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(N)

Tabulation :-

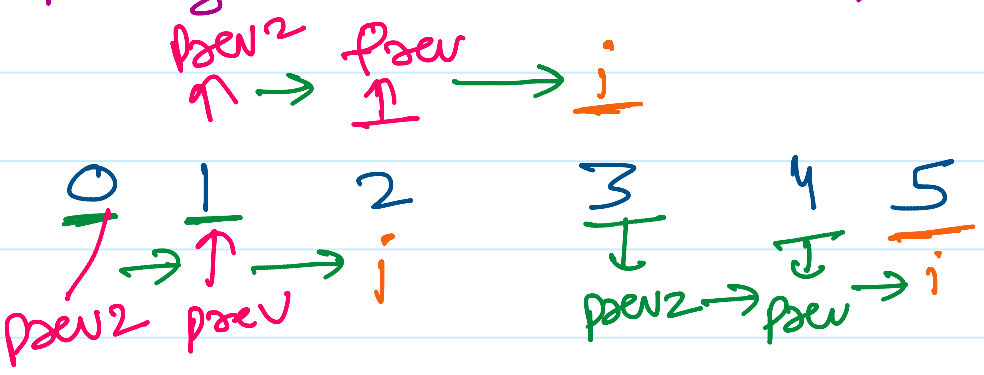
1.) Write base cases.

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;



at last, i will exceed n, and prev will be n, so return prev.

pzw will be n, so return pzw.

```
for(i=2; i<=N; i++)
{
    int curr = pzw + pzw2; pzw2 = pzw
     pzw = curr;
}
   return pzw;
```

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;
16     return v[currentkey];
17   }
```

Tabulation:

```cpp
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];
   }
```

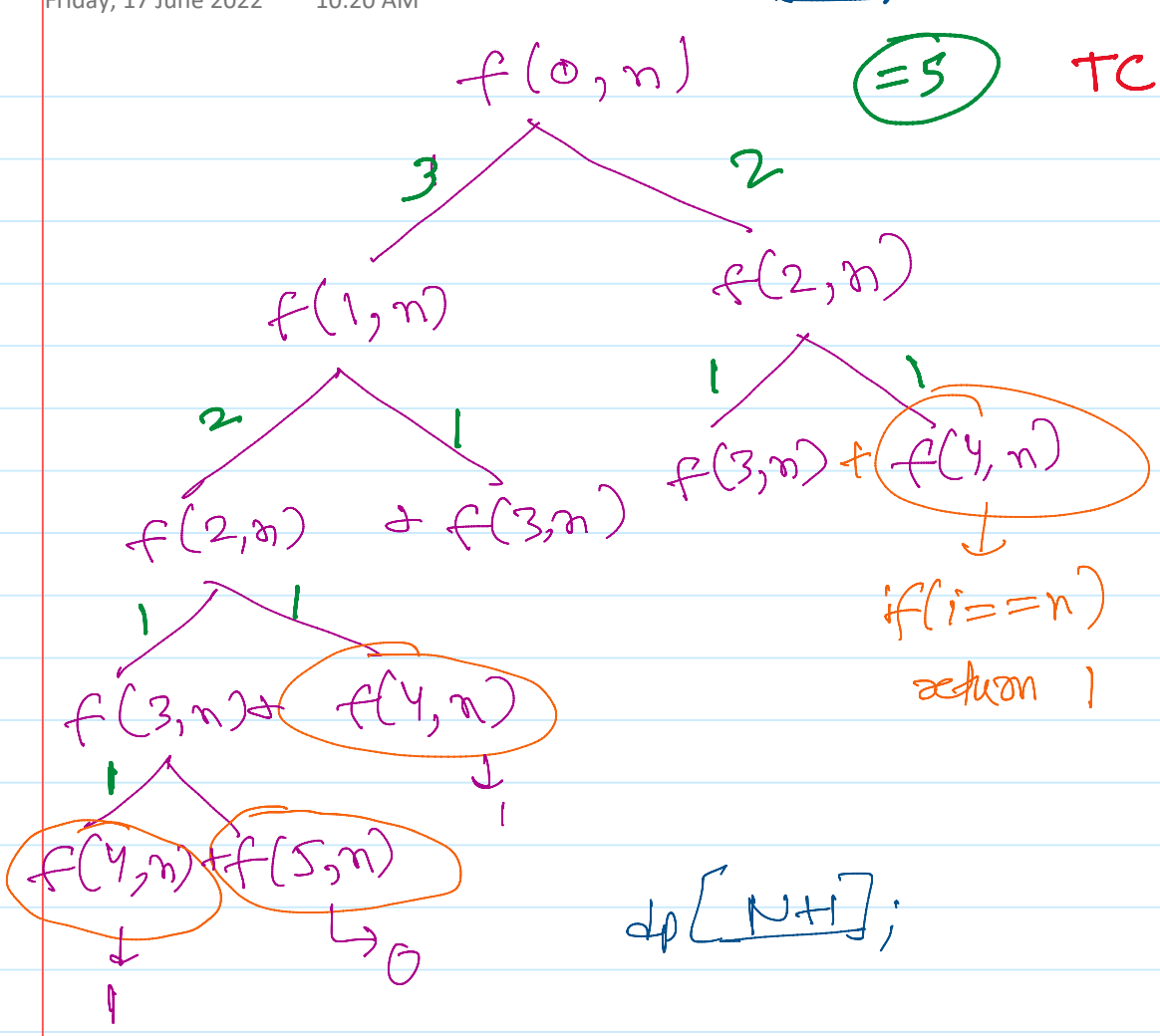
Space Optimization:

```cpp
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;
        prev2= prev;
        prev = curr;


       }
     return prev;
   }
```

LEC - 2, CLIMB STAIRS

Friday, 17 June 2022 10:20 AM

N = 4



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];
```

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

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

}

Memoization:

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

    int noofways(int currentindex, int n, vector<int>&v)
    {
      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);
        v[currentkey] = onestep+twosteps;
        return onestep+twosteps;
         }
    }
```

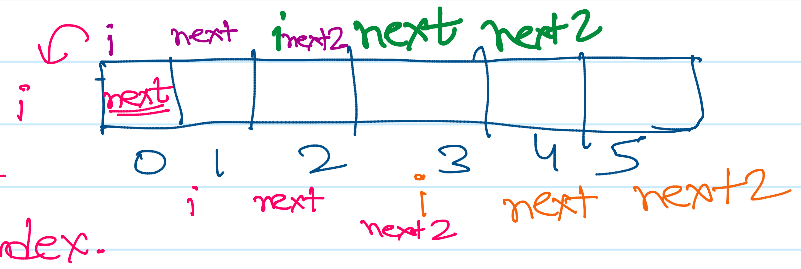
Tabulation :- take dp(n+2), because at 3, if we take 2 steps, we will reach to 5.

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

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

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

Space Optimization :-

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




```java
int climbStairs(int n) {
        int next2=0;
        int next1=1;

