow>TargetRow || CurrCol>TargetCol)
       return INT_MAX;
      if(dp[CurrRow][CurrCol]!=-1)
          return dp[CurrRow][CurrCol];
       int movedown =  findpath(CurrRow+1,CurrCol,TargetRow, TargetCol,gr
       int moveright = findpath(CurrRow,CurrCol+1,TargetRow,TargetCol,gri
    return dp[CurrRow][CurrCol] =  grid[CurrRow][CurrCol] + min(movedown,
```

TC : O(M*N)
SC : O(M*N) + O(M-1)+ O(N-1) recursive space.  (N-1)+(M-1) is the path length.

**TABULATION:**

```cpp
int minPathSum(vector<vector<int>>& grid) {
        int rowsize = grid.size();
         int colsize = grid[0].size();
         vector<vector<int>> dp(rowsize,vector<int> (colsize,INT_MAX));
         for(int CurrRow=rowsize-1;CurrRow>=0;CurrRow--)
         { for(int CurrCol = colsize-1;CurrCol>=0;CurrCol--)
             {
           if(CurrRow==rowsize-1 && CurrCol==colsize-1)
               dp[CurrRow][CurrCol]=grid[CurrRow][CurrCol];
            else{
             int movedown=INT_MAX;int moveright=INT_MAX;
            if(CurrRow<rowsize-1)
                movedown =  dp[CurrRow+1][CurrCol];
             if(CurrCol<colsize-1)
                 moveright =   dp[CurrRow][CurrCol+1];
            dp[CurrRow][CurrCol] =  grid[CurrRow][CurrCol] + min(movedown,

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

**SPACE OPTIMIZATION:**

```cpp
int minPathSum(vector<vector<int>>& grid) {
      int rowsize = grid.size();
       int colsize = grid[0].size();
       vector<int> next(colsize,INT_MAX);
       for(int CurrRow=rowsize-1;CurrRow>=0;CurrRow--)
       { vector<int> curr(colsize,INT_MAX);
           for(int CurrCol = colsize-1;CurrCol>=0;CurrCol--)
           {
         if(CurrRow==rowsize-1 && CurrCol==colsize-1)
             curr[CurrCol]=grid[CurrRow][CurrCol];
          else{
           int movedown=INT_MAX;int moveright=INT_MAX;
          if(CurrRow<rowsize-1)
              movedown =  next[CurrCol];
           if(CurrCol<colsize-1)
               moveright =   curr[CurrCol+1];
          curr[CurrCol] =  grid[CurrRow][CurrCol] + min(movedown,mover
           }
           }
        next = curr;
       }
      return next[0];
    }
```

TC : O(M
SC : O(N)

# LECTURE -11 TRIANGLE (DP ON GRIDS)

19 June 2022 18:40

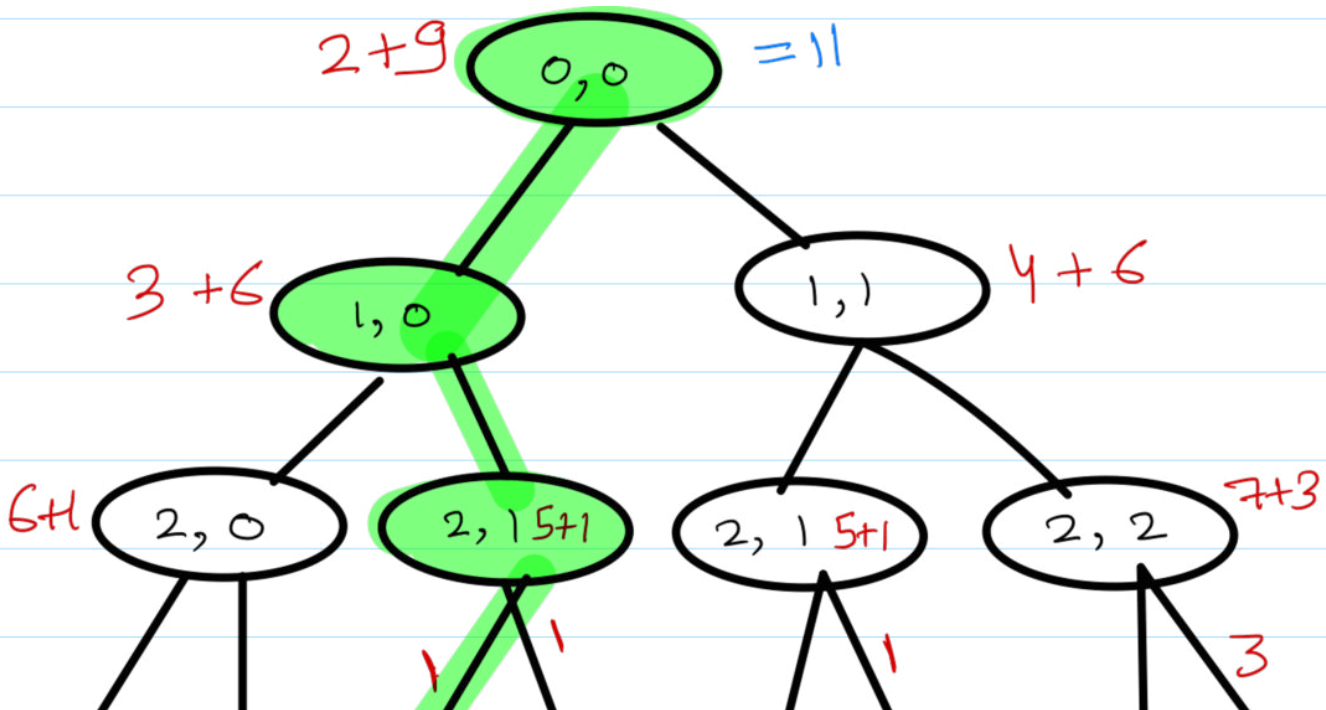
Given a `triangle` array, return *the minimum path sum from top to bottom.*

For each step, you may move to an adjacent number of the row below. More formally, if you are on index `i` on the current row, you may move to either index `i` or index `i + 1` on the next row.

**Example 1:**

```
Input: triangle = [[2],[3,4],[6,5,7],[4,1,8,3]]
Output: 11
Explanation: The triangle looks like:
   2
  3 4
 6 5 7
4 1 8 3
The minimum path sum from top to bottom is 2 + 3 + 5 + 1 = 11 (underlined above).
```

**RECURSIVE SOLUTION:**
**TC : O(2$^{(1+2+3+4+.....N)}$ where N is the number of rows**
**SC : O(N)**



**MEMOIZATION :**
**TC : N*N where N is the number of rows**
**SC : (N*N) + (N)**

```cpp
int findminpath(int CurrRow, int CurrCol,vector<vector<int>>& triangle,
vector<vector<int>> &dp)
    {   int rowsize = triangle.size();
        int colsize = triangle[CurrRow].size();
        if(CurrRow == rowsize-1)
            return triangle[CurrRow][CurrCol];

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

          int onestep = findminpath(CurrRow+1,CurrCol,triangle,dp);
          int twostep = findminpath(CurrRow+1, CurrCol+1,triangle,dp);
        dp[CurrRow][CurrCol]  = triangle[CurrRow][CurrCol] + min(onestep,twostep);

     return dp[CurrRow][CurrCol];
  }

    int minimumTotal(vector<vector<int>>& triangle) {
        int rowsize = triangle.size();
        int colsize = triangle[rowsize-1].size();
        vector<vector<int>> dp(rowsize,vector<int> (colsize,-1));
        return findminpath(0,0,triangle,dp);
    }
```

**Tabulation:**

```cpp
int minimumTotal(vector<vector<int>>& triangle) {
        int rowsize = triangle.size();
        int colsize = triangle[rowsize-1].size();

        vector<vector<int>> dp(rowsize,vector<int> (colsize,0));
        for(int i =0;i<colsize;i++)
            dp[rowsize-1][i] = triangle[rowsize-1][i];

        for(int CurrRow = rowsize-2;CurrRow>=0;CurrRow--)
        { for(int CurrCol = CurrRow;CurrCol>=0;CurrCol--)
            {
            int onestep = dp[CurrRow+1][CurrCol];
            int twostep = dp[CurrRow+1][CurrCol+1];
            dp[CurrRow][CurrCol]  = triangle[CurrRow][CurrCol] + min(onestep,twostep)
```

**SPACE OPTIMIZATION:**

```cpp
int minimumTotal(vector<vector<int>>& triangle) {
        int rowsize = triangle.size();
        int colsize = triangle[rowsize-1].size();

        vector<int> next(rowsize,0);
        vector<vector<int>> dp(rowsize,vector<int> (colsize,0));
        for(int i =0;i<colsize;i++)
            next[i] = triangle[rowsize-1][i];

        for(int CurrRow = rowsize-2;CurrRow>=0;CurrRow--)
        { vector<int> curr(rowsize,0);
            for(int CurrCol = CurrRow;CurrCol>=0;CurrCol--)
            {
            int onestep = next[CurrCol];
            int twostep = next[CurrCol+1];
            curr[CurrCol]  = triangle[CurrRow][CurrCol] + min(onestep,twostep);
            }
          next = curr;
        }
        return next[0];
    }
```

**TC : N*N where N is the number of rows**
**SC : O (N)**

# LECTURE-12 Minimum Falling Path Sum

19 June 2022 21:11

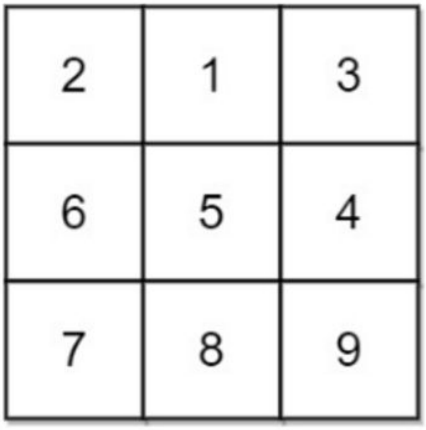
## 931. Minimum Falling Path Sum

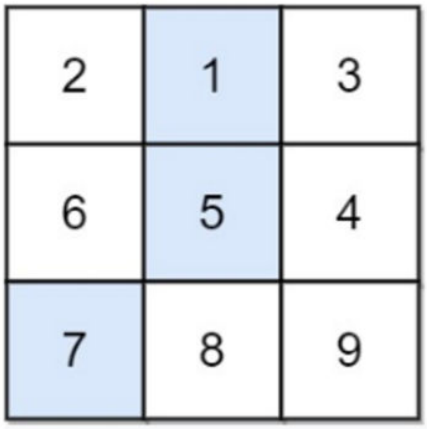
Medium 2493 95 Add to List Share

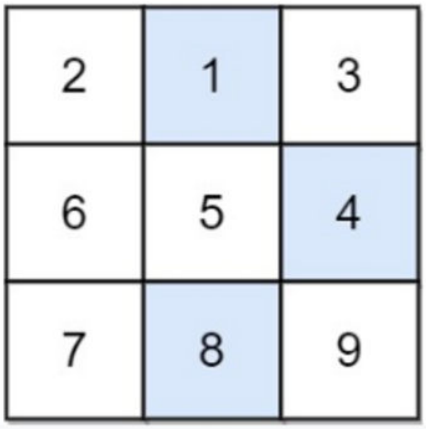
Given an `n x n` array of integers `matrix`, return *the* ***minimum sum*** *of any* ***falling path*** *through* `matrix`.

A **falling path** starts at any element in the first row and chooses the element in the next row that is either directly below or diagonally left/right. Specifically, the next element from position `(row, col)` will be `(row + 1, col - 1)`, `(row + 1, col)`, or `(row + 1, col + 1)`.