        for(int CurrInd = n-1;CurrInd>=0;CurrInd--)
        {  int steps= next1 + next2;
            next2 =next1;
            next1 = steps;
        }
        return next1;
    }
```

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

LinkedIn/kapilyadav22

Lecture-3 Frog JUMP

17 June 2022 11:14

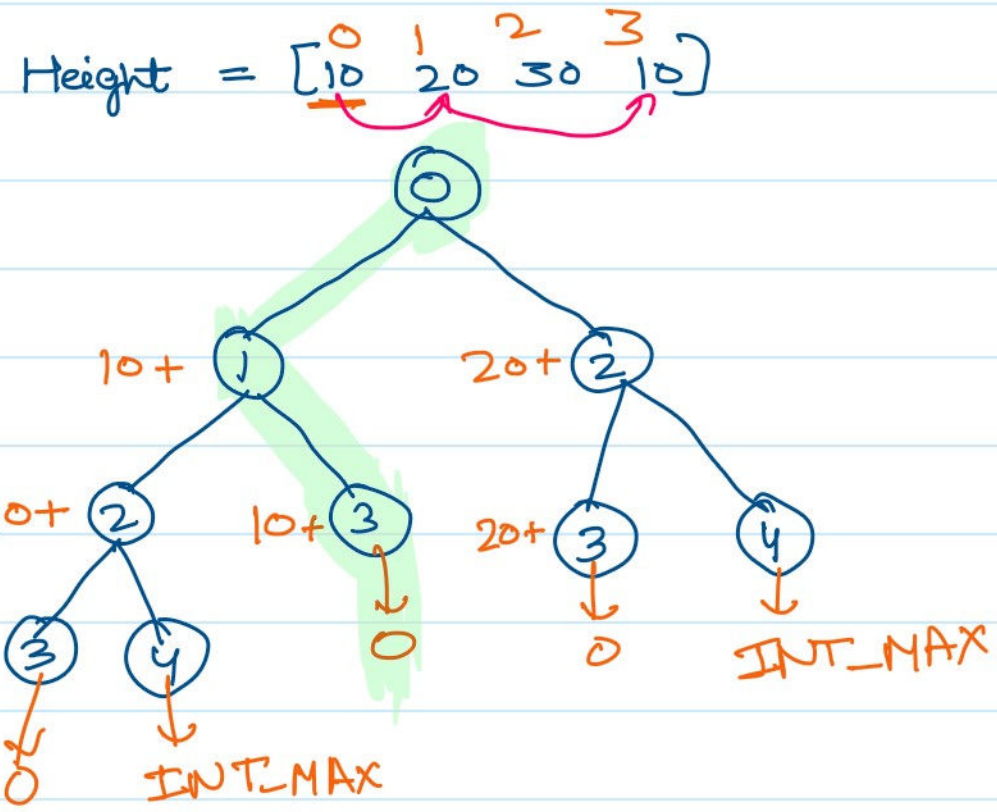
It is same as Min cost to climb the stairs (Leetcode), just the cost is added as
abs(heights[currindex]-height[currindex+1]) in case of jumping one step.
abs(heights[currindex]-height[currindex+2]) in case of jumping two steps

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:

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

```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;
}
```

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:

```cpp
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)XO(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 money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will 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 of money you can rob 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

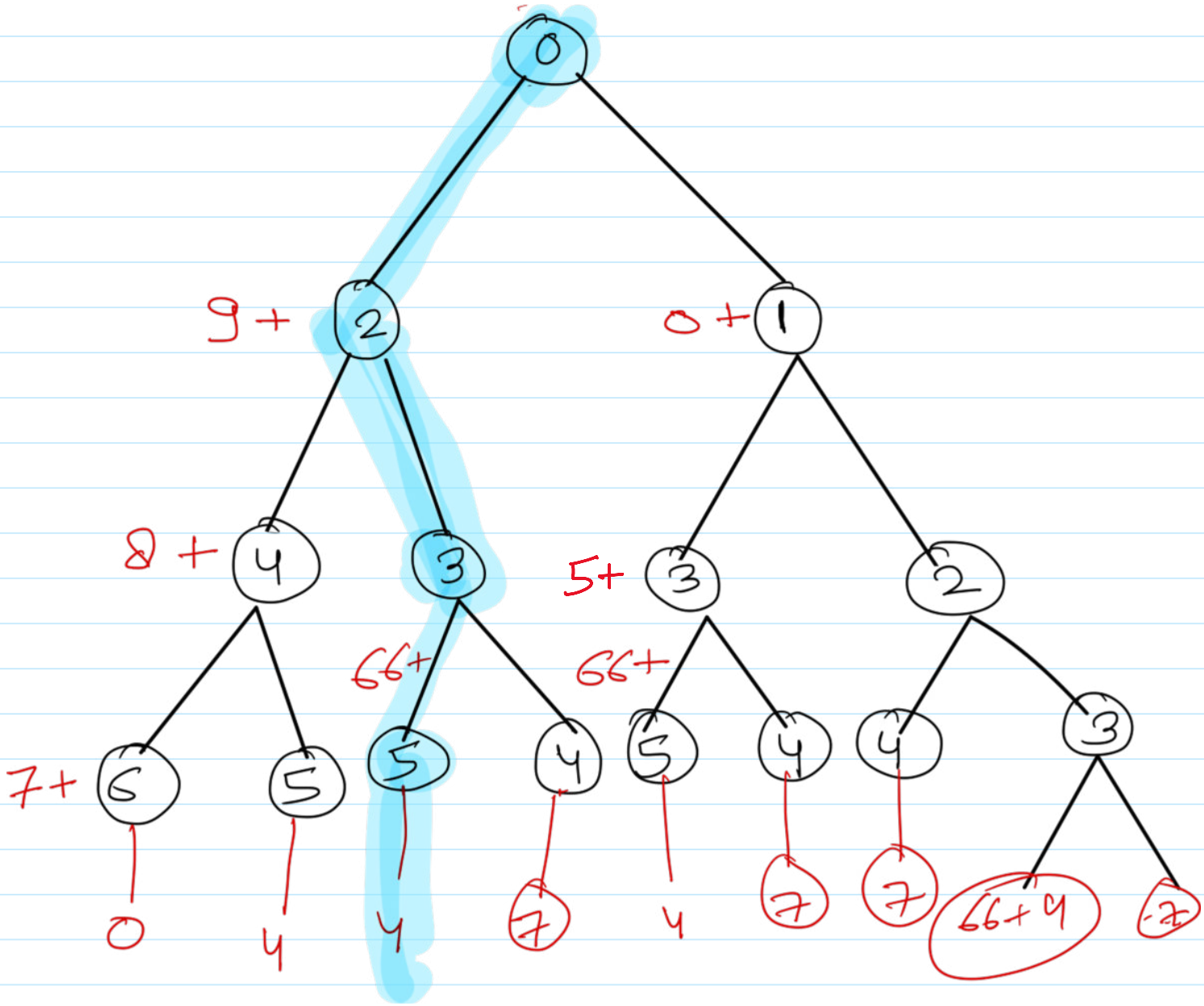
→ 7 + 1 = 8

→ 9 + 1 = 10

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

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

0 1 2 3 4 5


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
4
7
4
7
7
66+4
7

GFG:- Recursion + Memoization.

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

```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<int> &v, 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,nums);

       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(1)

BY DOING SAME AS DELETE AND EARN:

same as Delete & earn.

```
ni = arr[i] + excl;
ne = max(incl, excl);
```

LinkedIn/kapilyadav22

```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 money stashed. All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and **it will 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 of money 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 they are
adjacent houses.
```

- Same as House robber one, but since houses are in Circle, so if you rob first house, you cannot rob last house 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:

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

```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>& dp)
{  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);
}
```

TABULATION:

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