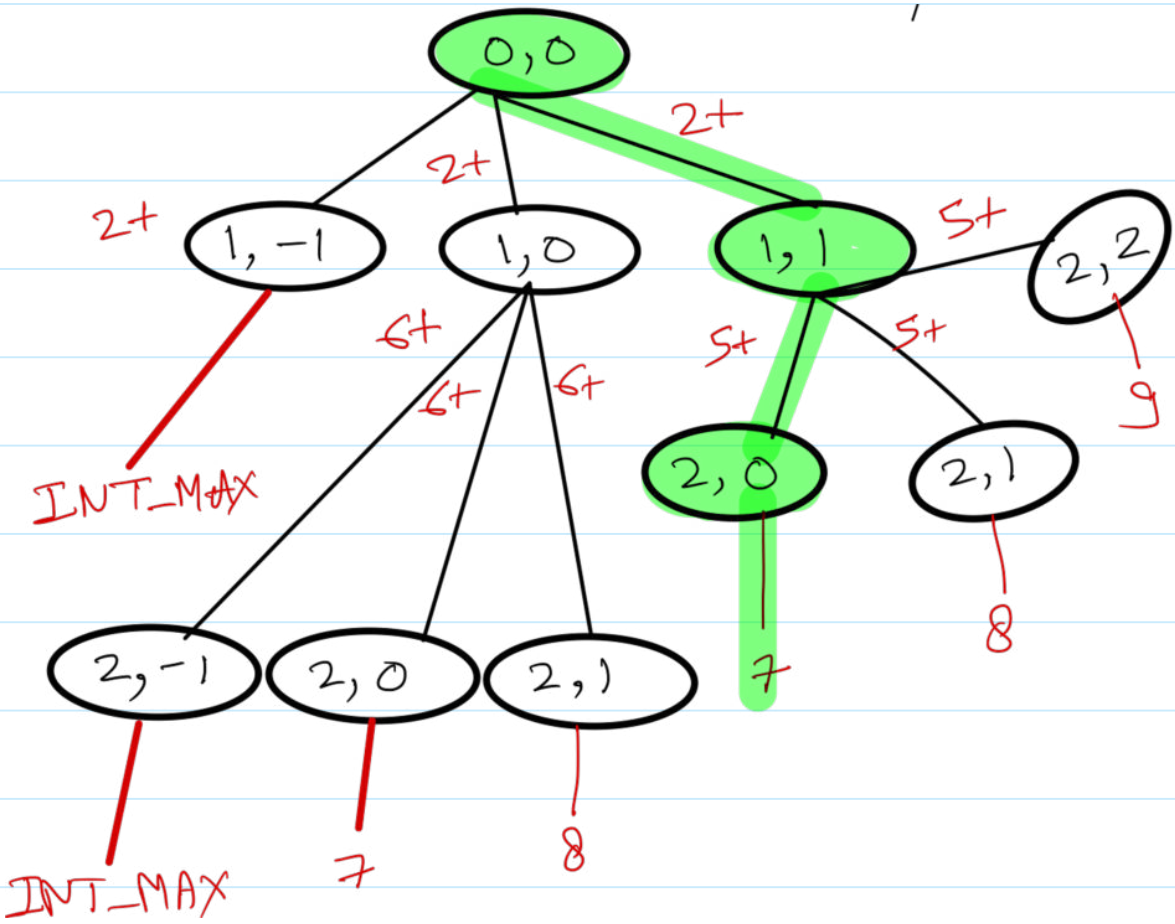
**Example 1:**

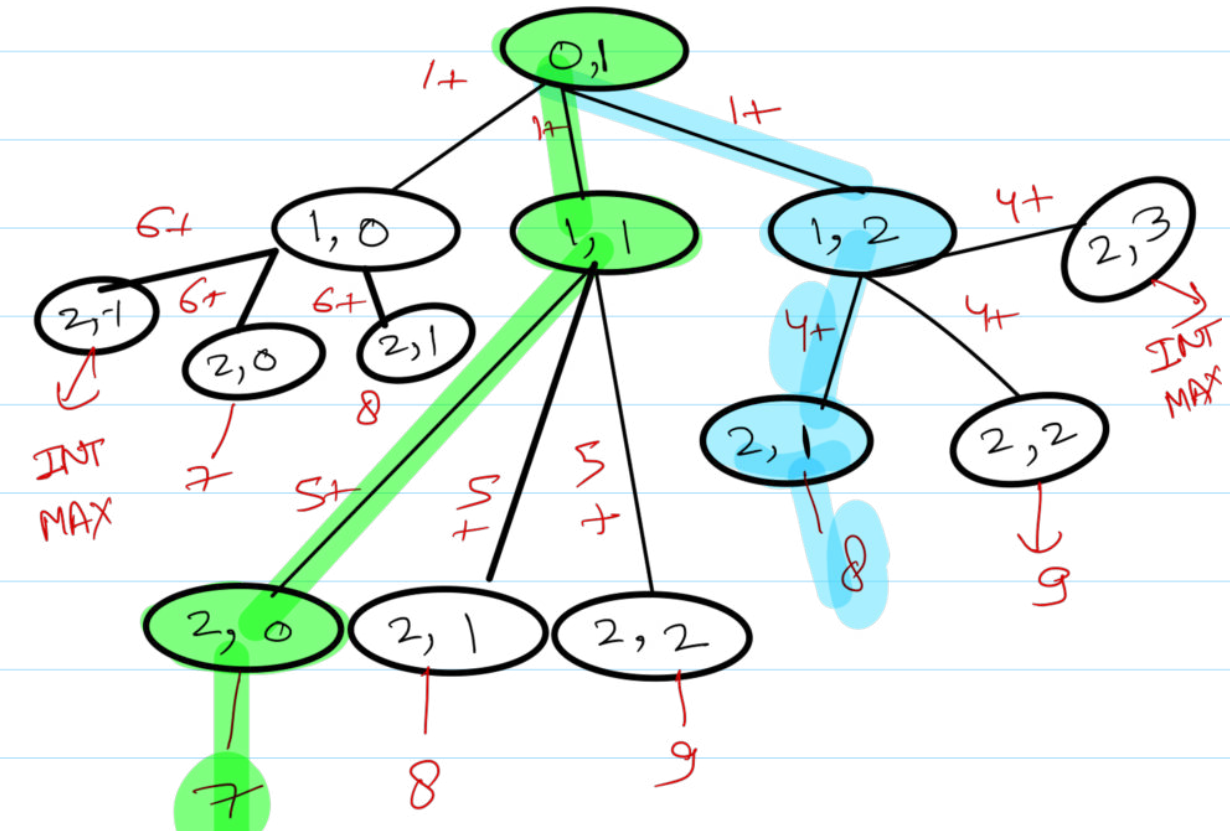


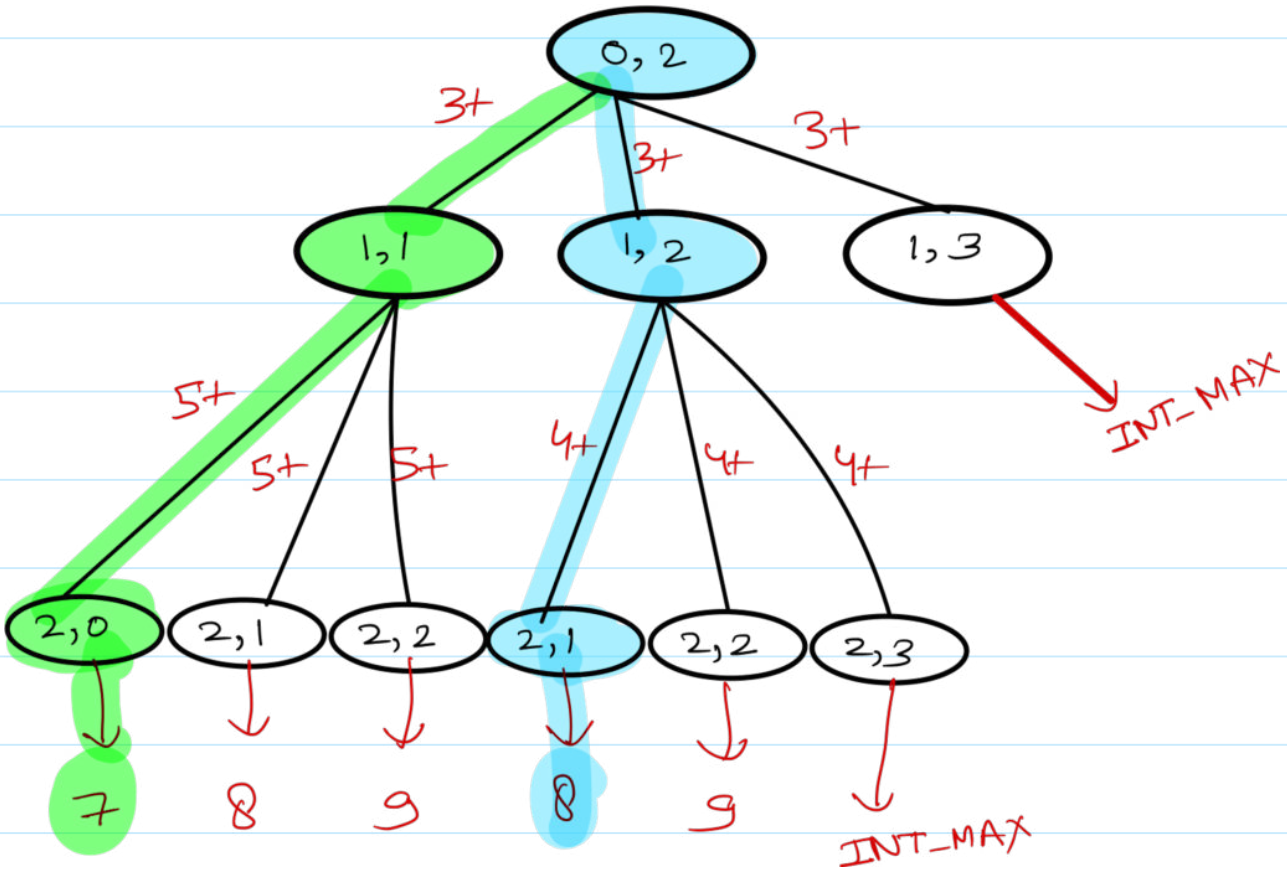




```
Input: matrix = [[2,1,3],[6,5,4],[7,8,9]]
Output: 13
Explanation: There are two falling paths with a minimum sum as shown.
```


0,0
2+
2+
2+
1,-1
1,0
1,1
5+
2,2
6+
6+
6+
5+
5+
9
INT_MAX
2,0
2,1
8
7
2,-1
2,0
2,1
INT_MAX
7
8


0,1
1+
1+
1+
6+
1,0
1,1
1,2
4+
2,3
2,-1
6+
6+
2,0
2,1
4+
4+
INT
MAX
INT
MAX
8
7
5+
5
+
5
+
2,1
2,2
8
9
2,0
2,1
2,2
7
8
9




**RECURSION:**
**TC: $3^N$ Exponential**
**SC: O(N)**

**MEMOIZATION:**

```cpp
int findmin(int CurrRow, int CurrCol,vector<vector<int>>& matrix, vector<vector<int>> &dp)
    {
        int n = matrix[0].size();

         if(CurrRow>n-1|| CurrRow<0 || CurrCol<0 || CurrCol>n-1)
          return INT_MAX;

        if(CurrRow==n-1)
        return matrix[CurrRow][CurrCol];

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

        int down = findmin(CurrRow+1,CurrCol,matrix,dp);
        int downleft = findmin(CurrRow+1,CurrCol-1,matrix,dp);
        int downright = findmin(CurrRow+1,CurrCol+1,matrix,dp);
        return dp[CurrRow][CurrCol] = matrix[CurrRow][CurrCol] + min(down,min(downleft,downright));
    }


    int minFallingPathSum(vector<vector<int>>& matrix) {
     int n = matrix.size();
```

**TABULATION:**

```cpp
int minFallingPathSum(vector<vector<int>>& matrix) {
    int n = matrix.size();
    vector<vector<int>> dp(n,vector<int> (n,0));


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

         for(int CurrRow =n-2;CurrRow>=0;CurrRow--)
       { for(int CurrCol=n-1;CurrCol>=0;CurrCol--)
           {  int down = INT_MAX;
              int downleft = INT_MAX;
              int downright = INT_MAX;
            down = dp[CurrRow+1][CurrCol];
          if(CurrCol>0)
              downleft = dp[CurrRow+1][CurrCol-1];
           if(CurrCol<n-1) downright =dp[CurrRow+1][CurrCol+1];
          dp[CurrRow][CurrCol] = matrix[CurrRow][CurrCol] + min(down,min(downleft,downright));
           }
       }

       int mini = INT_MAX;
       for(int i =0;i<n;i++)
       { mini = min(mini, dp[0][i]);
       }
       return mini;
   }
```

**TC: O(N*M) + O(M), where M is the number of columsn**
**SC : O(N*M)**

**SPACE OPTIMIZATION:**

```cpp
int minFallingPathSum(vector<vector<int>>& matrix) {
    int n = matrix.size();
  vector<int> next(n,0);

   for(int i =0;i<n;i++)
        { next[i]= matrix[n-1][i];
        }

          for(int CurrRow =n-2;CurrRow>=0;CurrRow--)
        { vector<int> curr(n,0);
              for(int CurrCol=n-1;CurrCol>=0;CurrCol--)
           {  int down = INT_MAX; int downleft = INT_MAX; int downright = INT_MAX;

             down = next[CurrCol];
           if(CurrCol>0)
               downleft = next[CurrCol-1];
            if(CurrCol<n-1) downright = next[CurrCol+1];
           curr[CurrCol] = matrix[CurrRow][CurrCol] + min(down,min(downleft,downright));
            }
         next = curr;
        }

        int mini = INT_MAX;
        for(int i =0;i<n;i++)
        { mini = min(mini, next[i]);
        }
        return mini;
    }
```

**TC: O(N*M) + O(M), where M is the number of columns**
**SC : O(M) + O(M)**

# LECTURE -13 Cherry Pickup (3d -DP)

19 June 2022 21:54

## 741. Cherry Pickup

Hard 2734 126 Add to List Share

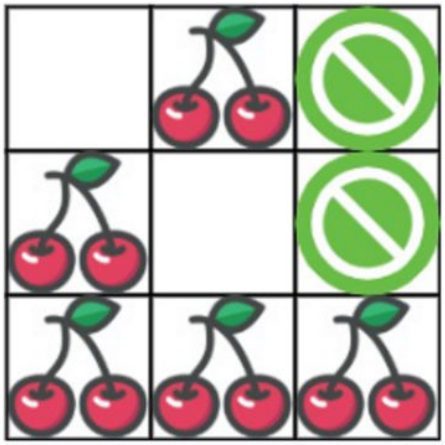
You are given an `n x n` `grid` representing a field of cherries, each cell is one of three possible integers.

- `0` means the cell is empty, so you can pass through,
- `1` means the cell contains a cherry that you can pick up and pass through, or
- `-1` means the cell contains a thorn that blocks your way.

Return *the maximum number of cherries you can collect by following the rules below*:

- Starting at the position `(0, 0)` and reaching `(n - 1, n - 1)` by moving right or down through valid path cells (cells with value `0` or `1`).
- After reaching `(n - 1, n - 1)`, returning to `(0, 0)` by moving left or up through valid path cells.
- When passing through a path cell containing a cherry, you pick it up, and the cell becomes an empty cell `0`.
- If there is no valid path between `(0, 0)` and `(n - 1, n - 1)`, then no cherries can be collected.

**Example 1:**

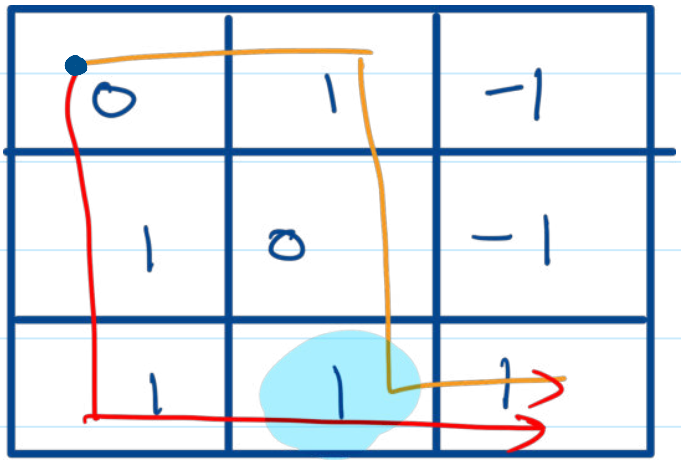


```
Input: grid = [[0,1,-1],[1,0,-1],[1,1,1]]
Output: 5
Explanation: The player started at (0, 0) and went down, down, right right to reach (2, 2).
4 cherries were picked up during this single trip, and the matrix becomes [[0,1,-1],[0,0,-1],[0,0,0]].
Then, the player went left, up, up, left to return home, picking up one more cherry.
The total number of cherries picked up is 5, and this is the maximum possible.
```

**Example 2:**

```
Input: grid = [[1,1,-1],[1,-1,1],[-1,1,1]]
Output: 0
```

**Solution:** We have to go till end and then need to come




→ There can be four possibilities at every index

- Both goes down. (dd)
- Both goes right. (rr)
- One go down one go right. (dr)
- One go right another go down. (rd).

→ If both are at same index, collect cherries only
Ex- at index (2, 1).

→ If both are at different index, collect cherries from Bo
the indexes.

**RECURSIVE SOLUTION:**

```cpp
//RECURSIVE SOLUTION
int cherryPickup(vector<vector<int>>& grid) {
      return max(0, maxcherry(0, 0, 0, 0, grid));
    }

  int maxcherry(int r1, int c1, int r2, int c2, vector<vector<int>>& grid)
  {
      int cherries = 0;
     int n = grid.size();
    if( r1>=n || r2>=n || c1 >=n || c2 >=n || grid[r1][c1]==-1 || grid[r2][c2
      return -100;

   if((r1==n-1) && (c1==grid[0].size()-1))
     return grid[r1][c1];


if(r1==r2 || c1==c2)
      cherries+=grid[r2][c2];
 else
   cherries+= grid[r1][c1]+grid[r2][c2];


   int rr = maxcherry(r1,c1+1,r2,c2+1,grid);
   int rd = maxcherry(r1,c1+1,r2+1,c2,grid);
   int dd = maxcherry(r1+1,c1,r2+1,c2,grid);
   int dr = maxcherry(r1+1,c1,r2,c2+1,grid);

   cherries+= max({rr,rd,dd,dr});
   return cherries;
  }
```

**MEMOIZATION:**

```cpp
int cherryPickup(vector<vector<int>>& grid) {
        int dp[50][50][50][50];
       memset(dp,-1,sizeof(dp));

      return max(0, maxcherry(0,0,0,0, grid,dp));
    }

  int maxcherry(int r1, int c1, int r2,int c2, vector<vector<int>>& grid, int d
[50][50][50])
  {
      int cherries = 0;
     //int c2=r1+c1-r2;
     int n = grid.size();
    if( r1>=n || r2>=n || c1 >=n || c2 >=n || grid[r1][c1]==-1 || grid[r2][c2]
      return -10000;

     if(dp[r1][c1][r2][c2]!=-1){
            return dp[r1][c1][r2][c2];
       }

   if((r1==n-1) && (c1==grid[0].size()-1))
     return grid[r1][c1];


if(r1==r2 || c1==c2)
     cherries+=grid[r1][c1];
  else
    cherries+= grid[r1][c1]+grid[r2][c2];


   int rr = maxcherry(r1,c1+1,r2,c2+1,grid,dp);
   int rd = maxcherry(r1,c1+1,r2+1,c2,grid,dp);
   int dd = maxcherry(r1+1,c1,r2+1,c2,grid,dp);
   int dr = maxcherry(r1+1,c1,r2,c2+1,grid,dp);

   cherries+= max({rr,rd,dd,dr});
    dp[r1][c1][r2][c2]=cherries;
   return cherries;
  }
```

**Further Optimization:**

$r_1, c_1$ $\quad$ $r_2, c_2$

$0, 0$ $\quad$ $0, 0$

Since both are taking equal steps, so

$$r_1 + c_1 = r_2 + c_2$$

OK, Let's understand more.

- Let's say one person move right, c
and another person move down, $r_2$ =

so

$$r_1 + \underline{c_1 + 1} = \underline{r_2 + 1} + c_2$$

$$\downarrow$$

$$r_1 + c_1 + 1 = r_2 + c_2 + 1$$

$$\downarrow$$

$$r_1 + c_1 = r_2 + c_2$$

$$\boxed{c_2 = r_1 + c_1 - r_2}$$

```cpp
int cherryPickup(vector<vector<int>>& grid) {
         int dp[50][50][50];
        memset(dp,-1,sizeof(dp));

      return max(0, maxcherry(0, 0, 0, grid,dp));
    }

 int maxcherry(int r1, int c1, int r2, vector<vector<int>>& grid, int dp[50
 {
     int cherries = 0;
     int c2=r1+c1-r2;
    int n = grid.size();
    if( r1>=n || r2>=n || c1 >=n || c2 >=n || grid[r1][c1]==-1 || grid[r2][c
     return -100000;

    if(dp[r1][c1][r2]!=-1){
           return dp[r1][c1][r2];
       }

  if((r1==n-1) && (c1==grid[0].size()-1))
    return grid[r1][c1];


if(r1==r2 || c1==c2)
     cherries+=grid[r1][c1];
 else
   cherries+= grid[r1][c1]+grid[r2][c2];


  int rr = maxcherry(r1,c1+1,r2,grid,dp);
  int rd = maxcherry(r1,c1+1,r2+1,grid,dp);
  int dd = maxcherry(r1+1,c1,r2+1,grid,dp);
  int dr = maxcherry(r1+1,c1,r2,grid,dp);

  cherries+= max({rr,rd,dd,dr});
dp[r1][c1][r2]=cherries;
  return cherries;
 }
```

LinkedIn/kapilyadav22

# *LECTURE - 14 Subset Sum Equals to Target Identify DP on Subsequences

19 June 2022 22:17

## Subset Sum Equal To K

53 Difficulty: MEDIUM

Avg. time to solve
30 min

Success Rate
65%

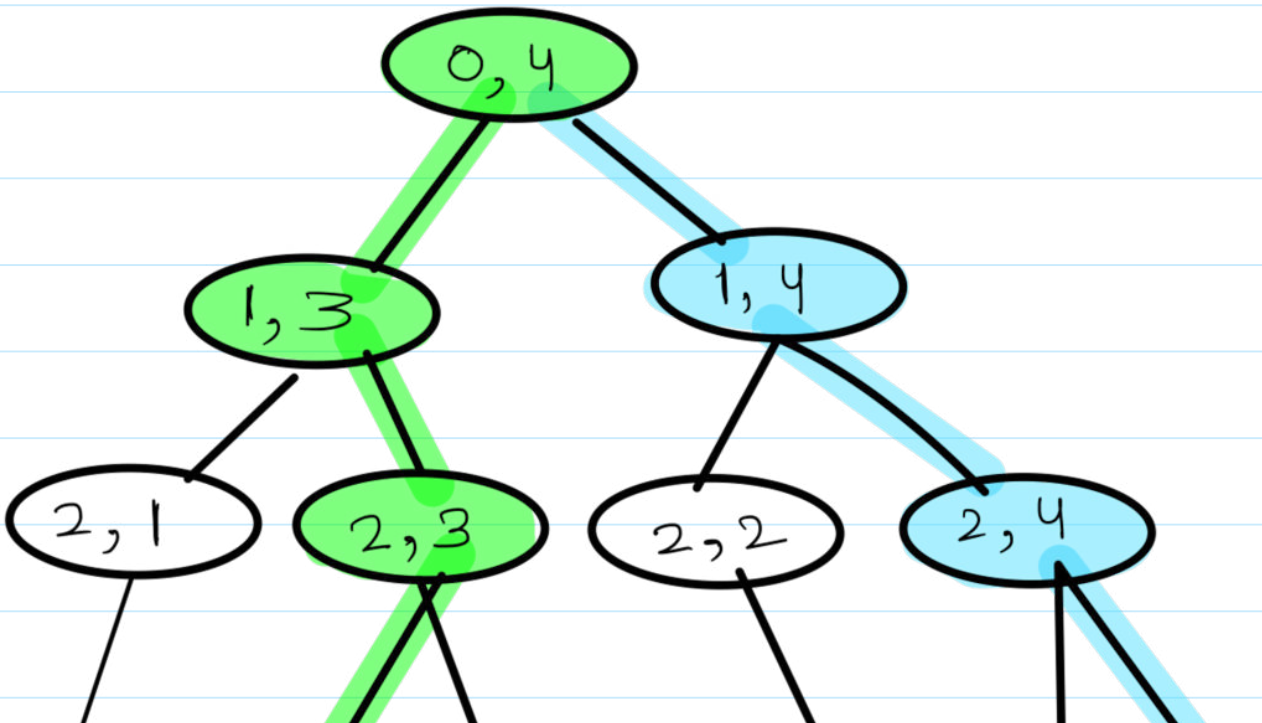
Problem Statement

You are given an array/list 'ARR' of 'N' positive integers and an integer 'K'. Your ta
check if there exists a subset in 'ARR' with a sum equal to 'K'.

Note: Return true if there exists a subset with sum equal to 'K'. Otherwise, return f

For Example :

```
If 'ARR' is {1,2,3,4} and 'K' = 4, then there exists 2 subsets with sum = 4
are {1,3} and {4}. Hence, return true.
```



**MEMOIZATION:**

```cpp
bool findpartition(int currind, vector<int>& nums, vector<vector<int>> & dp
    {
        if(sum==0)
          return 1;
       if(currind>=nums.size())
          return 0;
     if(dp[currind][sum]!=-1)
         return dp[currind][sum];
      bool consider = false;
        if(sum>=nums[currind])
        consider = findpartition(currind+1,nums,dp,sum-nums[currind]);
     bool notconsider = findpartition(currind+1,nums,dp,sum);
    return dp[currind][sum] = consider + notconsider;
    }

bool subsetSumToK(int n, int k, vector<int> &nums) {
        vector<vector<int>> dp(n,vector<int> (k+1,-1));
        return findpartition(0,nums,dp,k);
}

```

**TC: O(N*K)**

**SC : O(N*k) + O(N)**

We are using a recursion stack space(O(N)) and a 2D array ( O(N*K)).

**TABULATION:**

```cpp
bool subsetSumToK(int n, int k, vector<int> &nums) {
        vector<vector<bool>> dp(n,vector<bool> (k+1,false))
        for(int i=0;i<n;i++)
        {   dp[i][0] = true; }
    if(nums[n-1]<=k)
        dp[n-1][nums[n-1]] = true;
        for(int currind=n-2;currind>=0;currind--)
        {  for(int target=k;target>=1;target--)
```

**TC: O(N*K)**
**SC : O(N*k)**

- The last row dp[n-1][] indicates that only the last element of the array is considered, ther target value equal to arr[n-1], only cell with that target will be true, so explicitly set dp[n- =true, ( dp[n-1][nums[n-1]] means that we are considering the last element of the array equal to the last element itself). Please note that it can happen that arr[n-1]>target, so w if(arr[n-1]<=target) then set dp[n-1][nums[n-1]] = true.

**SPACE OPTIMIZATION:**

```cpp
bool subsetSumToK(int n, int k, vector<int> &nums) {
        vector<bool> next(k+1,0);
        next[0]=1;
        if(nums[n-1]<=k)
        next[nums[n-1]] = true;
        for(int currind=n-2;currind>=0;currind--)
        {  vector<bool> curr(k+1,0);
            for(int target=k;target>=0;target--)
            { bool consider = false;
                if(target>=nums[currind])
                consider = next[target-nums[currind]];
                bool notconsider = next[target];
                curr[target] = consider + notconsider;
            }
          next = curr;
        }
        return next[k];
}
```

```cpp
bool subsetSumToK(int n, int k, vector<int> &nums) {
          vector<bool> next(k+1,0);
        next[0]=1;
       if(nums[n-1]<=k)
        next[nums[n-1]] = true;
        for(int currind=n-2;currind>=0;currind--)
        {  vector<bool> curr(k+1,0);
           curr[0]=1;
            for(int target=k;target>=1;target--)
            { bool consider = false;
                if(target>=nums[currind])
                consider = next[target-nums[currind]];
                bool notconsider = next[target];
                curr[target] = consider + notconsider;
            }
         next = curr;
       }
        return next[k];
}
```

LinkedIn/kapilyadav22

# LECTURE - 15 Partition Equal Subset Sum | DP on Subsequences

20 June 2022 09:46

## 416. Partition Equal Subset Sum

Medium 7844 124 Add to List Share

Given a **non-empty** array `nums` containing **only positive integers**, find if the array can be partiti
subsets such that the sum of elements in both subsets is equal.

**Example 1:**

```
Input: nums = [1,5,11,5]
Output: true
Explanation: The array can be partitioned as [1, 5, 5] and [11].
```

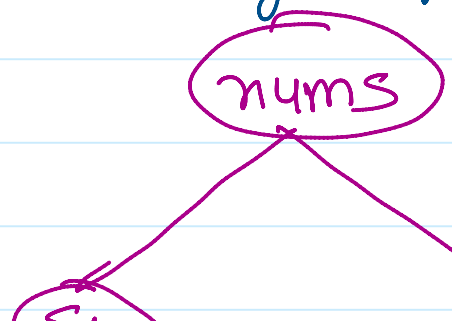
**Example 2:**

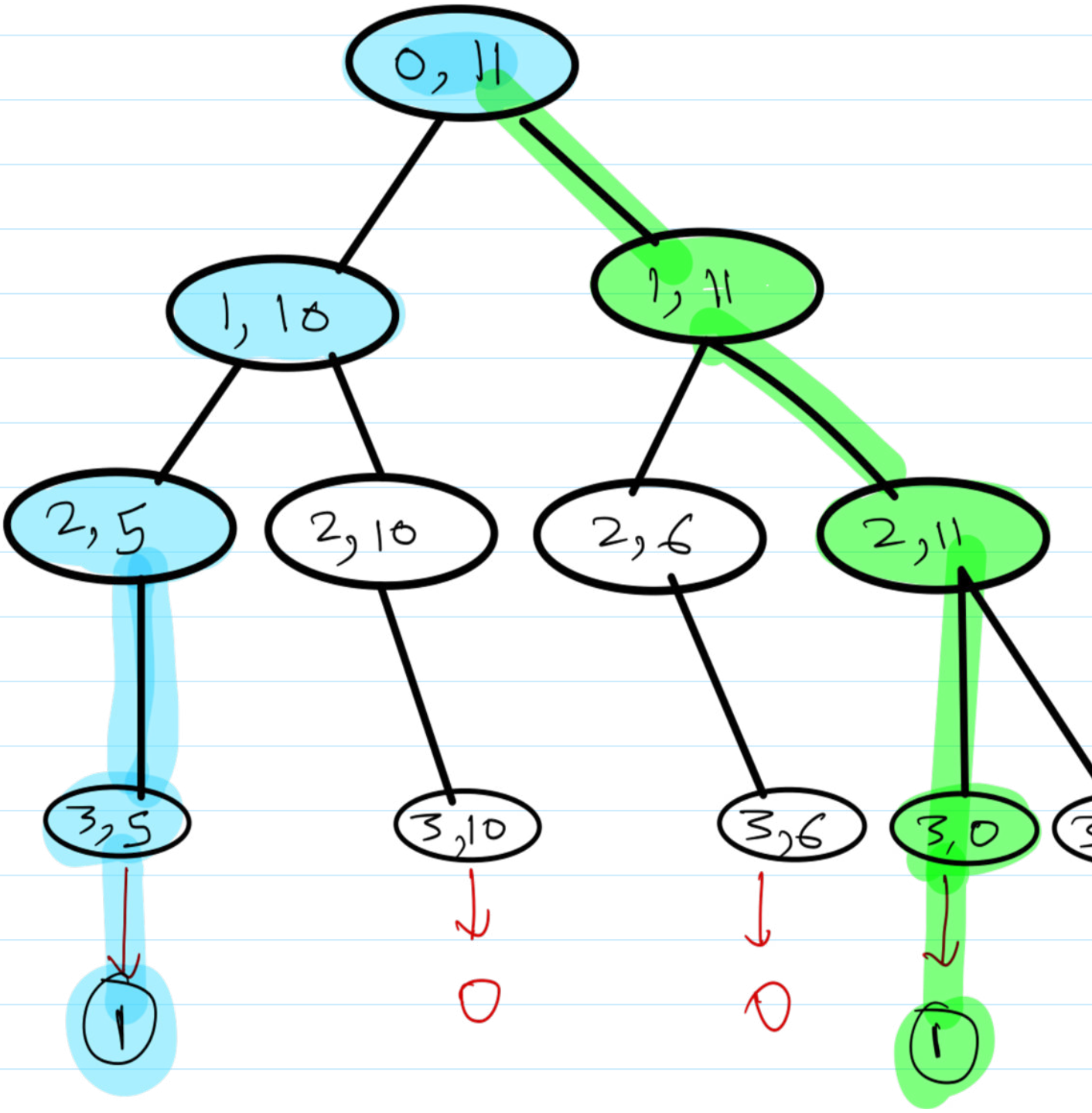
```
Input: nums = [1,2,3,5]
Output: false
Explanation: The array cannot be partitioned into equal sum subsets.
```

**Same as Lecture 14. We just need to check if total sum is odd, we cannot partition it, return false. Else check for the sum of totalsum/2.**

ex → {1, 5, 11, 5}

totalsum = 22, so, we can divide it into subsets having equal sum.




0, 11
1, 10
1, 11
2, 5
2, 10
2, 6
2, 11
3, 5
3, 10
3, 6
3, 0
1
0
0
1

**MEMOIZATION:**

```cpp
bool findpartition(int currind, vector<int>& nums, vector<vecto
& dp,int sum)
    {
        if(sum==0)
          return 1;
       if(currind>=nums.size())
          return 0;
     if(dp[currind][sum]!=-1)
         return dp[currind][sum];
      bool consider = false;
        if(sum>=nums[currind])
        consider = findpartition(currind+1,nums,dp,sum-nums[cur
     bool notconsider = findpartition(currind+1,nums,dp,sum);
    return dp[currind][sum] = consider + notconsider;
    }

bool canPartition(vector<int> &nums, int n)
{  int sum=0;
        for(int i =0;i<n;i++)
            sum+=nums[i];
        if(sum%2!=0)
            return false;
        vector<vector<int>> dp(n,vector<int> (sum/2+1,-1));
        return findpartition(0,nums,dp,sum/2);
}
```

**TC: O(N*K)**

**SC : O(N*k) + O(N)**

We are using a recursion stack space(O(N)) and a 2D array ( O(N*K)).
Where K = sum/2

**TABULATION:**

```cpp
bool canPartition(vector<int>& nums) {
        int n = nums.size();
        int sum=0;
        for(int i =0;i<n;i++)
            sum+=nums[i];
        if(sum%2!=0)
            return false;
        vector<vector<bool>> dp(n,vector<bool> (sum/2+1,0));
        for(int i=0;i<n;i++)
            {       dp[i][0] = true;
            }
     if(nums[n-1]<=sum/2)
        dp[n-1][nums[n-1]] = true;
        for(int currind=n-2;currind>=0;currind--)
        {  for(int target=sum/2;target>=0;target--)
            { bool consider = false;
                if(target>=nums[currind])
                consider = dp[currind+1][target-nums[currind
                bool notconsider = dp[currind+1][target];
                dp[currind][target] = consider + notconsider
            }
       }
        return dp[0][sum/2];
        }
```

**TC: O(N*K)**
**SC : O(N*k)**
**Where k = sum/2**

**SPACE OPTIMIZATION:**

```cpp
bool canPartition(vector<int>& nums) {
        int n = nums.size();
        int sum=0;
        for(int i =0;i<n;i++)
            sum+=nums[i];
        if(sum%2!=0)
            return false;

        vector<bool> next(sum/2+1,0);
        next[0]=1;
       if(nums[n-1]<=sum/2)
        next[nums[n-1]] = true;
        for(int currind=n-2;currind>=0;currind--)
        {  vector<bool> curr(sum/2+1,0);
            for(int target=sum/2;target>=0;target--)
            { bool consider = false;
                if(target>=nums[currind])
                consider = next[target-nums[currind]];
                bool notconsider = next[target];
                curr[target] = consider + notconsider;
            }
         next = curr;
       }
        return next[sum/2];
    }
```

**Time Complexity: O(N*K)**

**Space Complexity: O(K)**

**Where k = sum/2**

# Lecture - 16. Partition A Set Into Two Subsets With Minimum Absolute Sum Difference

20 June 2022 13:31

**Problem Statement** Sug

You are given an array containing N non-negative integers. Your task is to partition array into two subsets such that the absolute difference between subset sums is mi

You just need to find the minimum absolute difference considering any valid division array elements.