```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);
}
```

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 Suggest Edit

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

You are given a 2D array of size N*3 'POINTS' with the points corresponding to each day and 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 will
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.
so maxprofit = 50+11 = 61.

→ But, if i choose 10 + 100
maxprofit = 110

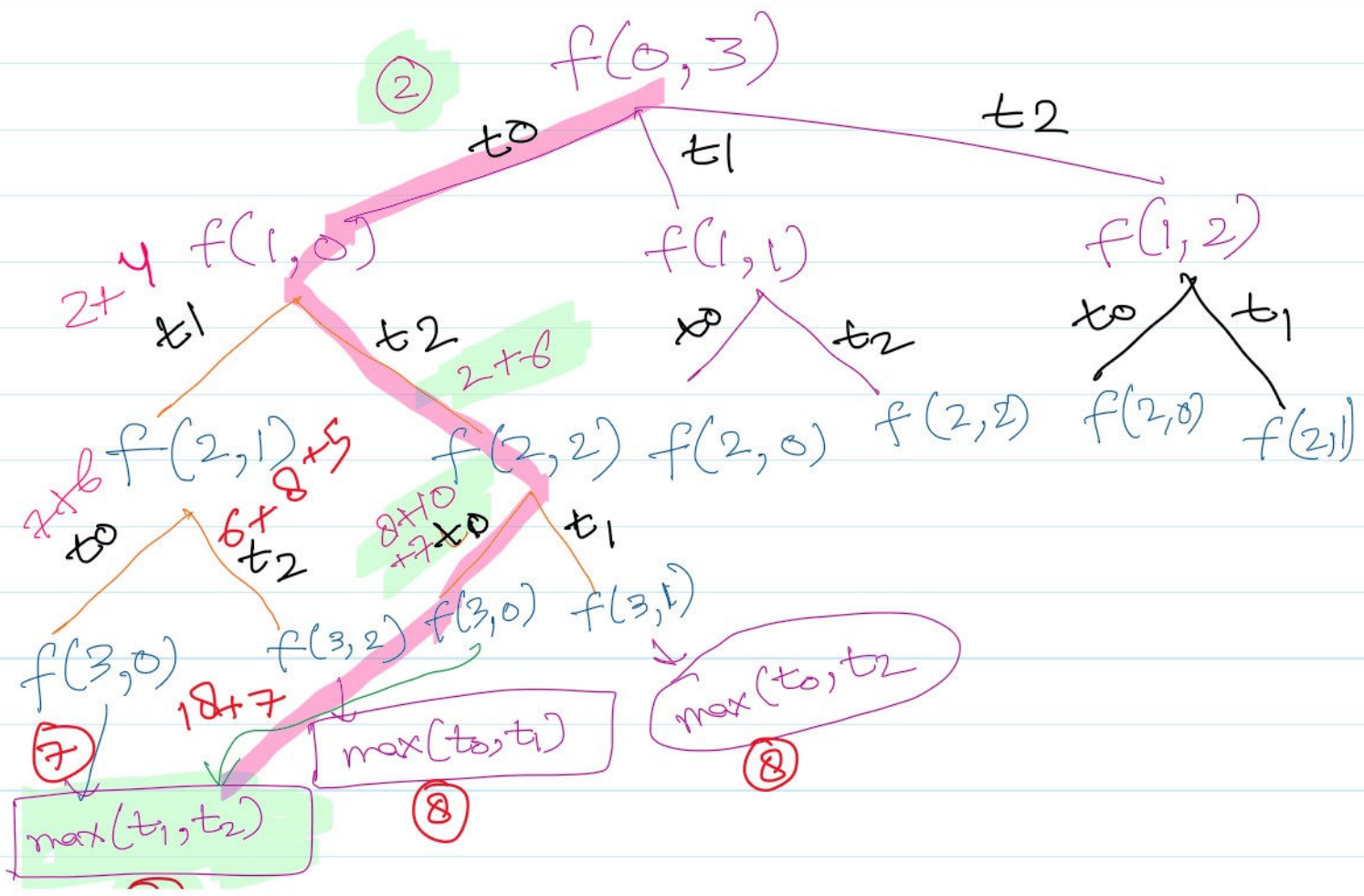
DP → Memoization.
f(day, lasttask) 3
{ if (ind = n-1)

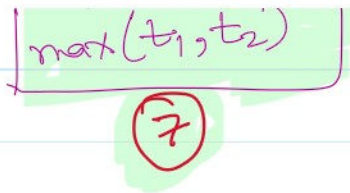
```
if(ind = n-1)
{     int maxi=0;
      for(int i=0; i<3; i++)
          if(i != lasttask)
            maxi = max(maxi, task[ind][i]);
      return maxi;
}
int maxi=0;
for(int i=0; i<3; i++)
{ if(i != lasttask)
    points = task[day][i] + f(day+1, i   )
    maxi = max(maxi, points);
} return maxi;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

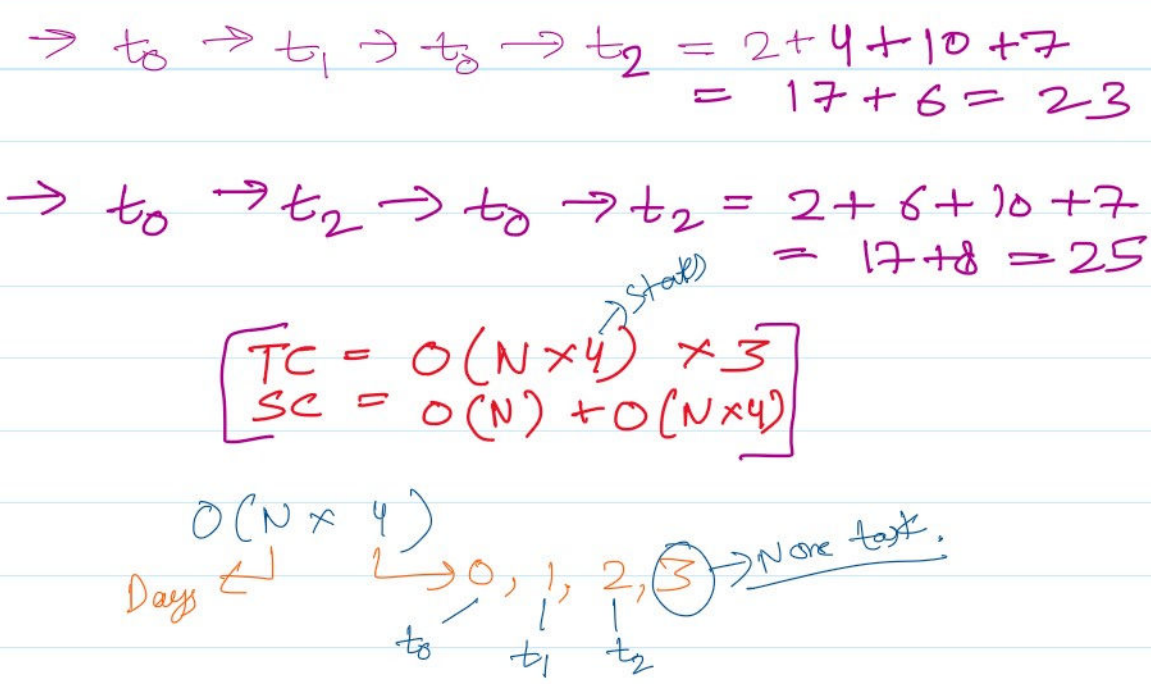
n = 4

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:

```
1 int calcpoints(int currday, int lasttask,int lastday,vector<vector<int>>&
  dp, vector<vector<int>> &points)
2 {   if(currday == lastday) {
3     int maxi = 0;
4     for(int i =0;i<=2;i++){
5         if(i!=lasttask)
6             maxi = max(maxi,points[currday][i]);
7       }
8     return maxi;}
9
10    if(dp[currday][lasttask]!=-1)
11        return dp[currday][lasttask];
12     int maxi  =0;
13     for(int i =0;i<=2;i++){
14        if(i!=lasttask){
15            int reward = points[currday][i] +
  calcpoints(currday+1,i,lastday,dp,points);
16            maxi = max(maxi, reward);}
17     }
18    return dp[currday][lasttask] = maxi;
19 }
20
21    int ninjaTraining(int n, vector<vector<int>> &points)
22 {  vector<vector<int>> dp(n,vector<int>(4,-1));
23   return calcpoints(0,3,n-1,dp,points);
24 }
```

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

$SC = O(N \times 4)$

TC explained?

	t_0	T_1	T_2	T_3
d_0	d_1t_0 d_1t_1 d_1t_2	d_1t_0 d_1t_1 d_1t_2	d_1t_0 d_1t_1 d_1t_2	Final answer
d_1	d_2t_0 d_2t_1 d_2t_2	d_2t_0 d_2t_1 d_2t_2	d_2t_0 d_2t_1 d_2t_2	-1
d_2	d_3t_0 d_3t_1 d_3t_2	d_3t_0 d_3t_1 d_3t_2	d_3t_0 d_3t_1 d_3t_2	-1
d_3	$\max(t_0, t_1, t_2)$			-1

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

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

	t_0	T_1	T_2	T_3
d_0	t_0+ d_1t_1 d_1t_2	d_1t_0 t_1+ d_1t_2	d_1t_0 d_1t_1 t_2+	Final answer
d_1	t_0+ d_2t_1 d_2t_2	d_2t_0 t_1+ d_2t_2	d_2t_0 d_2t_1 t_2+	$=1$
d_2	t_0+ d_3t_1 d_3t_2	d_3t_0 t_1+ d_3t_2	d_3t_0 d_3t_1 t_2+	$=1$
d_3	Some value	Some value	Some value	$=1$

```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>> &points)
{  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[i];
           temp[lasttask] = max(temp[lasttask], reward);
        }
     }
    }
    prev=temp;
  }
  return prev[3];
}
```