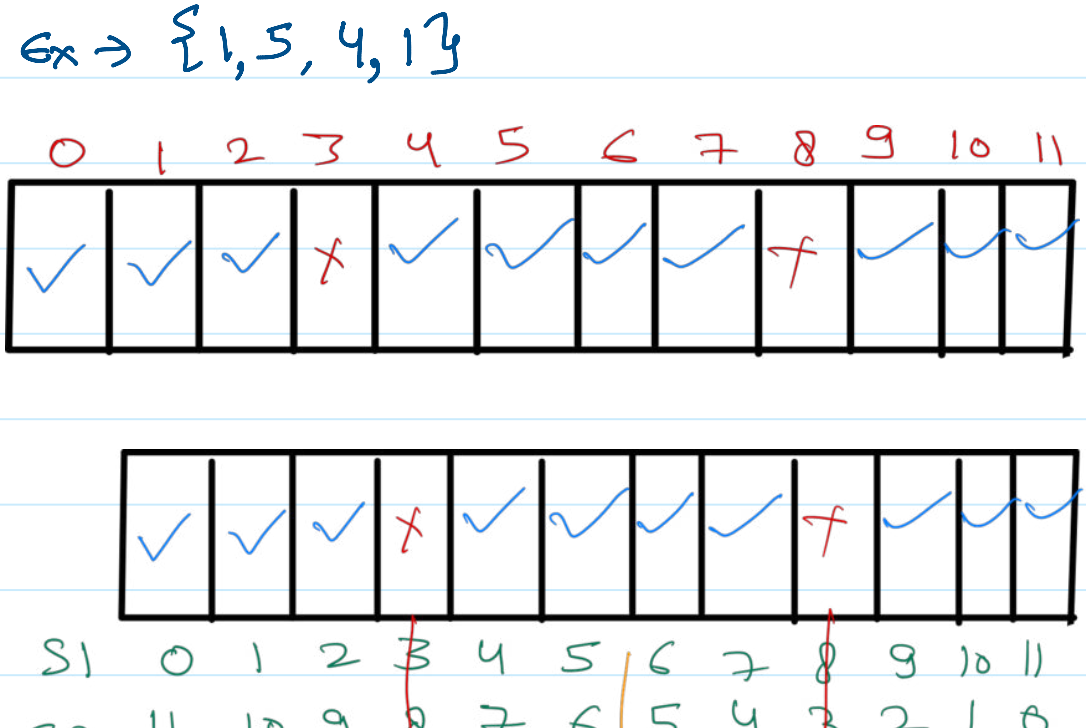
Note:

1. Each element of the array should belong to exactly one of the subset.

2. Subsets need not be contiguous always. For example, for the array : {1,2,3 some of the possible divisions are a) {1,2} and {3} b) {1,3} and {2}.

3. Subset-sum is the sum of all the elements in that subset.

From subset sum problem (Lecture - 14), we can derive if every possible target between 1 and k is possible or



```cpp
int minSubsetSumDifference(vector<int>& nums, int n)
{   int k=0;
    for(int i =0;i<n;i++)
    {   k+=nums[i];
    }
     vector<vector<bool>> dp(n,vector<bool> (k+1,false));
        for(int i=0;i<n;i++)
        {   dp[i][0] = true; }
    if(nums[n-1]<=k)
        dp[n-1][nums[n-1]] = true;
        for(int currind=n-2;currind>=0;currind--)
        {  for(int target=k;target>=1;target--)
            { bool consider = false;
                if(target>=nums[currind])
                consider = dp[currind+1][target-nums[currin
                bool notconsider = dp[currind+1][target];
                dp[currind][target] = consider + notconside
            }
       }

 int mini = 1e9;
 for(int s1=0;s1<=k;s1++)
 {   if(dp[0][s1]==true)
     {  int s2 = k-s1;
     mini = min(mini,abs(s2-s1));
     }
 }
return mini;
 }
```

TC: O(N*K) + O(K)
SC: O(N*K)

**SPACE OPTIMIZATION:**

```
int minSubsetSumDifference(vector<int>& nums, int
{   int k=0;
    for(int i =0;i<n;i++)
    {   k+=nums[i];
    }
      vector<bool> next(k+1,0);

       next[0]=1;
       if(nums[n-1]<=k)
        next[nums[n-1]] = true;
        for(int currind=n-2;currind>=0;currind--
        {  vector<bool> curr(k+1,0);
            for(int target=k;target>=0;target--)
            { bool consider = false;
                if(target>=nums[currind])
                consider = next[target-nums[curri
                bool notconsider = next[target];
                curr[target] = consider + notcons
            }
          next = curr;
        }
 int mini = 1e9;
 for(int s1=0;s1<=k/2;s1++)
 {   if(next[s1]==true)
     {  int s2 = k-s1;
     mini = min(mini,abs(s2-s1));
```

# LECTURE-17 Counts Subsets with Sum K | Dp on Subsequences

20 June 2022 15:20

- Same as Lecture -14, we just need to print the count, everything remain same.

## Number Of Subsets

25 Difficulty: MEDIUM

Contributed By
KAPEESH UPADHYAY | Level 1

**Problem Statement** Suggest Edit

You are given an array (0-based indexing) of positive integers and you have to tell how many different ways of selecting the elements from the array are there such that the sum of chosen elements is equal to the target number “tar”.

**Note:**

```
Two ways are considered different if sets of indexes of elements chosen by these
ways are different.

Input is given such that the answer will fit in a 32-bit integer.
```

**For Example:**

```
If N = 4 and tar = 3 and the array elements are [1, 2, 2, 3], then the number of
possible ways are:
{1, 2}
{3}
{1, 2}
Hence the output will be 3.
```

**Memoization:**

```cpp
int countways(int currind, int tar, vector<int>& nums,
vector<vector<int>>& dp)
{   if(tar==0)
      return 1;

  if(currind==nums.size()-1)
      return (nums[currind]==tar);
  if(currind>nums.size()-1 )
    return 0;
  if(dp[currind][tar]!=-1)
      return dp[currind][tar];
   int pick = 0;
 if(nums[currind]<=tar)
     pick = countways(currind+1,tar-
nums[currind],nums,dp);
     int notpick = countways(currind+1,tar,nums,dp);
 return dp[currind][tar] = pick+notpick;
 }

int findWays(vector<int> &nums, int tar)
{  int n= nums.size();
 vector<vector<int>> dp(n,vector<int> (tar+1,-1));
 return countways(0,tar,nums,dp);
}
```

**TC: O(N*K)**

**SC : O(N*k) + O(N)**

We are using a recursion stack space(O(N)) and a 2D array ( O(N*K)).

**Tabulation:**

```cpp
int findWays(vector<int> &nums, int tar)
{  int n= nums.size();
 vector<vector<int>> dp(n,vector<int> (tar+1,0));
 for(int i =0;i<n;i++)
     dp[i][0]=1;
   if(nums[n-1]<=tar)
       dp[n-1][nums[n-1]] = 1;
  for(int currind=n-2;currind>=0;currind--)
  {   for(int target=tar;target>=1;target--)
          {  int pick = 0;
           if(nums[currind]<=target)
             pick = dp[currind+1][target-nums[currind]];
             int notpick = dp[currind+1][target];
             dp[currind][target] = pick+notpick;
        }
  }return dp[0][tar];
}
```

**TC: O(N*K)**
**SC : O(N*k)**

**Space Optimization:**

```cpp
int findWays(vector<int> &nums, int k)
{  int n= nums.size();
 vector<int> next(k+1,0);
         next[0]=1;
        if(nums[n-1]<=k)
         next[nums[n-1]] = 1;
         for(int currind=n-2;currind>=0;currind--)
         {   vector<int> curr(k+1,0);
             curr[0]=1;
             for(int target=k;target>=1;target--)
             { int consider = 0;
                 if(target>=nums[currind])
                 consider = next[target-nums[currind]];
                 int notconsider = next[target];
                 curr[target] = consider + notconsider;
             }
          next = curr;
        }
         return next[k];
}
```

**Time Complexity: O(N*K)**

**Space Complexity: O(K)**

```cpp
int findWays(vector<int> &nums, int k)
{  int n= nums.size();
 vector<int> next(k+1,0);
         next[0]=1;
        if(nums[n-1]<=k)
         next[nums[n-1]] = 1;
         for(int currind=n-2;currind>=0;currind--)
         {   vector<int> curr(k+1,0);
             //curr[0]=1;
             for(int target=k;target>=0;target--)
             { int consider = 0;
                 if(target>=nums[currind])
                 consider = next[target-nums[currind]];
                 int notconsider = next[target];
                 curr[target] = consider + notconsider;
             }
          next = curr;
        }
         return next[k];
}
```

**Time Complexity: O(N*K)**

**Space Complexity: O(K)**

**Edge case:**

Curr: { 1,0,0 } our code will give the answer as 1 .
But ideally it should be 4….
{1,0}
{1,0}
{1,0,0}
{1}
Our code has passed in codestudio, because the constraint in the question was given as 1<=nums[i]<=1000, so modify our base cases.

**Memoization:**

```
int countways(int currind, int tar, vector<int>& nums,
vector<vector<int>>& dp)
{  int n=nums.size();
    if(currind==n-1)
      {  if(tar==0 && nums[n-1]==0)
          return 2;
          if(tar==0 || tar==nums[n-1]) return 1;
            return 0;
      }

 if(dp[currind][tar]!=-1)
      return dp[currind][tar];
   int pick = 0;
 if(nums[currind]<=tar)
     pick = countways(currind+1,tar-nums[currind],nums,dp);
     int notpick = countways(currind+1,tar,nums,dp);
 return dp[currind][tar] = pick+notpick;
 }

int findWays(vector<int> &nums, int tar)
{  int n= nums.size();
 vector<vector<int>> dp(n,vector<int> (tar+1,-1));
 return countways(0,tar,nums,dp);
}
```

If in constraint, 0<=N, then, we have to modify the base conditions and have to run the target loop till >=0, because we are now computing the values at target==0, if target ==0, there can be case, nums[i] might be 0, so it can also contribute to the subset. And there can be multiple 0 in nums array. Like
Ex: 1 1 0 0 0 0 0 , and sum = 0, so subset can be {1} , {1,0}, {1,0}, {1,0},{1,0},{1,0}, {1,0,0},{1,0,0},{1,0,0},{1,0,0},{1,0,0}

2 + 2(5C1 + 5C2 + 5C3 + 5C4) = 2 + 2 (5 + 10 + 10 +5) = 2 + 2(30) = 2+62 = 64

OUTPUT : 64

**Tabulation:**

```cpp
int findWays(vector<int> &nums, int tar)
{  int n= nums.size();
 vector<vector<int>> dp(n,vector<int> (tar+1,0));
 if(nums[n-1]==0)
     dp[n-1][0]=2;
     else dp[n-1][0]=1;
 if(nums[n-1]!=0 && nums[n-1]<=tar)
     dp[n-1][nums[n-1]]=1;

  for(int currind=n-2;currind>=0;currind--)
  {   for(int target=tar;target>=0;target--)
          {  int pick = 0;
           if(nums[currind]<=target)
             pick = dp[currind+1][target-nums[currind]];
             int notpick = dp[currind+1][target];
             dp[currind][target] = pick+notpick;
        }
  }return dp[0][tar];
}
```

**Space Optimization:**

```cpp
int findWays(vector<int> &nums, int tar)
{  int n= nums.size();
 vector<int> next(tar+1,0);
  if(nums[n-1]==0)
     next[0]=2;
     else next[0]=1;
 if(nums[n-1]!=0 && nums[n-1]<=tar)
     next[nums[n-1]]=1;
  for(int currind=n-2;currind>=0;currind--)
  {  for(int target=tar;target>=0;target--)
          {  int pick = 0;
           if(nums[currind]<=target)
             pick = next[target-nums[currind]]
             int notpick = next[target];
             next[target] = pick+notpick;
        }
  }
 return next[tar];
}
```

# LECTURE - 18 Count Partitions With Given Difference | Dp on Subsequences

20 June 2022 19:13

42

## Partitions With Given Difference

Difficulty: MEDIUM

Contributed By
Abhishek Bansal | Level 1

Problem Statement Suggest Edit

Given an array 'ARR', partition it into two subsets (possibly empty) such that their union is the original array. Let the sum of the elements of these two subsets be 'S1' and 'S2'.

Given a difference 'D', count the number of partitions in which 'S1' is greater than or equal to 'S2' and the difference between 'S1' and 'S2' is equal to 'D'. Since the answer may be too large, return it modulo '10^9 + 7'.

If 'Pi_Sj' denotes the Subset 'j' for Partition 'i'. Then, two partitions P1 and P2 are considered different if:

```
1) P1_S1 != P2_S1 i.e, at least one of the elements of P1_S1 is different from
P2_S2.
2) P1_S1 == P2_S2, but the indices set represented by P1_S1 is not equal to the
indices set of P2_S2. Here, the indices set of P1_S1 is formed by taking the
indices of the elements from which the subset is formed.
Refer to the example below for clarification.
```

Note that the sum of the elements of an empty subset is 0.

For Example :

```
If N = 4, D = 3, ARR = {5, 2, 5, 1}
There are only two possible partitions of this array.
Partition 1: {5, 2, 1}, {5}. The subset difference between subset sum is: (5 + 2 +
1) - (5) = 3
Partition 2: {5, 2, 1}, {5}. The subset difference between subset sum is: (5 + 2 +
1) - (5) = 3
```

$S_1 > S_2$,

$S_1 = totalsum - S_2$

$S_1 - S_2 = D$

$totalsum - S_2 - S_2 = D$

$2S_2 = totalsum - D$

$$S_2 = \frac{totalsum - D}{2}$$

→ The question broke down to count s whose sum is equal to $\left(\frac{totalsum - D}{2}\right)$

ex = N = 4, D = 3, arr = {5, 2, 5, 1}.

P1 = {5, 2, 1}, P2 = {5}

P1 = {5, 2, 1}, P2 = {5}

totalsum = 5 + 2 + 5 + 1 = 13

$$S_2 = \frac{13 - 3}{2} = 5$$

→ So find the count of subsets having sum equal to $S_2$, i.e. $\left(\frac{totalsum - D}{2}\right)$.

→ There will be 2 edge cases, since nums[i] is in integer, $S_2$ will be in int, so totalsum-D, should be even.

→ totalsum >= d, then only we can find count of subsets

**Memoization:**

```cpp
int mod =(int)(1e9 + 7);
int countways(int currind, int tar, vector<int>& nums,
vector<vector<int>>& dp)
{    int n=nums.size();
     if(currind==n-1)
       {  if(tar==0 && nums[n-1]==0)
            return 2;
            if(tar==0 || tar==nums[n-1]) return 1;
              return 0;
       }
   if(dp[currind][tar]!=-1)
       return dp[currind][tar];
    int pick = 0;
 if(nums[currind]<=tar)
     pick = countways(currind+1,tar-nums[currind],nums,dp);
     int notpick = countways(currind+1,tar,nums,dp);
 return dp[currind][tar] = (pick+notpick)%mod;
 }

int countPartitions(int n, int d, vector<int> &arr) {
    int totalsum = 0;
    for(int i=0;i<n;i++)
        totalsum+=arr[i];
    if((totalsum-d)<0 || (totalsum-d)%2!=0)
        return 0;
    int tar = (totalsum-d)/2;
     vector<vector<int>> dp(n,vector<int> (tar+1,-1));
     return countways(0,tar,arr,dp);
}
```

**Time Complexity: O(N*K) + O(N)**

**Space Complexity: O(N*K) + O(N), where K = totalsum-d/2**

**Tabulation:**

```cpp
int mod =(int)(1e9 + 7);
int countways(int tar, vector<int>& nums)
{    int n= nums.size();
 vector<vector<int>> dp(n,vector<int> (tar+1,0));
 if(nums[n-1]==0)
     dp[n-1][0]=2;
     else dp[n-1][0]=1;
 if(nums[n-1]!=0 && nums[n-1]<=tar)
     dp[n-1][nums[n-1]]=1;

  for(int currind=n-2;currind>=0;currind--)
  {   for(int target=tar;target>=0;target--)
          {  int pick = 0;
           if(nums[currind]<=target)
             pick = dp[currind+1][target-nums[currind]];
             int notpick = dp[currind+1][target];
             dp[currind][target] = (pick+notpick)%mod;
        }
  }return dp[0][tar];
 }

int countPartitions(int n, int d, vector<int> &arr) {
    int totalsum = 0;
    for(int i=0;i<n;i++)
        totalsum+=arr[i];
    if((totalsum-d)<0 || (totalsum-d)%2!=0)
        return 0;
    int tar = (totalsum-d)/2;
     return countways(tar,arr);
}
```



**Time Complexity: O(N*K) + O(N)**

**Space Optimization:**

**Time Complexity: O(N*K) + O(N)**

**Space Complexity: O(K), where K = totalsum-d/2**

```cpp
int mod =(int)(1e9 + 7);
int countways(int tar, vector<int>& nums)
{   int n= nums.size();
 vector<int> next(tar+1,0);
  if(nums[n-1]==0)
     next[0]=2;
     else next[0]=1;
 if(nums[n-1]!=0 && nums[n-1]<=tar)
     next[nums[n-1]]=1;
  for(int currind=n-2;currind>=0;currind--)
  {    vector<int> curr(tar+1,0);
      for(int target=tar;target>=0;target--)
          {  int pick = 0;
           if(nums[currind]<=target)
             pick = next[target-nums[currind]];
             int notpick = next[target];
             curr[target] = (pick+notpick)%mod;
        }
   next = curr;
  }
 return next[tar];
 }

int countPartitions(int n, int d, vector<int> &arr)
{
    int totalsum = 0;
    for(int i=0;i<n;i++)
        totalsum+=arr[i];
    if((totalsum-d)<0 || (totalsum-d)%2!=0)
```

# *LECTURE - 19 0/1 KNAPSACK ( VVI)

20 June 2022 23:03

## 0 - 1 Knapsack Problem

**Medium** Accuracy: 47.21% Submissions: 100k+ Points: 4

You are given weights and values of **N** items, put these items in a knapsack of capacit
the maximum total value in the knapsack. Note that we have only **one quantity of eac**
In other words, given two integer arrays **val[0..N-1]** and **wt[0..N-1]** which represent
and weights associated with **N** items respectively. Also given an integer W which repr
knapsack capacity, find out the maximum value subset of **val[]** such that sum of the w
this subset is smaller than or equal to **W.** You cannot break an item, **either pick the co**
**item or dont pick it (0-1 property)**.

**Example 1:**

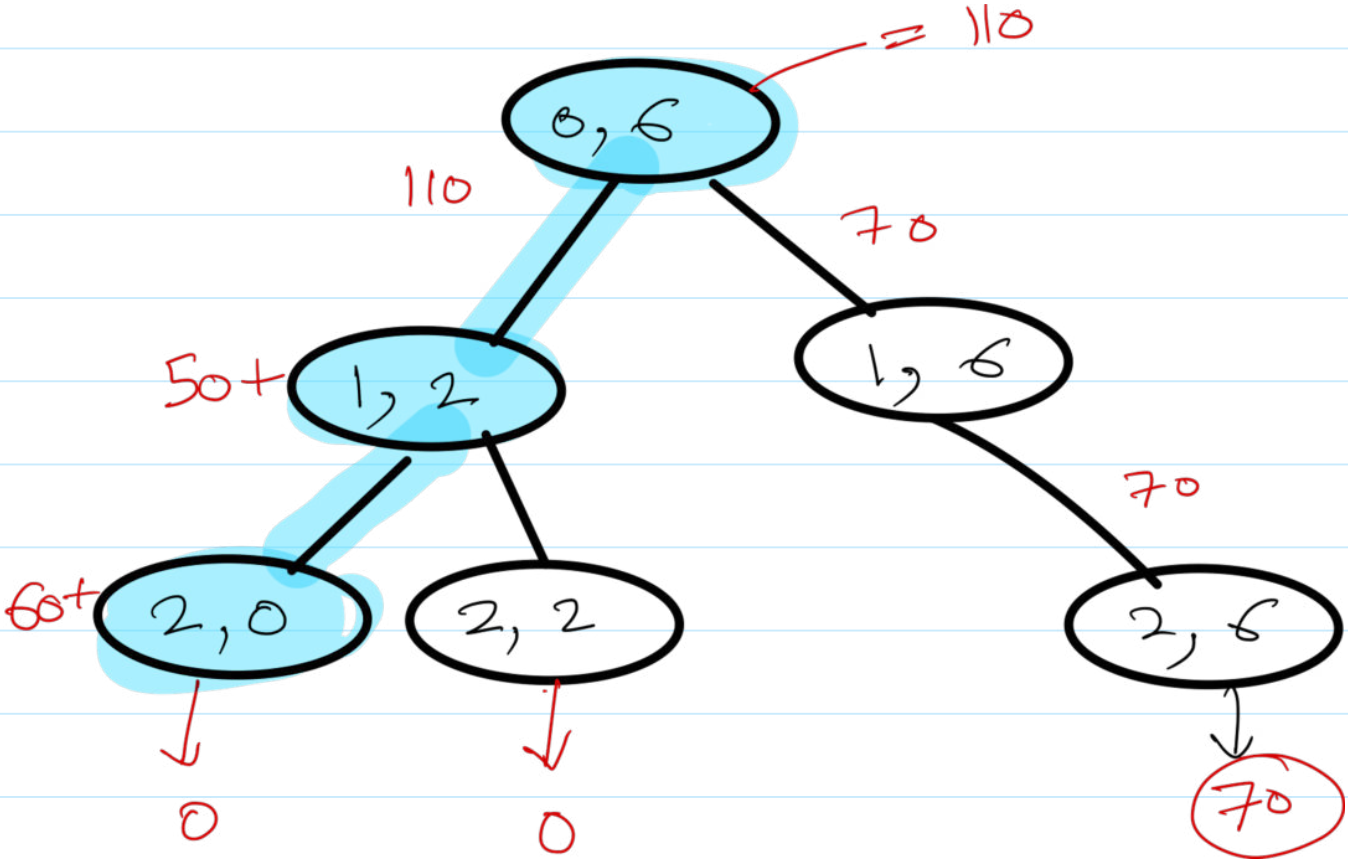
```
Input:
N = 3
W = 4
values[] = {1,2,3}
weight[] = {4,5,1}
Output: 3
```

Can we use greedy ?

Ex → weight → 4 2 6
value → 50 60 70
W = 6.

so greedy approach will take item 3, i.e. having weight 6 and value 70.

→ so, we have to explore all the possibilitie
optimal solution will be → 50 + 60 = 11




→ These can be overlapping subproblems.

**MEMOIZATION:**

```
int findknapsack(int currind, int capacity,vector<int>& wt,vector<int>& profit, int n,
vector<vector<int>>& dp)
    {   if(capacity==0)
            return 0;
        if(currind>n-1)
            return 0;

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

         int notconsider = findknapsack(currind+1,capacity,wt,profit,n,dp);
         int consider = INT_MIN;
         if(wt[currind]<=capacity)
```

**Tabulation:**

```cpp
int knapsack(vector<int>& wt, vector<int>& profit, int n, int W)
{  vector<vector<int>> dp(n,vector<int> (W+1,0));

   for(int i=wt[n-1];i<=W;i++)
   {  dp[n-1][i]= profit[n-1];
   }
 for(int currind=n-2;currind>=0;currind--)
 {  for(int capacity=W;capacity>=0;capacity--)
     {   int notconsider = 0 + dp[currind+1][capacity];
         int consider = INT_MIN;
         if(wt[currind]<=capacity)
         consider = profit[currind] + dp[currind+1][capacity-wt[c
        dp[currind][capacity] = max(consider,notconsider);
   }
 }
   return dp[0][W];
}
```

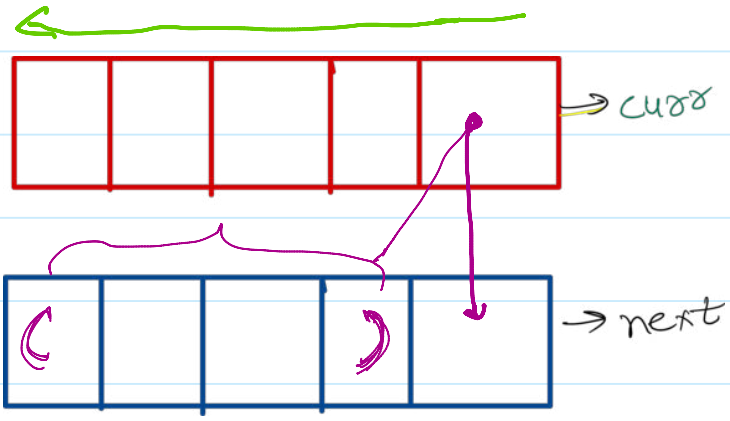
TC : O(N*W)
SC : O(N*W)

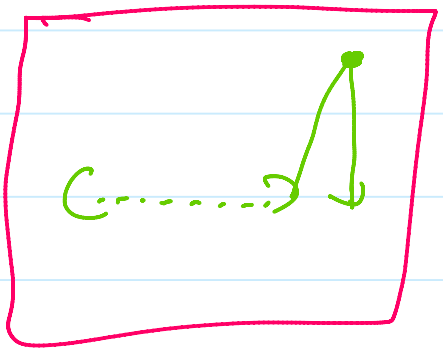
**Space Optimization:**

```cpp
int knapsack(vector<int>& wt, vector<int>& profit, int n, in
{   vector<int> next(W+1,0);
   for(int i=wt[n-1];i<=W;i++)
   {  next[i]= profit[n-1];
   }
 for(int currind=n-2;currind>=0;currind--)
 {  vector<int> curr(W+1,0);
     for(int capacity=W;capacity>=0;capacity--)
     {   int notconsider =next[capacity];
         int consider = INT_MIN;
         if(wt[currind]<=capacity)
```

**Further Space Optimization:** Using next vector only.

- In 0/1 Knapsack, we need both the values from next array, that means we are using values from unupdated array can start our loop from right side to access the unupdated values from next array, and use the right part of the arr And keep on going from right to left.
- If we start loop from 0 to W or 1 to W, We will get updated values on the right side, but we want to access from



```cpp
int knapsack(vector<int>& wt, vector<int>& profit, int n
{   vector<int> next(W+1,0);
    for(int i=wt[n-1];i<=W;i++)
    {  next[i]= profit[n-1];
    }
 for(int currind=n-2;currind>=0;currind--)
 {   for(int capacity=W;capacity>=0;capacity--)
     {   int notconsider =next[capacity];
         int consider = INT_MIN;
         if(wt[currind]<=capacity)
         consider = profit[currind] + next[capacity-wt[cu
        next[capacity] = max(consider,notconsider);
    }
 }
    return next[W];
}
```

TC : O(N*W)
SC : O(W)

# *LECTURE-20 MINIMUM COINS (VARIABLE DESTINATION INDEX) MINIMUM ELEMENTS CODESTUDIO

21 June 2022 12:29

## 322. Coin Change

Medium 12401 282 Add to List Share

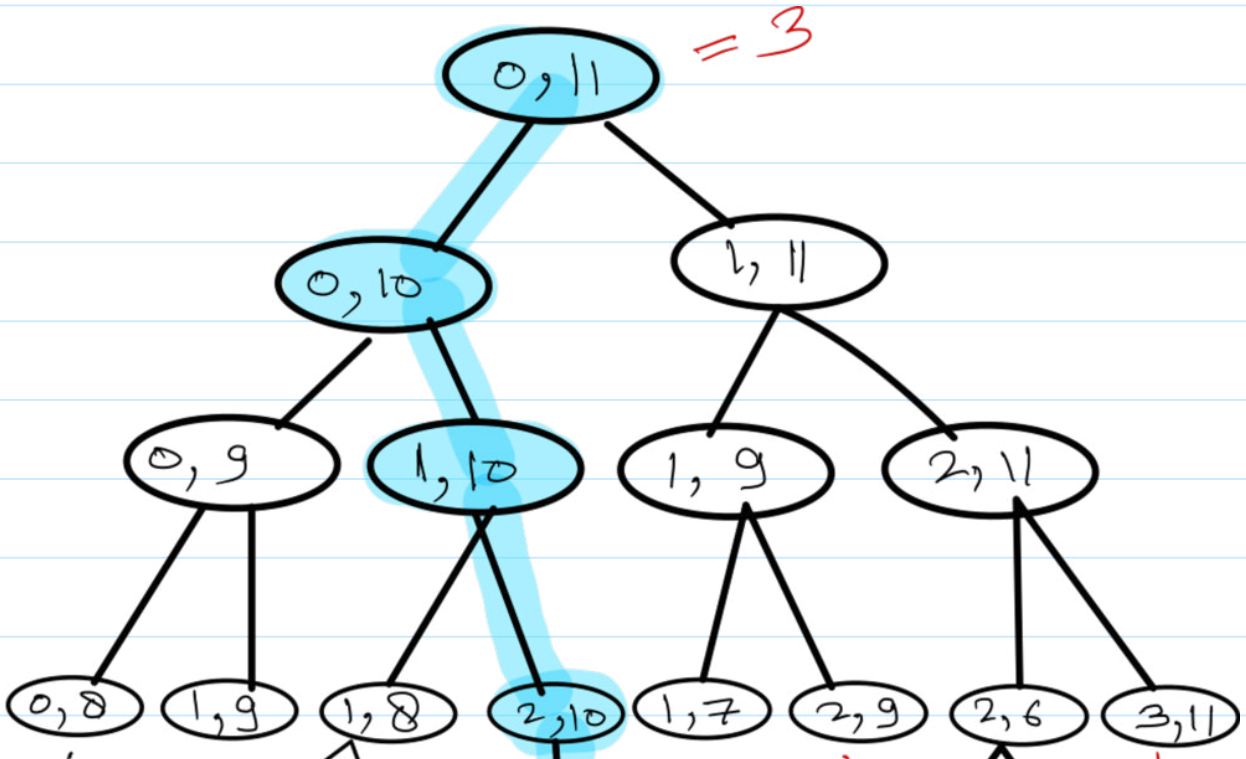
You are given an integer array `coins` representing coins of different denominations and an integer `amount` amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be combination of the coins, return `-1`.

You may assume that you have an infinite number of each kind of coin.

**Example 1:**

```
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
```



→ There can be multiple ways to make the amount, but we have to make it using minimum coins.