	T_0	T_1	T_2	T_3
d_0	$\max(arr[d_0][T_0] + dp[d_1][T_1], arr[d_0][T_2] + dp[d_1][T_2])$	$\max(arr[d_0][T_1] + dp[d_1][T_1], arr[d_0][T_2] + dp[d_1][T_2])$	$\max(arr[d_0][T_0] + dp[d_1][T_0], arr[d_0][T_1] + dp[d_1][T_1])$	$\max(d_0t_0, d_0t_1, d_0t_2)$
d_1	$\max(arr[d_1][T_0] + dp[d_2][T_1], arr[d_1][T_2] + dp[d_2][T_2])$	$\max(arr[d_1][T_1] + dp[d_2][T_0], arr[d_1][T_2] + dp[d_2][T_2])$	$\max(arr[d_1][T_0] + dp[d_2][T_0], arr[d_1][T_1] + dp[d_2][T_1])$	$\max(d_1t_0, d_1t_1, d_1t_2)$
d_2	$\max(arr[d_2][t_1] + dp[d_3][t_1], arr[d_2][t_2] + dp[d_3][t_2])$	$\max(arr[d_2][t_0] + dp[d_3][t_0], arr[d_2][t_2] + dp[d_3][t_2])$	$\max(arr[d_2][t_0] + dp[d_3][t_0], arr[d_2][t_1] + dp[d_3][t_1])$	$\max(d_2t_0, d_2t_1, d_2t_2)$
d_3	$\max(t_1, t_2)$	$\max(t_0, t_2)$	$\max(t_0, t_1)$	$\max(t_0, t_1, t_2)$

$n = 4$

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
d_1	23	23	21	23
d_2	14	17	17	17
d_3	7	8	8	8

LinkedIn/kapilyadav22

*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 to move 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 time.

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

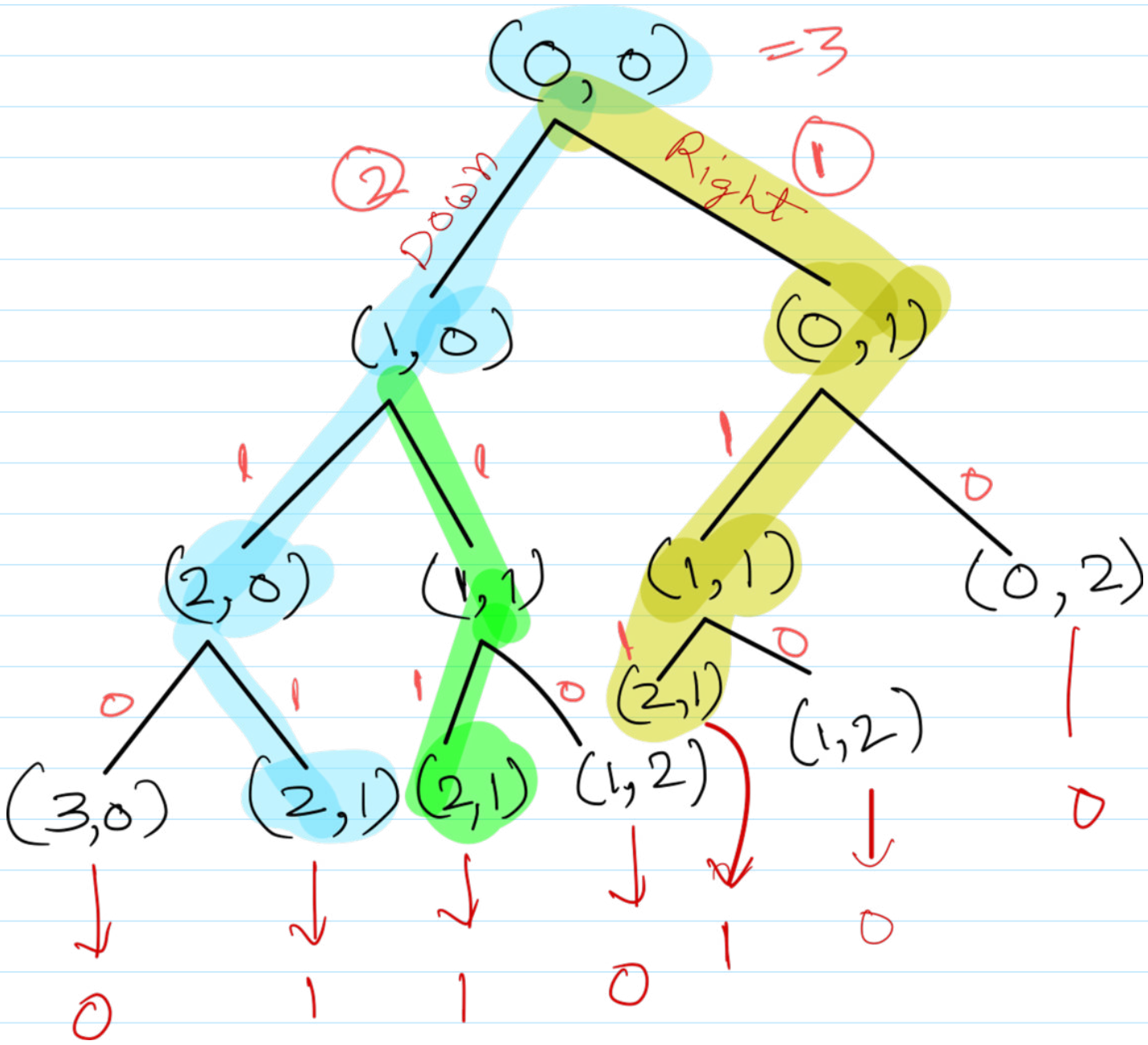
The test cases are generated so that the answer will be less than or equal to `2 * 10^9`.

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:

```cpp
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;
    }


    int uniquePaths(int m, int n) {
     vector<vector<int>> dp(m, vector<int> (n,-1));
       return totalpaths(0,0,m-1,n-1,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:

- 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)

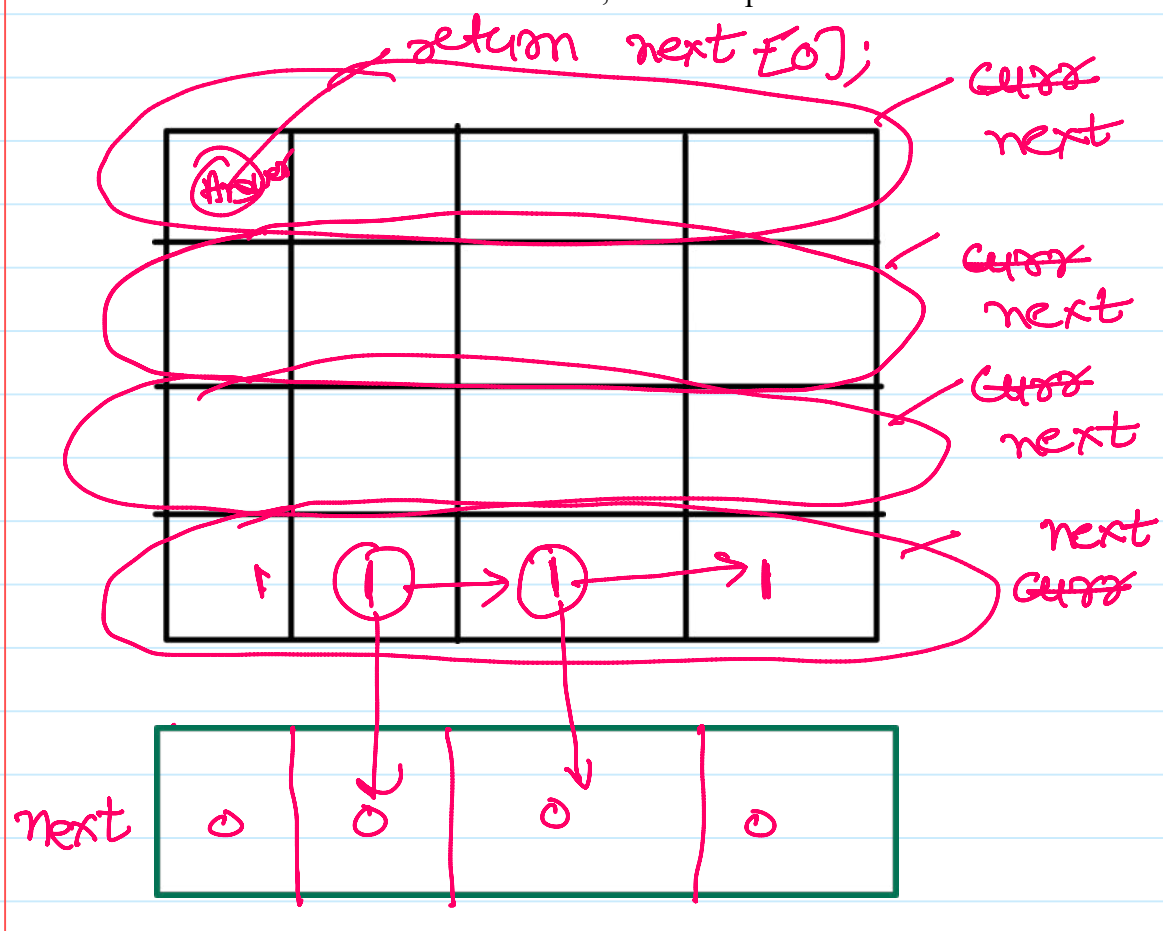
OR

```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.