→ 1 + 1 + 1 + 1 . . . . . 1 = 11

→ 1 + 2 + 2 + 2 + 2 + 2 = 11

→ 1 + 1 + 2 + 2 + 5 = 11

→ 2 + 2 + 2 + 5 = 11

→ 1 + 5 + 5 = 11

**MEMOIZATION:**

```cpp
int calcoins(int currind, int lastindex, int price,vector<int> & coins, vector<vector<int>> & dp)
{   if(price==0)
      return 0;
   if(currind>lastindex)
       return 1e9;
    if(dp[currind][price]!=-1)
        return dp[currind][price];
     int notconsider = calcoins(currind+1,lastindex, price,coins, dp);
     int consider = 1e9;
    if(coins[currind]<=price)
        consider = 1 +calcoins(currind,lastindex,price-coins[currind],coin
    return dp[currind][price] = min(consider,notconsider);
}

int minimumElements(vector<int> &num, int x)
{    int n = num.size();
    vector<vector<int>> dp(n, vector<int> (x+1,-1));
 int ans =  calcoins(0,n-1,x,num,dp);
 if(ans>=1e9) return -1;
 return ans;
}
```

TC : O(N*x), where x is the amount
SC : O(N*x) + O(x),

**Tabulation:**

```cpp
int minimumElements(vector<int> &coins, int x)
{    int n = coins.size();
    vector<vector<int>> dp(n, vector<int> (x+1,0));
     for(int i =0;i<=x;i++)
     {   if(i%coins[n-1]==0)
         dp[n-1][i] = (i/coins[n-1]);
        else
            dp[n-1][i] =1e9;
     }
        for(int currind=n-2;currind>=0;currind--)
        {  for(int price =0;price<=x;price++)
            {int notconsider = dp[currind+1][price];
             int consider = 1e9;
            if(coins[currind]<=price)
                consider = 1 + dp[currind][price-coins[curri
            dp[currind][price] = min(consider,notconsider);
            }
    }
 int ans = dp[0][x];
 if(ans>=1e9)
     return -1;
 else return ans;
}
```

TC : O(N*x), where x is the amount
SC : O(N*x)

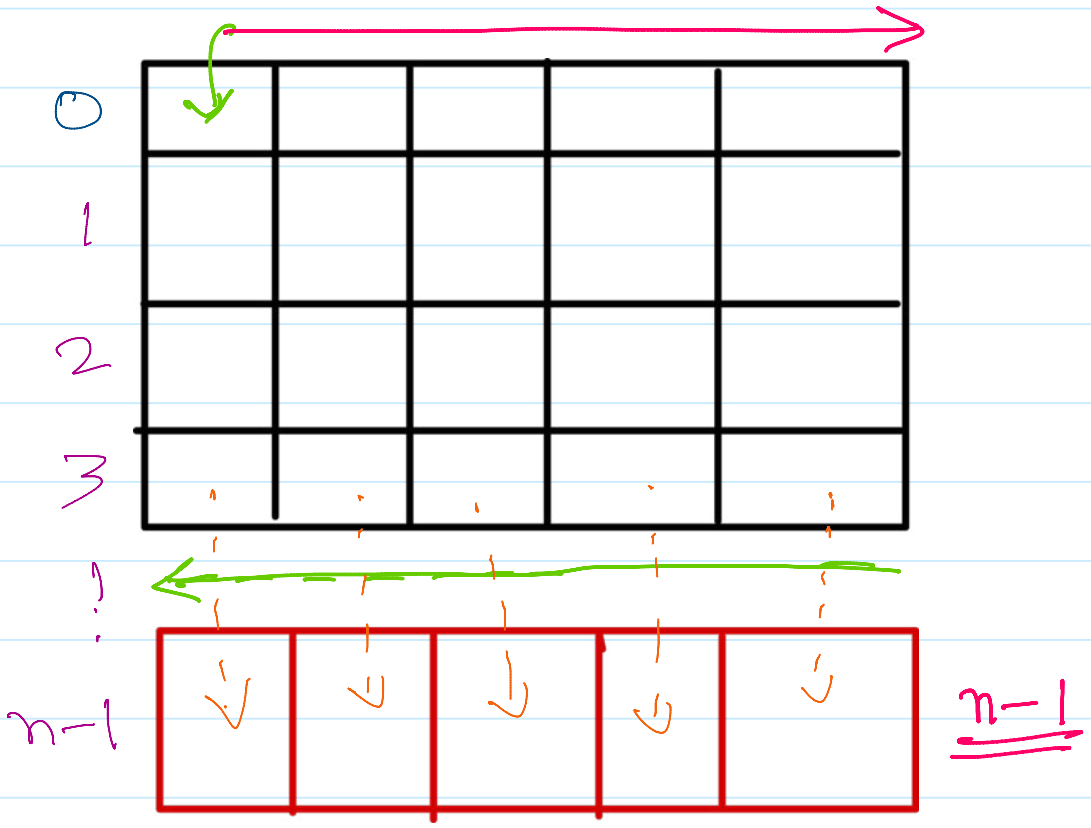
**Space Optimization:**

```cpp
int minimumElements(vector<int> &coins, int x)
{    int n = coins.size();
     vector<int> next(x+1,0);
     for(int i =0;i<=x;i++)
     {   if(i%coins[n-1]==0)
         next[i] = (i/coins[n-1]);
        else
            next[i] =1e9;
     }
        for(int currind=n-2;currind>=0;currind--)
        {  vector<int> curr(x+1,0);
            for(int price =0;price<=x;price++)
            {int notconsider = next[price];
             int consider = 1e9;
            if(coins[currind]<=price)
                consider = 1 + curr[price-coins[currind]]
            curr[price] = min(consider,notconsider);
            }
         next = curr;
    }
 int ans = next[x];
 if(ans>=1e9)
     return -1;
 else return ans;
}
```

TC : O(N*x), where x is the amount
SC : O(x)

Till now, We had done variable source index or fixed source and fixed destination index problems.

→ The second loop was dependent on the value of next row, i.e. dp[currentindex+1]




→ so either we start our second loop from 0 to target, or target to 0, the answer was same.

NOTE:- But when we are referring from current row only, we should start with base case.

**Further Space Optimization:** LinkedIn/kapilyadav22

```
int minimumElements(vector<int> &coins, int x)
{    int n = coins.size();
     vector<int> next(x+1,0);
     for(int i =0;i<=x;i++)
     {   if(i%coins[n-1]==0)
         next[i] = (i/coins[n-1]);
        else
            next[i] =1e9;
     }
        for(int currind=n-2;currind>=0;currind--)
        {
            for(int price =0;price<=x;price++)
            {int notconsider = next[price];
             int consider = 1e9;
            if(coins[currind]<=price)
                consider = 1 + next[price-coins[currind]]
            next[price] = min(consider,notconsider);
            }
    }
 int ans = next[x];
 if(ans>=1e9)
     return -1;
 else return ans;
}
```

TC : O(N*x), where x is the amount
SC : O(x)

# LECTURE-21 Target Sum (same as count partition with given difference)

Tuesday, 21 June 2022 6:46 PM

## 494. Target Sum

Medium 7157 267 Add to List Share

You are given an integer array `nums` and an integer `target`.

You want to build an **expression** out of nums by adding one of the symbols `'+'` and `'-'` before each integer in nums and then co

- For example, if `nums = [2, 1]`, you can add a `'+'` before `2` and a `'-'` before `1` and concatenate them to build the expre

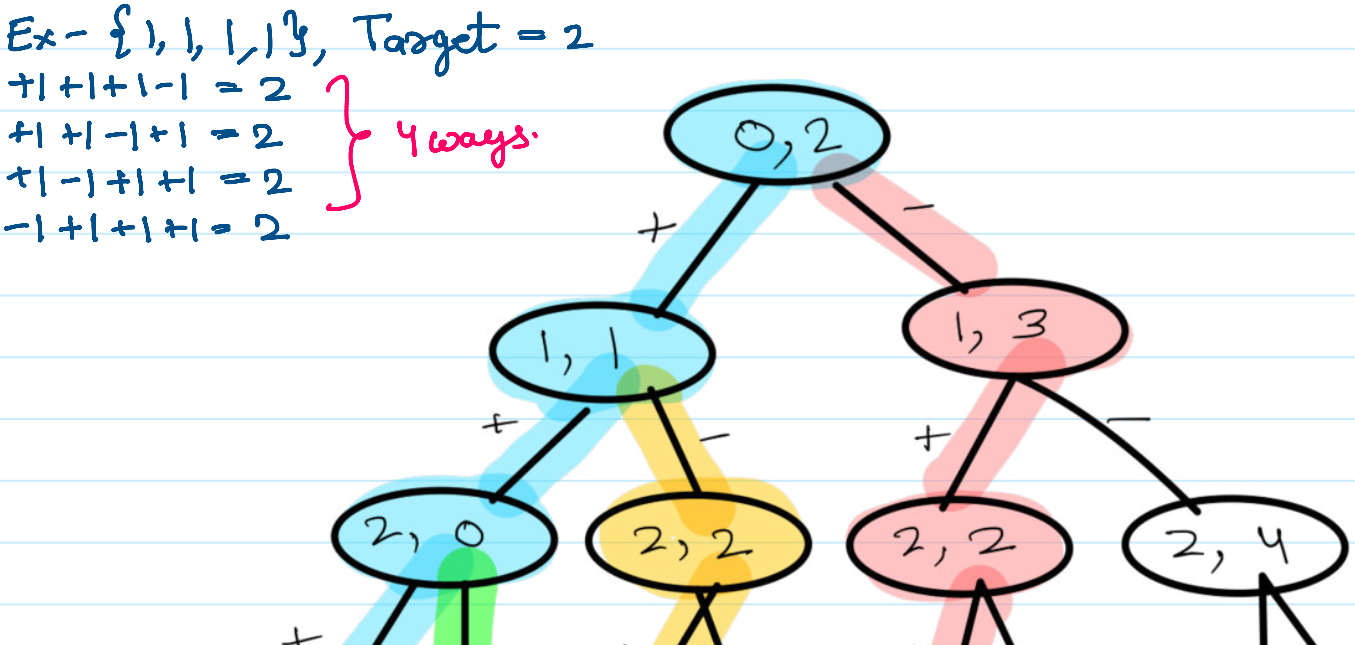
Return the number of different **expressions** that you can build, which evaluates to `target`.

**Example 1:**

```
Input: nums = [1,1,1,1,1], target = 3
Output: 5
Explanation: There are 5 ways to assign symbols to make the sum of nums be target 3.
-1 + 1 + 1 + 1 + 1 = 3
+1 - 1 + 1 + 1 + 1 = 3
+1 + 1 - 1 + 1 + 1 = 3
+1 + 1 + 1 - 1 + 1 = 3
+1 + 1 + 1 + 1 - 1 = 3
```

**There can be 2 ways to solve it:**

1. **You can do combinations of additions and subtraction in the array elements.**
2. **You can do using** Count Partitions With Given Difference .Here target can be your given difference, which reduce to Counts Subsets with Sum K, where K will (totalsum-targetsum)/2;



```cpp
int findTargetSumWays(vector<int>& nums, int target) {
        unordered_map<string,int>mp;
       return totalways(0,target,nums,mp);
    }

 int totalways(int currentindex, int target,vector<int>& nums,unordered_map<string
 {    int n=nums.size();
      if(currentindex>=n & target!=0)
          return 0;

     if(currentindex>=n & target==0)
         return 1;
  string currentkey = to_string(currentindex)+"_"+to_string(target);
  if(mp.find(currentkey)!=mp.end())
      return mp[currentkey];

   int plus = totalways(currentindex+1,target - nums[currentindex], nums,mp);
   int minus = totalways(currentindex+1, target+nums[currentindex], nums,mp);

  return mp[currentkey]= plus+minus;

 }
```

**TC:** O(N*K), where K is the target
**SC:** O(N*K) + O(N)

**2nd Way:**

{ 1, 1, 1, 1 } Target = 2

{1+1+1} − {1} = 2
 S1  S2  D

→ It can be solved using count partition with given differen

**Tabulation:**

```cpp
int countways(int tar, vector<int>& nums)
{   int n= nums.size();
 vector<int> next(tar+1,0);
  if(nums[n-1]==0)
     next[0]=2;
     else next[0]=1;
 if(nums[n-1]!=0 && nums[n-1]<=tar)
     next[nums[n-1]]=1;
  for(int currind=n-2;currind>=0;currind--)
  {    vector<int> curr(tar+1,0);
      for(int target=tar;target>=0;target--)
          {  int pick = 0;
           if(nums[currind]<=target)
             pick = next[target-nums[currind]];
             int notpick = next[target];
             curr[target] = (pick+notpick);
        }
   next = curr;
  }
 return next[tar];
 }


int targetSum(int n, int target, vector<int>& arr) {
    int totalsum = 0;
    for(int i=0;i<n;i++)
        totalsum+=arr[i];
    if((totalsum-target)<0 || (totalsum-target)%2!=0)
        return 0;
    int tar = (totalsum-target)/2;
```

# LECTURE - 22 Coin Change 2 (Count ways )

Tuesday, 21 June 2022 6:49 PM

## Ways To Make Coin Change

### 518. Coin Change 2

Medium 5421 107 Add to List Share

You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Return *the number of combinations that make up that amount.* If that amount of money cannot be made up by any combination of the coins, return `0`.

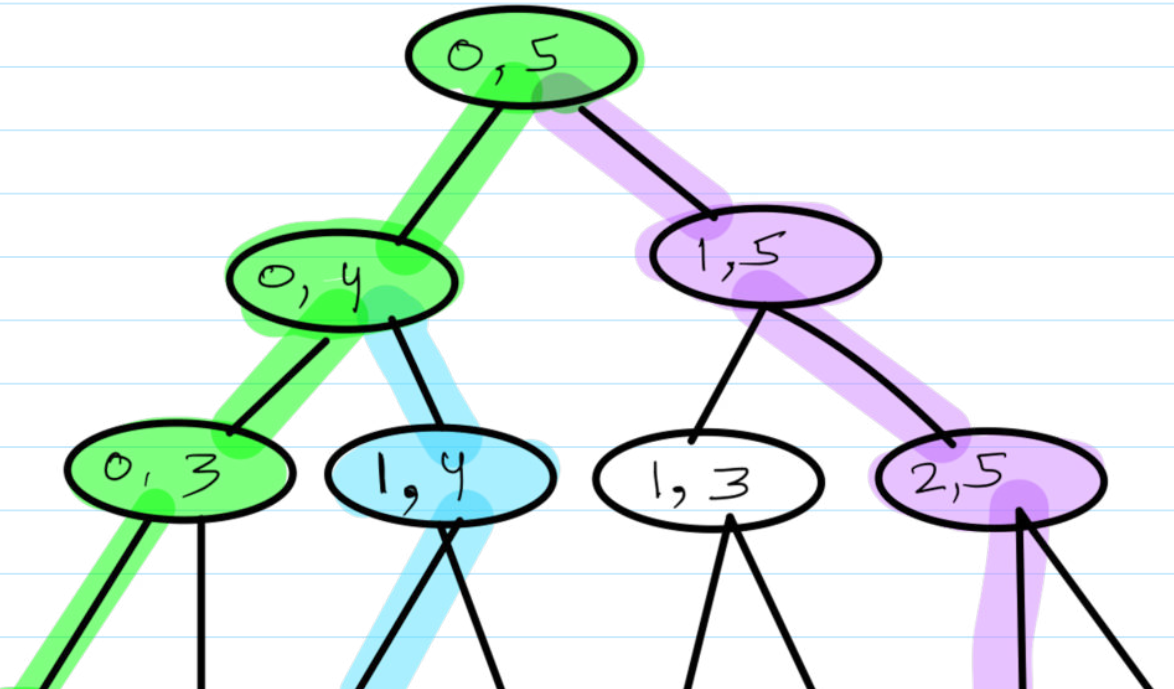
You may assume that you have an infinite number of each kind of coin.

The answer is **guaranteed** to fit into a signed **32-bit** integer.

**Example 1:**

```
Input: amount = 5, coins = [1,2,5]
Output: 4
Explanation: there are four ways to make up the amount:
5=5
5=2+2+1
5=2+1+1+1
5=1+1+1+1+1
```

We need to count all the possible ways to make the amount using given denomination.