→ If we observe, to find the no of paths from every index, we will need the paths from Down & right.

Down & right.

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

```
1 int uniquePaths(int m, int n) {
2       vector<int> next(n,0);
3       for(int CurrRow=m-1;CurrRow>=0;CurrRow--)
4       {   vector<int> curr(n,0);
5           for(int CurrCol=n-1;CurrCol>=0;CurrCol--)
6           { if(CurrRow==m-1 && CurrCol==n-1)
7                curr[CurrCol] =1;
8         else{   int movedown=0,moveright=0;
9           if(CurrRow<m-1)
10               movedown = next[CurrCol];
11          if(CurrCol<n-1)
12               moveright = curr[CurrCol+1];
13          curr[CurrCol] = movedown + moveright;
14         }
15        }
16          next= curr;
17      }
18       return next[0];
19    }
```

OR

```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)

LinkedIn/kapilyadav22

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++)
        {
            totalpaths= totalpaths*(Number_of_steps-r+i)/i;
        }
        return (int) totalpaths;
    }
```

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

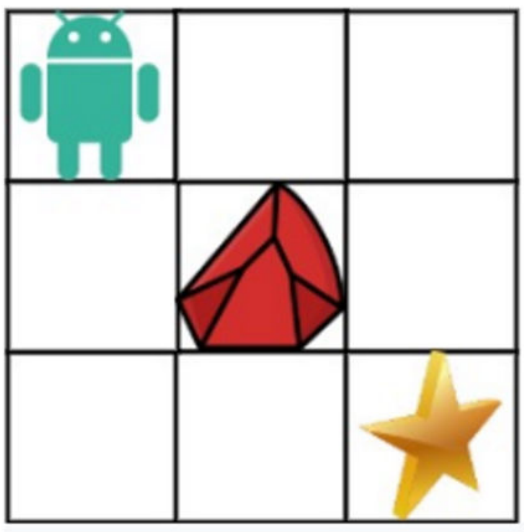
You are given an `m x n` integer array `grid`. There is a robot initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m-1][n-1]`). The robot can only move either down or right at any point in time.

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

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

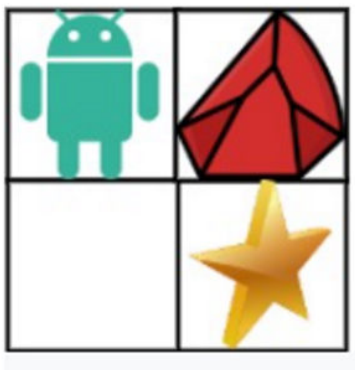
The testcases are generated so that the answer will be less than or equal to `2 * 10^9`.

Example 1:



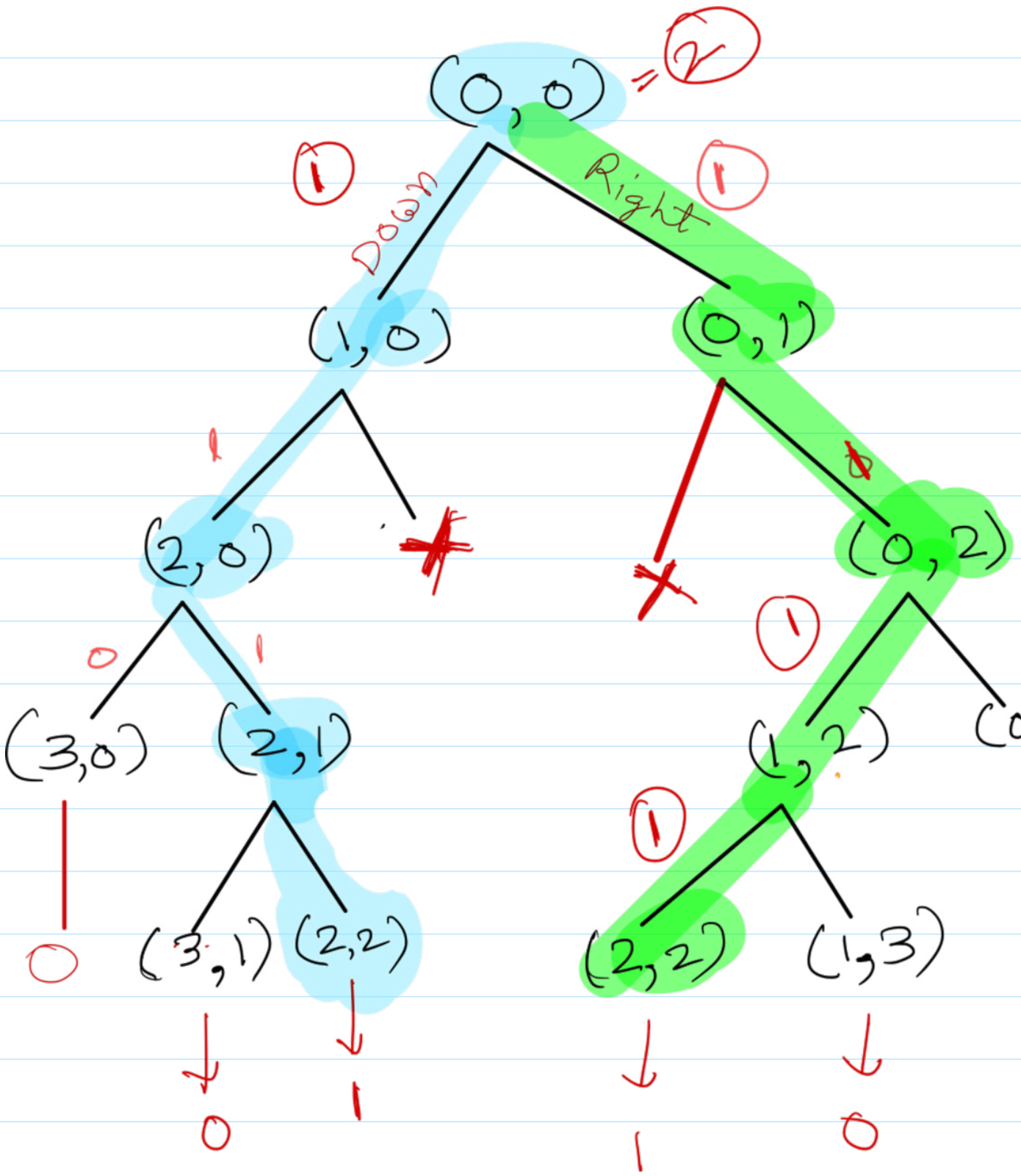
```
Input: obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]
Output: 2
Explanation: There is one obstacle in the middle of the 3x3 grid above.
There are two ways to reach the bottom-right corner:
1. Right -> Right -> Down -> Down
2. Down -> Down -> Right -> Right
```

Example 2:



```
Input: obstacleGrid = [[0,1],[0,0]]
Output: 1
```

Ex → [[0,0,0],[0,1,0],[0,0,0]]



MEMOIZATION:

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

     if(CurrRow == targetRow && CurrCol == targetCol && obstacleGrid[CurrRow][CurrCol]!=1)
         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:

```cpp
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][CurrCol]==0)
              dp[CurrRow][CurrCol]=1;
          else if(obstacleGrid[CurrRow][CurrCol]==1)
              dp[CurrRow][CurrCol]==0;
         else { double movedown =0;
           double 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 (int)dp[0][0];
}
```

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

SPACE OPTIMIZATION:

```cpp
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

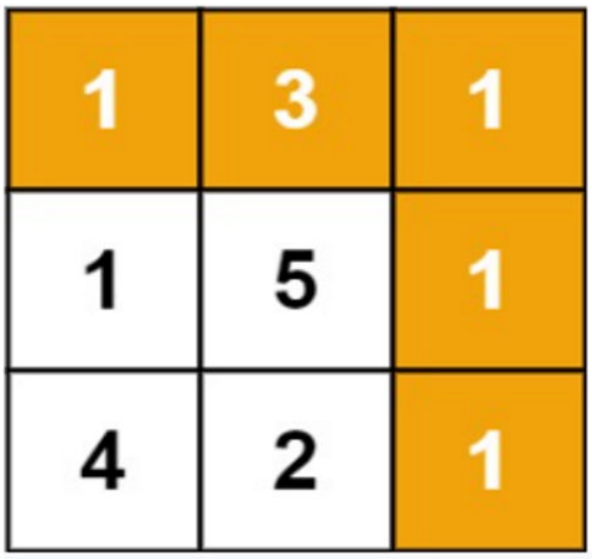
64. Minimum Path Sum

Medium 7583 106 Add to List Share

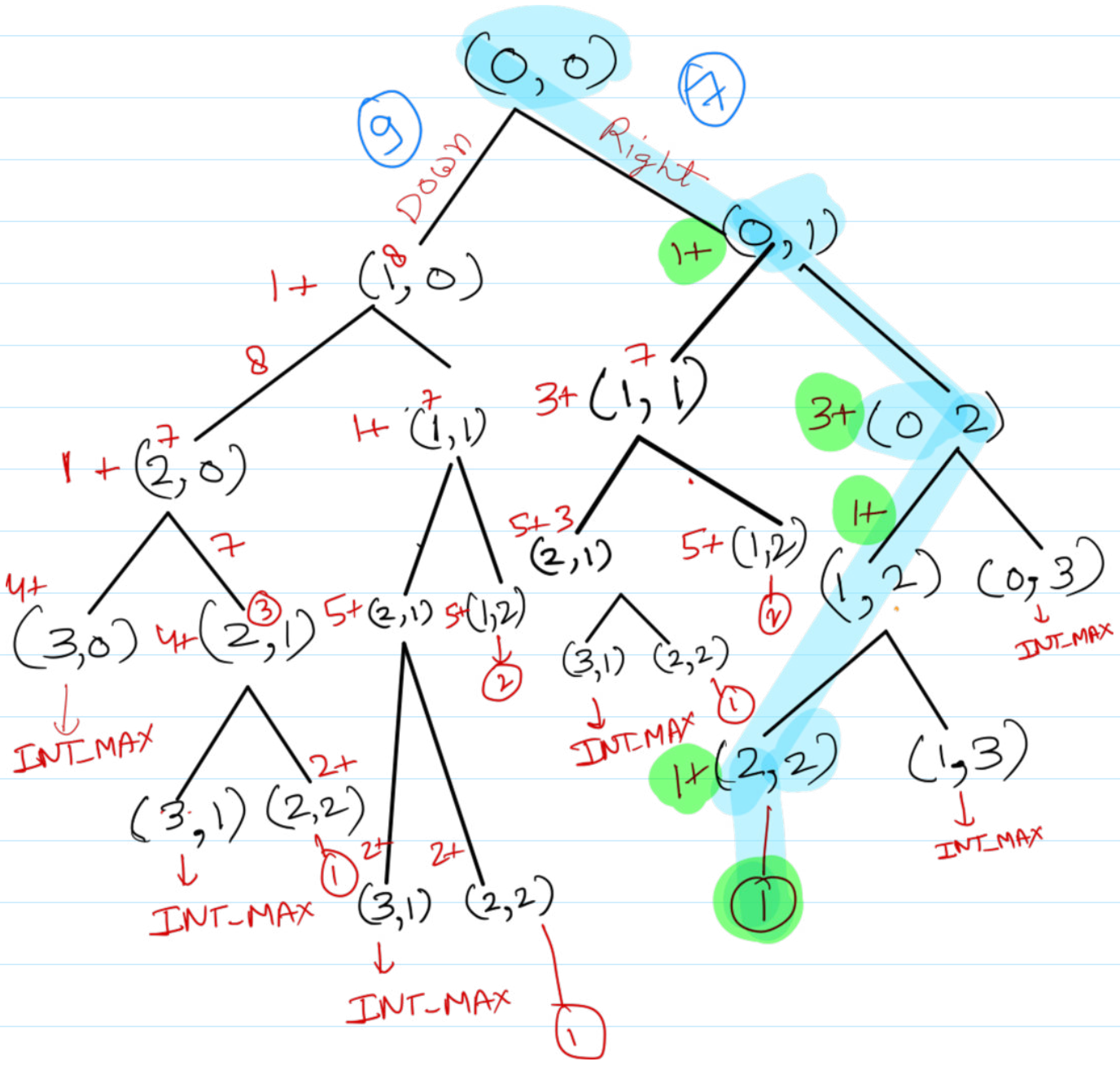
Given a `m x n` `grid` filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

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\:

```cpp
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,grid,dp);
       int moveright = findpath(CurrRow,CurrCol+1,TargetRow,TargetCol,grid,dp);
    return dp[CurrRow][CurrCol] =  grid[CurrRow][CurrCol] + min(movedown,moveright);
    }


    int minPathSum(vector<vector<int>>& grid) {
       int rowsize = grid.size();
        int colsize = grid[0].size();
        vector<vector<int>> dp(rowsize,vector<int> (colsize,-1));
       return findpath(0,0,rowsize-1,colsize-1,grid,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:

```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,moveright);

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

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

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,moveright);
           }
           }
        next = curr;
       }
      return next[0];
    }
```

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

LinkedIn/kapilyadav22

LECTURE -11 TRIANGLE (DP ON GRIDS)

19 June 2022 18:40

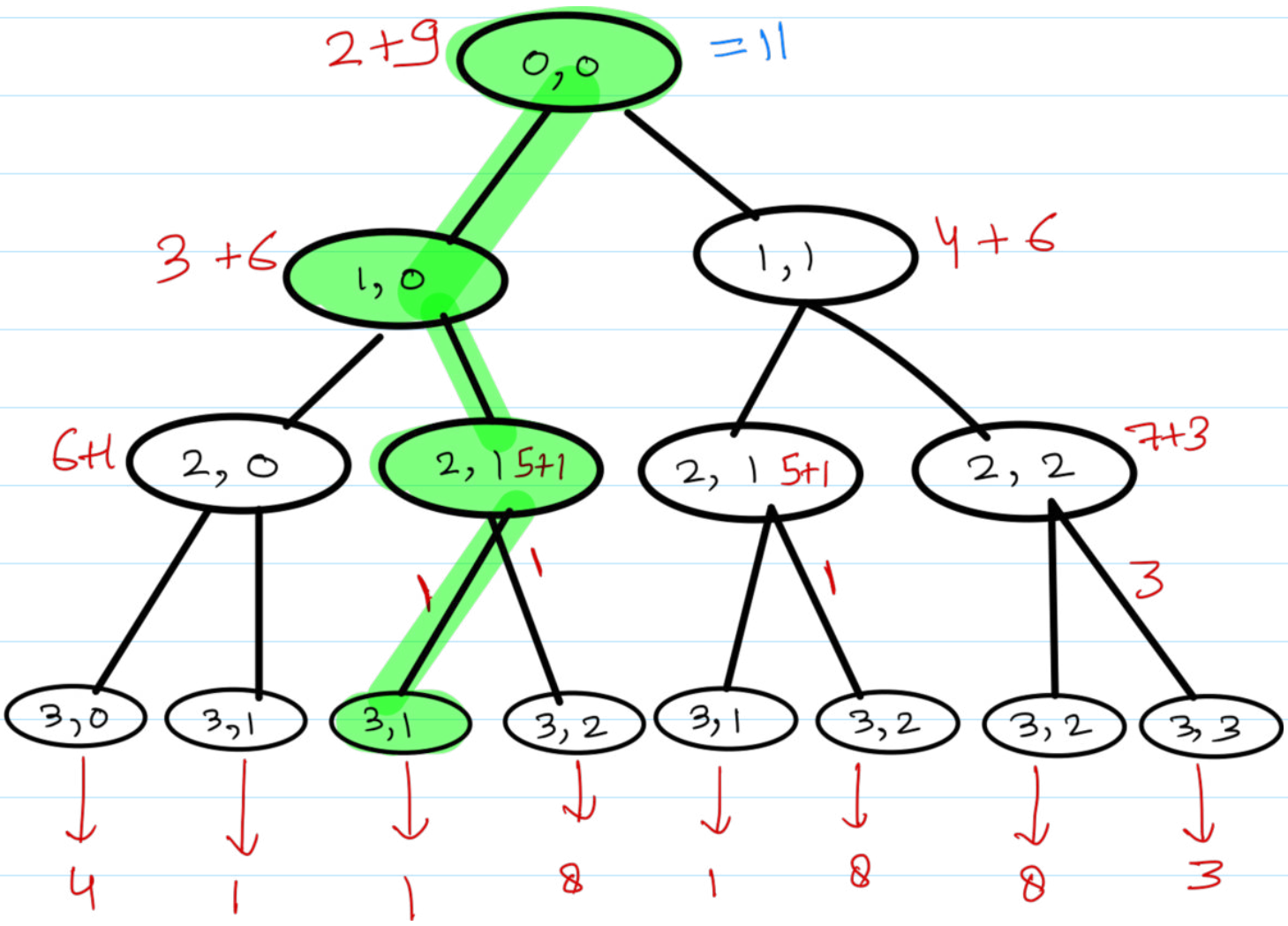
Given a `triangle` array, return *the minimum path sum from top to bottom.*

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

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

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)

LinkedIn/kapilyadav22

LECTURE-12 Minimum Falling Path Sum

19 June 2022 21:11

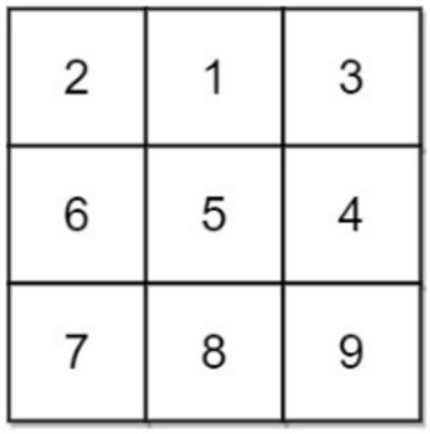
931. Minimum Falling Path Sum

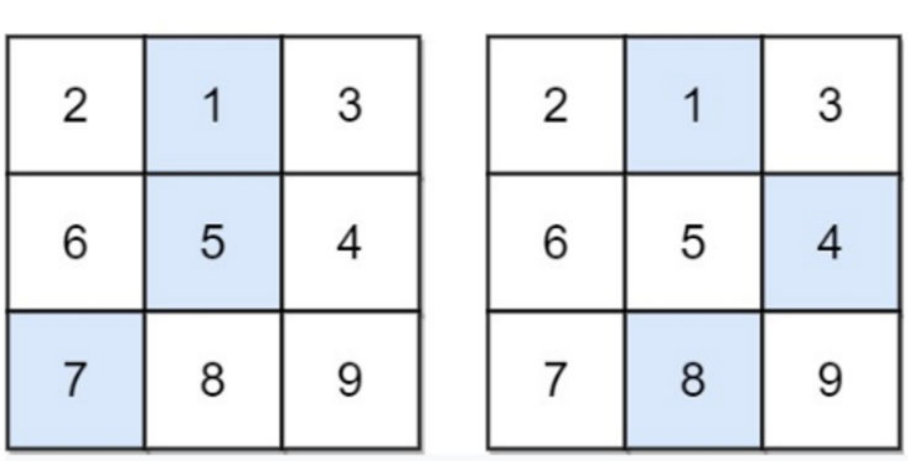
Medium 2493 95 Add to List Share

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

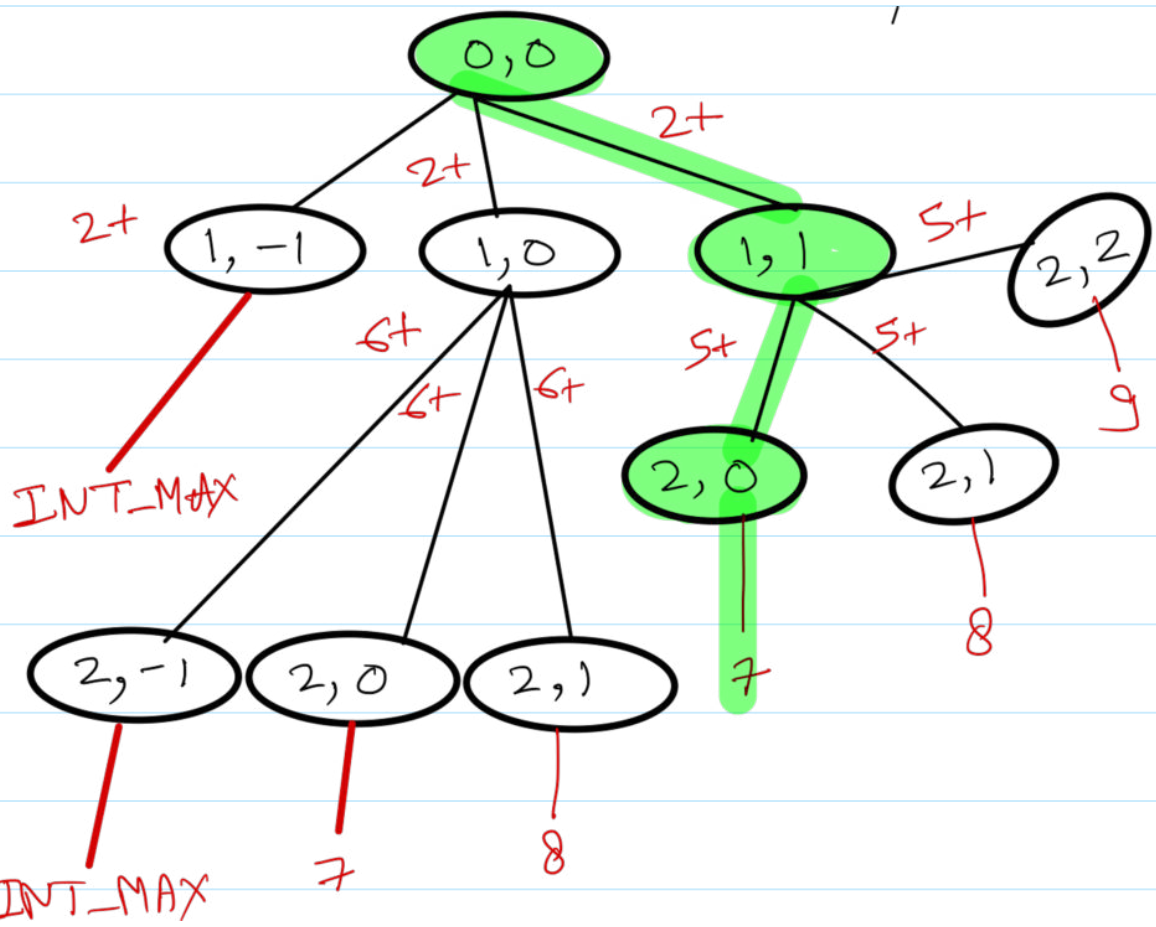
A **falling path** starts at any element in the first row and chooses the element in the next row that is either directly below or diagonally left/right. Specifically, the next element from position `(row, col)` will be `(row + 1, col - 1)`, `(row + 1, col)`, or `(row + 1, col + 1)`.