**MEMOIZATION:**

```cpp
#include<bits/stdc++.h>
long countways(int currind,int lastindex,int value,int *denominations,
vector<vector<long>> & dp)
{    if(value==0)
     return 1;
     if(currind>lastindex)
        return 0;
    if(dp[currind][value]!=-1)
        return dp[currind][value];
  long notpick = countways(currind+1,lastindex,value,denominations,dp);
  long pick = 0;
     if(denominations[currind]<=value)
         pick = countways(currind,lastindex,value-
denominations[currind],denominations,dp);
    return dp[currind][value]= pick + notpick;
}


long countWaysToMakeChange(int *denominations, int n, int value)
{  vector<vector<long>> dp(n,vector<long> (value+1,-1));
    return countways(0,n-1,value,denominations,dp);
}
```

| **TC:** O(N*K), where K is the amount | **SC:** O(N*K) + O(K) |
|---|---|

**Tabulation:**

```cpp
long countWaysToMakeChange(int *denominations, int n, int value)
{  vector<vector<long>> dp(n,vector<long> (value+1,0));

 for(int i=0;i<=value;i++)
 {   dp[n-1][i]= (i%denominations[n-1]==0);
 }
 for(int currind=n-2;currind>=0;currind--)
        {  for(int price =0;price<=value;price++)
           { long notpick = dp[currind+1][price];
             long pick = 0;
    if(denominations[currind]<=price)
        pick = dp[currind][price-denominations[currind]];
      dp[currind][price]= pick + notpick;
     }
    }
 return dp[0][value];
}
```

**Space Optimization:**

```
long countWaysToMakeChange(int *denominations, int n, int value)
{  vector<int> next(value+1,0);
 for(int i=0;i<=value;i++)
 {   next[i]= (i%denominations[n-1]==0);
 }
 for(int currind=n-2;currind>=0;currind--)
        {  vector<int> curr(value+1,0);
     for(int price =0;price<=value;price++)
           { long notpick = next[price];
             long pick = 0;
    if(denominations[currind]<=price)
        pick = curr[price-denominations[currind]];
      curr[price]= pick + notpick;
     }
         next = curr;
    }
 return next[value];
}
```

**TC:** O(N*K), where K is the amount
**SC:** O(K)

**Further Space Optimization:** Using one 1D array only

```
long countWaysToMakeChange(int *denominations, int n, int value)
{  vector<long> next(value+1,0);
 for(int i=0;i<=value;i++)
 {   next[i]= (i%denominations[n-1]==0);
 }
 for(long currind=n-2;currind>=0;currind--)
        {
     for(long price =0;price<=value;price++)
           { long notpick = next[price];
             long pick = 0;
    if(denominations[currind]<=price)
```

# LECTURE-23 Unbounded Knapsack

Tuesday, 21 June 2022 6:50 PM

**Problem Statement**

You are given 'N' items with certain 'PROFIT' and 'WEIGHT' and a knapsack with weight
You need to fill the knapsack with the items in such a way that you get the maximum prof
allowed to take one item multiple times.

**For Example**

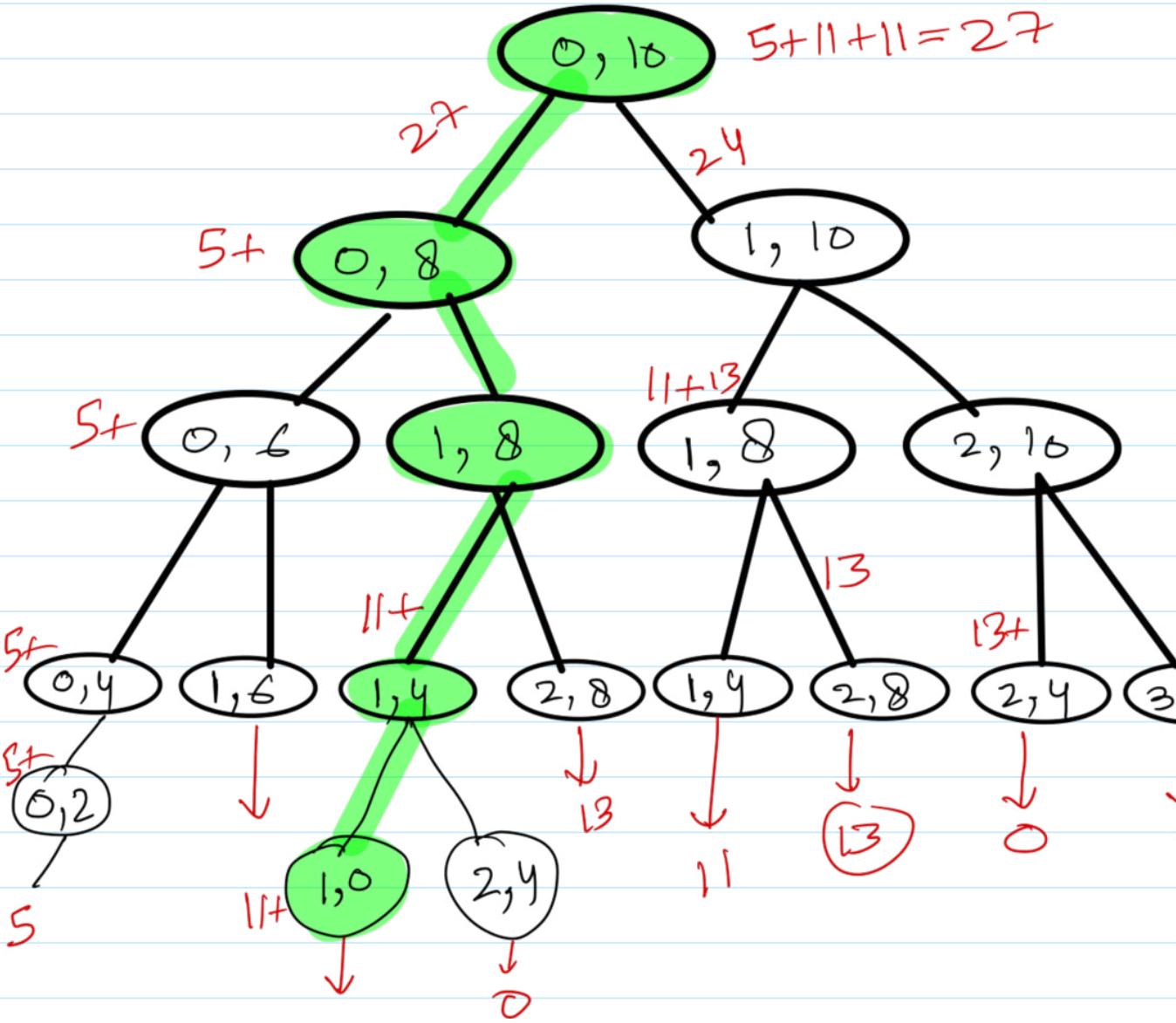
```
Let us say we have 'N' = 3 items and a knapsack of capacity 'W' =  10
'PROFIT' = { 5, 11, 13 }
'WEIGHT' = { 2, 4, 6 }

We can fill the knapsack as:

1 item of weight 6 and 1 item of weight 4.
1 item of weight 6 and 2 items of weight 2.
2 items of weight 4 and 1 item of weight 2.
5 items of weight 2.

The maximum profit will be from case 3 i.e '27'. Therefore maximum profit = 27.
```

| wt | 2 | 4 | 6 |
|---|---|---|---|
| value | 5 | 11 | 13 |

← unlimited supply

W = 10

→ 5x5 = 25,
→ 11+11+5 = 27
→ 13+11 = 24
→ 13+10 = 23




**MEMOIZATION:**

```
1 int findknap(int currind, int lastind, int capacity, vector<int>
  &profit, vector<int> &weight,
2               vector<vector<int>> & dp)
3 {   if(currind==lastind)
4     { return ((int)(capacity/weight[lastind])) * profit[lastind
5     }
6      if(dp[currind][capacity]!=-1)
7          return dp[currind][capacity];
8      int notpick =
  findknap(currind+1,lastind,capacity,profit,weight,dp);
9      int pick=-1e9;
10     if(weight[currind]<=capacity)
```

**TC:** O(N*W), where W is the Weight capacity of the knapsack
**SC:** O(N*W) + O(W)

**Tabulation:**

```cpp
int unboundedKnapsack(int n, int w, vector<int> &profit, vector<int>
&weight)
{  vector<vector<int>> dp(n,vector<int> (w+1,0));
    for(int i=weight[n-1];i<=w;i++)
        dp[n-1][i] = ((int)(i/weight[n-1])) * profit[n-1];

      for(int currind=n-2;currind>=0;currind--)
      {   for(int capacity=0;capacity<=w;capacity++)
          { int notpick = dp[currind+1][capacity];
     int pick=0;
     if(weight[currind]<=capacity)
         pick = profit[currind] + dp[currind][capacity-weight[currind]
        dp[currind][capacity]= max(pick,notpick);
          }
     }
   return dp[0][w];
}
```

**TC:** O(N*W), where W is the Weight capacity of the knapsack
**SC:** O(N*W)

**Space Optimization:**

```cpp
int unboundedKnapsack(int n, int w, vector<int> &profit, vector<int>
&weight)
{  vector<int> next(w+1,0);
    for(int i=weight[n-1];i<=w;i++)
        next[i] = ((int)(i/weight[n-1])) * profit[n-1];

      for(int currind=n-2;currind>=0;currind--)
      {  vector<int> curr(w+1,0);
          for(int capacity=0;capacity<=w;capacity++)
```

**TC:** O(N*W), where W is the Weight capacity of the knapsack
**SC:** O(W)

**Further Space Optimization:** Using one 1D array

```
int unboundedKnapsack(int n, int w, vector<int> &profit, vector<
&weight)
{  vector<int> next(w+1,0);
    for(int i=weight[n-1];i<=w;i++)
        next[i] = ((int)(i/weight[n-1])) * profit[n-1];

      for(int currind=n-2;currind>=0;currind--)
      {  for(int capacity=0;capacity<=w;capacity++)
          { int notpick = next[capacity];
     int pick=0;
     if(weight[currind]<=capacity)
         pick = profit[currind] + next[capacity-weight[currind]]
        next[capacity]= max(pick,notpick);
          }

     }
   return next[w];
}
```



**TC:** O(N*W), where W is the Weight capacity of the knapsack
**SC:** O(W)

# LECTURE - 24 ROD CUTTING

22 June 2022 12:13

## Rod cutting problem

53 Difficulty: MEDIUM

Contributed By
Mutiur Rehman khan | Level 1

Avg. time to solve
40 min

Success Rate
75%

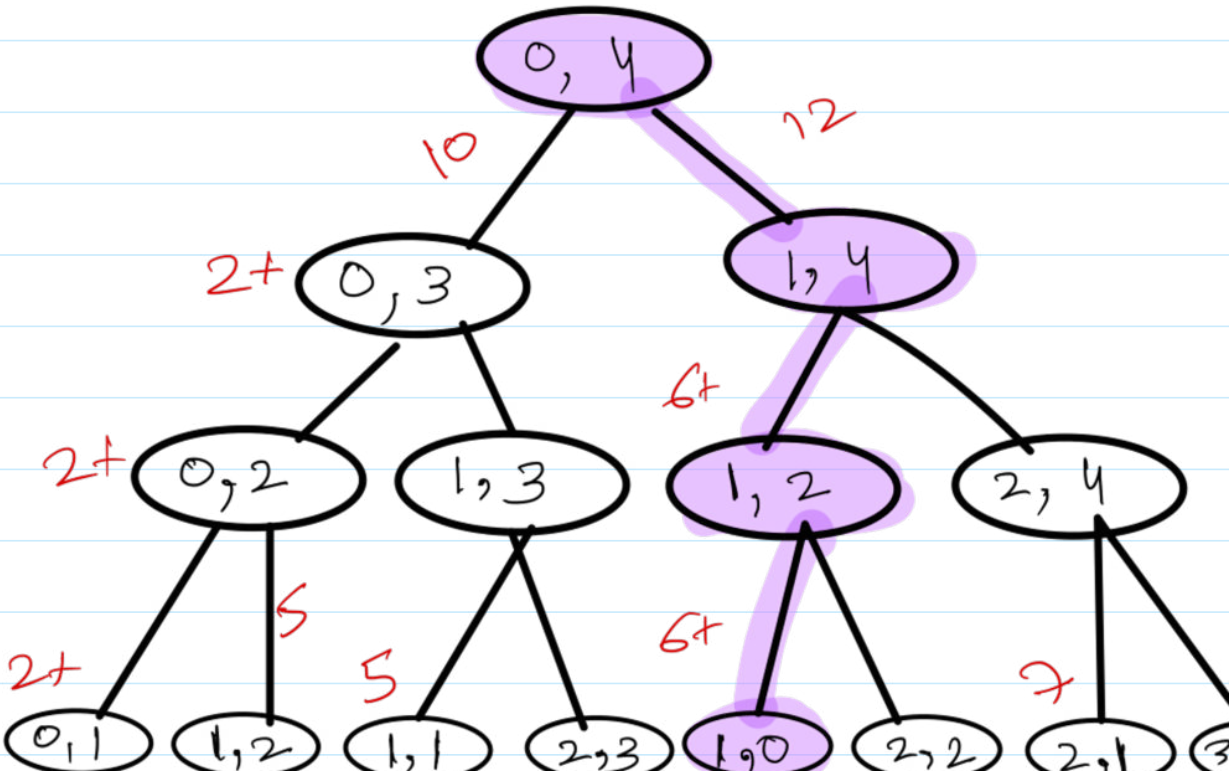
### Problem Statement

Given a rod of length 'N' units. The rod can be cut into different sizes and each size h
associated with it. Determine the maximum cost obtained by cutting the rod and sell

Note:

1. The sizes will range from 1 to 'N' and will be integers.
2. The sum of the pieces cut should be equal to 'N'.
3. Consider 1-based indexing.

**Input: 2 6 7 8**



**MEMOIZATION:**

```
int cutRod(int price[], int n) {
        //code here
        vector<vector<int>> v(n+1,vector<int> (n+1,-1));
        return maxprofit(price,0,n,n,v);
    }

    int maxprofit(int price[], int currentindex, int n,int length,vector<vector<int>
    {
        if(currentindex>=n || length==0)
            return 0;


    int currentlength= currentindex+1;

    if(v[currentlength][length]!=-1)
        return v[currentlength][length];


        int consider=0;
       if(currentlength<=length)
        consider= price[currentindex] + maxprofit(price, currentindex,n, length-curr
        int notconsider= maxprofit(price,currentindex+1,n,length,v);

    return v[currentlength][length]=max(consider,notconsider);

    }
```

**TC:** O(N*N), where  N is the length of the Rod
**SC:** O(N*N) + O(N) recursve stack space where the rod is divided into N parts

**Tabulation:**

```
int cutRod(vector<int> &price, int n)
{ vector<vector<int>> dp(n,(vector<int> (n+1,0)));
   for(int i = 0;i<=n;i++)
     { dp[0][i] = i* price[0];
      }

 for(int currind = 1;currind<n;currind++)
 { for(int N =0;N<=n;N++)
     { int nottake = dp[currind-1][N];
     int take = -10000000;
     int rodlength = currind+1;
     if(rodlength<=N){
     take = price[currind] + dp[currind][N - rodlen
 }    dp[currind][N] = max(take,nottake);
     }
 }
 return dp[n-1][n];
}
```

**TC:** O(N*N), where N is the length of the Rod
**SC:** O(N*N)

**Space Optimization:**

```cpp
int cutRod(vector<int> &price, int n)
{ vector<int> next(n+1,0);
   for(int i = 0;i<=n;i++)
     { next[i] = i* price[0];
      }

 for(int currind = 1;currind<n;currind++)
 { vector<int> curr(n+1,0);
     for(int N =0;N<=n;N++)
     { int nottake = next[N];
     int take = -10000000;
     int rodlength = currind+1;
     if(rodlength<=N){
     take = price[currind] + curr[N - rodlength];
 }    curr[N] = max(take,nottake);
     }
  next = curr;
 }
 return next[n];
}
```

**TC:** O(N*N), where  N is the length of the Rod
**SC:** O(N)

**Further Space Optimization:** Using one 1D array

```cpp
int cutRod(vector<int> &price, int n)
{ vector<int> next(n+1,0);
   for(int i = 0;i<=n;i++)
     { next[i] = i* price[0];
      }

 for(int currind = 1;currind<n;currind++)
 {
     for(int N =0;N<=n;N++)
     { int nottake = next[N];
     int take = -10000000;
     int rodlength = currind+1;
     if(rodlength<=N){
     take = price[currind] + next[N - rodlength];
 }    next[N] = max(take,nottake);
     }
 }
 return next[n];
}
```



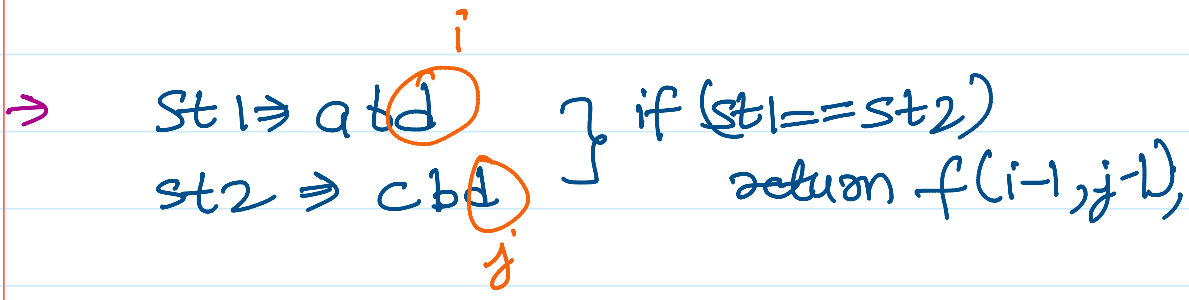
**TC:** O(N*N), where N is the length of the Rod
**SC:** O(N)

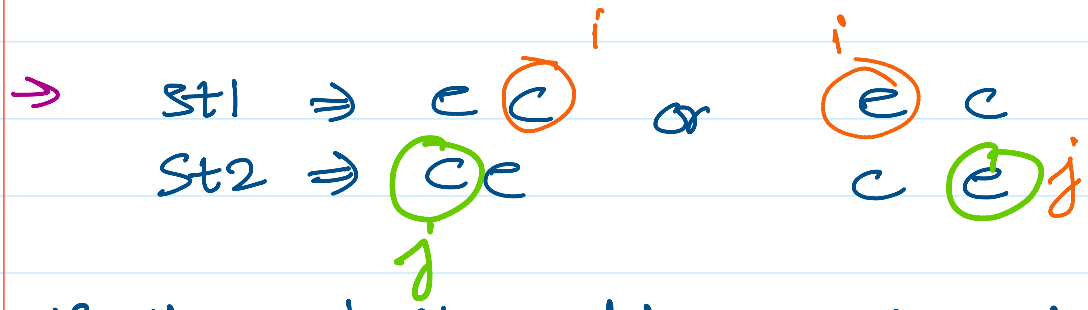
# *LECTURE - 25 LONGEST COMMON SUBSEQUENCE (DP ON STRINGS)

22 June 2022 12:13

- A **subarray is a contiguous part of array and maintains relative ordering of elements**. For an array/string of n*(n+1)/2 non-empty subarrays/substrings.
- A **subsequence maintain relative ordering of elements but may or may not be a contiguous part of an arra** size n, we can have 2^n-1 non-empty sub-sequences in total.
- A **subset does not maintain relative ordering of elements and is neither a contiguous part of an array**. For a have (2^n) sub-sets in total.

**Abc = a, b, c, ab, bc, ac, abc, "**



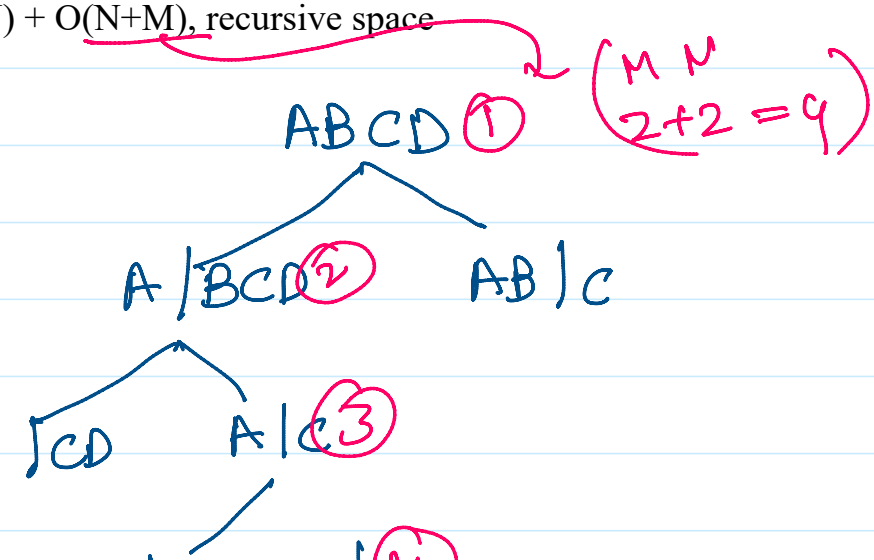

if they don't match, we will explore all other possibilities.

f(i-1, j) } max
f(i, j-1) }

**MEMOIZATION:**

```
int findlcs(int ind1, int ind2, string s1, string s2, vector<vecto
    {   if(ind1>s1.length()-1 || ind2>s2.length()-1)
        return 0;
        if(dp[ind1][ind2]!=-1)
            return dp[ind1][ind2];
        if(s1[ind1]==s2[ind2])
         return dp[ind1][ind2]=1 + findlcs(ind1+1,ind2+1,s1,s2,dp)
       if(s1[ind1]!=s2[ind2])
           return dp[ind1][ind2]=max(findlcs(ind1+1, ind2,s1,s2,dp
                                     findlcs(ind1,ind2+1,s1,s2,dp))
     return 0;
    }


    int longestCommonSubsequence(string s1, string s2) {
        int m = s1.length();
        int n = s2.length();
        vector<vector<int>> dp(m+1,vector<int>(n+1,-1));
       return findlcs(0,0,s1,s2,dp);
    }
```

TC: O(M*N), where M is the length of string1 and N is the length of string2.
SC : O(M*N) + O(N+M), recursive space



**Tabulation:**

```
int longestCommonSubsequence(string s1, string s2) {
        int m = s1.length();
        int n = s2.length();

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

        for(int ind1 = m-1;ind1>=0;ind1--)
        { for(int ind2 = n-1;ind2>=0;ind2--)
            { if(s1[ind1]==s2[ind2])
             dp[ind1][ind2]=1 + dp[ind1+1][ind2+1];
            else
            dp[ind1][ind2]=max(dp[ind1+1][ind2],dp[ind1][in
            }
        }
        return dp[0][0];
    }
```

TC: O(M*N), where M is the length of string1 and N is the length of string2.
SC : O(M*N)



**Space Optimization:**

```
int longestCommonSubsequence(string s1, string s2) {
        int m = s1.length();
        int n = s2.length();

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

        for(int ind1 = m-1;ind1>=0;ind1--)
        { vector<int> curr(n+1,0);
            for(int ind2 = n-1;ind2>=0;ind2--)
            { if(s1[ind1]==s2[ind2])
```

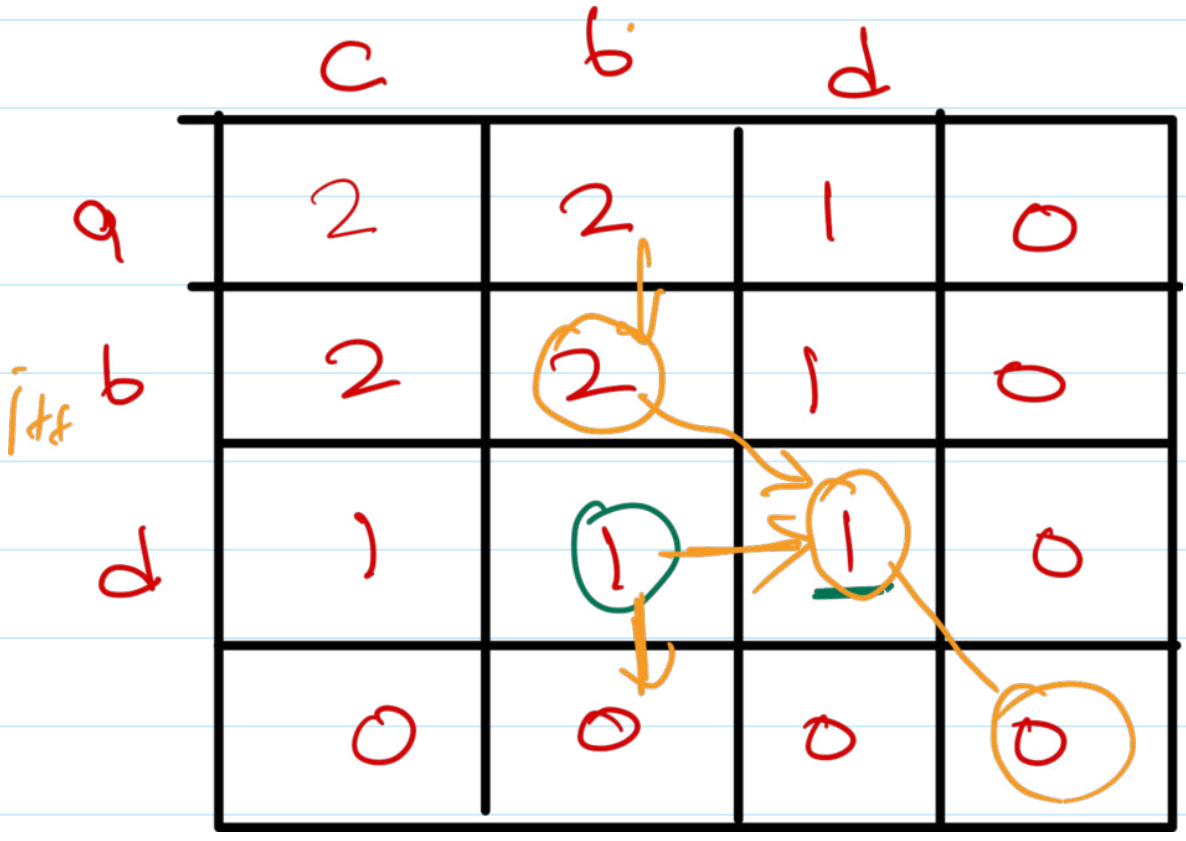
# LECTURE - 26 PRINT LONGEST COMMON SUBSEQUENCE

22 June 2022 17:30

**First Way:**

```cpp
//print the longest common susbsequence
for(int i =0;i<m+1;i++)
        { for(int j=0;j<n+1;j++)
            {  cout<<dp[i][j]<<" ";
               if(dp[i][j]>dp[i+1][j] && dp[i][j]>dp[i][j+1])
                   cout<<s2[j]<<" ";
               //or print s1[i];
            }
         cout<<endl;
        }
```

**Second Way:**



```
string ans = "";
int i=0;int j=0;
while(i<m && j<n)
{   if(s1[i]==s2[j])
    {  ans+=s1[i];
       i++;
       j++;
    }
 else if(dp[i+1][j]>dp[i][j+1])
       i++;
 else j++;
}
cout<<ans;
```

LinkedIn/kapilyadav22

# LECTURE - 27 LONGEST COMMON SUBSTRING

23 June 2022 12:40

**Problem Statement**

You have been given two strings 'STR1' and 'STR2'. You have to find the length of the lon substring.

A string "s1" is a substring of another string "s2" if "s2" contains the same characters as same order and in continuous fashion also.

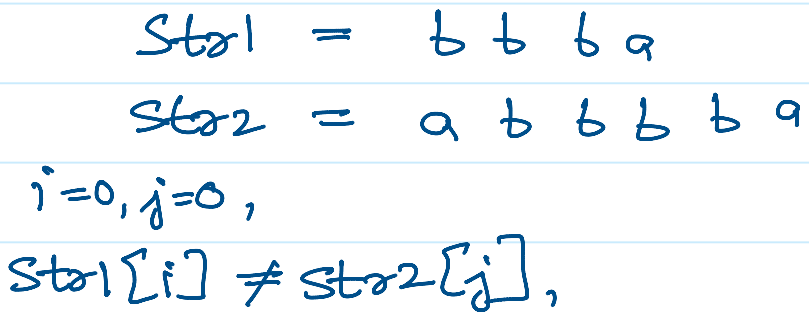
Same as Longest CommonSubsequence, But here If the strings are not matching, just pass the countlcs =0, and fi common string from left and right.

There will be three recursive calls everytime.

1. If both the strings are matching, increament countlcs and move the indexes of both the strings.
2. If strings are not matching, we need to check by iterating both the strings one by one.
   Whichever will give the maximum, we will take it.

   Example: str1 : bbba str2: abbbbba.

   If we only increament count by comparing 2 strings then answer will be 3 as bbb matching,
   But actual answer is 4, because bbba matching in both the strings. So we need to check for both the cases(i+1,j) and (
   Characters of both the strings are matching.




**Recursion:**

```
1   int findlcs(int ind1, int ind2, string s1, string s2,int countlcs)
2    {   if(ind1>s1.length()-1 || ind2>s2.length()-1)
3        return countlcs;
4
5         if(s1[ind1]==s2[ind2])
6         countlcs  = findlcs(ind1+1,ind2+1,s1,s2,countlcs+1);
7
8        int notmatch= max(findlcs(ind1,ind2+1,s1,s2,0),findlcs(ind1+1,ind2,s1,
```

**MEMOIZATION:**

```
  int findlcs(int ind1, int ind2,int countlcs, string s1, string s2,vector<vector<vector<int
   {   if(ind1>s1.length()-1 || ind2>s2.length()-1)
       return countlcs;

        if (dp[ind1][ind2][countlcs]!= -1)
           return dp[ind1][ind2][countlcs];
      int match=countlcs;
        if(s1[ind1]==s2[ind2])
      match  = findlcs(ind1+1,ind2+1,countlcs+1,s1,s2,dp);

       int notmatch= max(findlcs(ind1,ind2+1,0,s1,s2,dp),findlcs(ind1+1,ind2,0,s1,s2,dp));
       return dp[ind1][ind2][countlcs] =max(match,notmatch);
   }

int lcs(string &s1, string &s2){
    int m = s1.length();
       int n = s2.length();
   int countlcs=min(m,n);
       vector<vector<vector<int>>> dp(m+1,vector<vector<int>> (n+1,vector<int>(countlcs,-1))
      return findlcs(0,0,0,s1,s2,dp);
}
```

**TC: O(M*N*C)**
**SC:O(M*N*C) + O(M+N),where C is the min(M,N)**

we just don't need to iterate when the characters of two strings are not matching.

## Tabulation Method:

```
int lcs(string &s1, string &s2){
    int m = s1.length();
        int n = s2.length();

        vector<vector<int>> dp(m+1,vector<int>(n+1,0));
        int ans = -10000;
        for(int ind1 = m-1;ind1>=0;ind1--)
        { for(int ind2 = n-1;ind2>=0;ind2--)
            { if(s1[ind1]==s2[ind2])
             dp[ind1][ind2]=1 + dp[ind1+1][ind2+1];
             ans = max(ans,dp[ind1][ind2]);
            }
        }
```

**Space Optimization:**

```cpp
int lcs(string &s1, string &s2){
    int m = s1.length();
        int n = s2.length();

       vector<int> next(n+1,0);
        int ans = -10000;
        for(int ind1 = m-1;ind1>=0;ind1--)
        { vector<int> curr(n+1,0);
            for(int ind2 = n-1;ind2>=0;ind2--)
            { if(s1[ind1]==s2[ind2])
             curr[ind2]= 1 + next[ind2+1];
             ans = max(ans,curr[ind2]);
            }
         next = curr;
        }
        return ans;
}
```

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

# LECTURE - 28 LONGEST PALINDROMIC SUBSEQUENCE

23 June 2022 12:41

Ex → "bbaabaabb" → S1

Take reverse of string

"bbaabaabb" → S2

Now LPS of S1 will be LCS of (S1, S2)

⇒ bbaaabb. or bbabaabb

**Tabulation:**

```
int longestCommonSubsequence(string s1, string s2) {
        int m = s1.length();
        int n = s2.length();

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

        for(int ind1 = m-1;ind1>=0;ind1--)
        { vector<int> curr(n+1,0);
            for(int ind2 = n-1;ind2>=0;ind2--)
            { if(s1[ind1]==s2[ind2])
             curr[ind2]=1 + next[ind2+1];
            else
            curr[ind2]=max(next[ind2],curr[ind2+1]);
            }
         next = curr;
        }
        return next[0];
    }
```

# Congrats,

# You Have studied DP