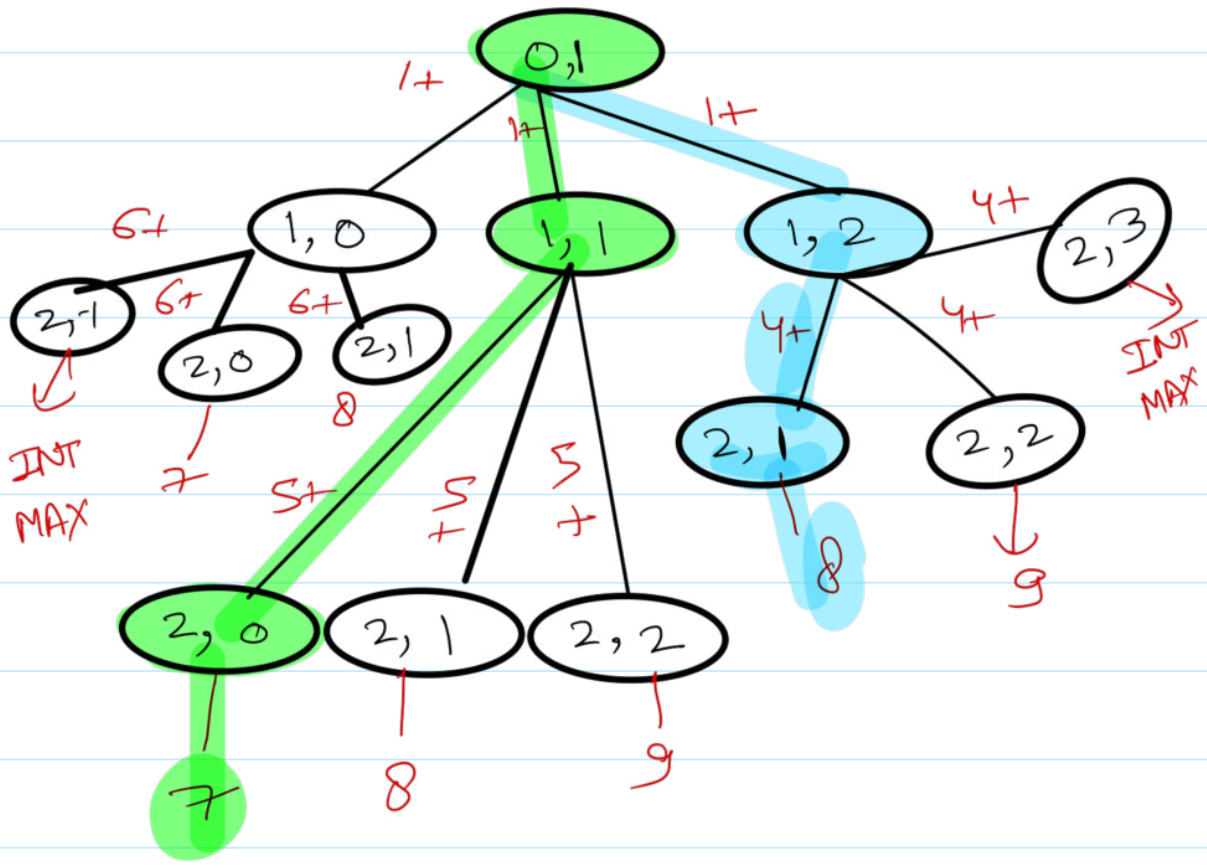
Example 1:

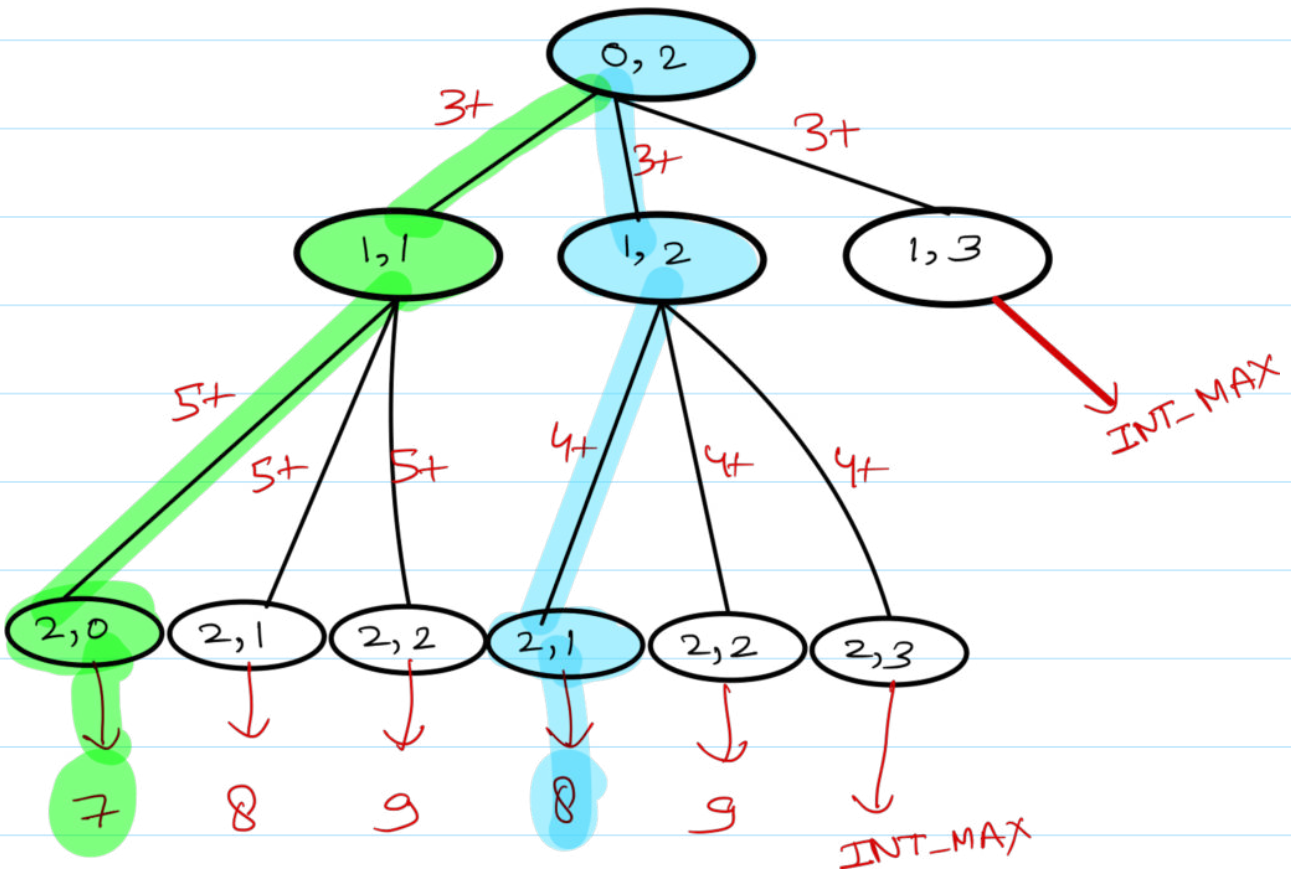





```
Input: matrix = [[2,1,3],[6,5,4],[7,8,9]]
Output: 13
Explanation: There are two falling paths with a minimum sum as shown.
```


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+
1,0
1,1
1,2
6+
4+
2,3
2,-1
6+
6+
2,0
2,1
4+
4+
INT
MAX
INT
MAX
7
8
2,1
2,2
5+
5+
5+
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();
     vector<vector<int>> dp(n,vector<int> (n,-1));
        int mini = INT_MAX;
        for(int i =0;i<n;i++)
        { mini = min(mini, findmin(0,i,matrix,dp));
        }
        return mini;
    }
```

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

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

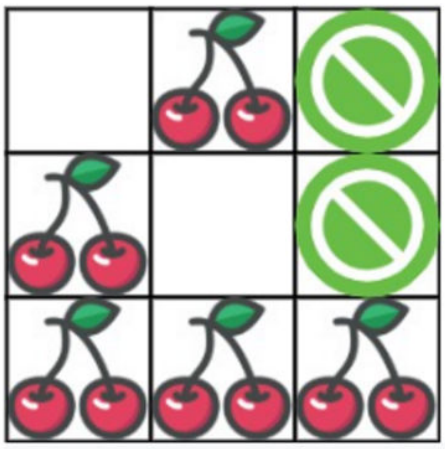
You are given an `n x n` `grid` representing a field of cherries, each cell is one of three possible integers.

- `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:



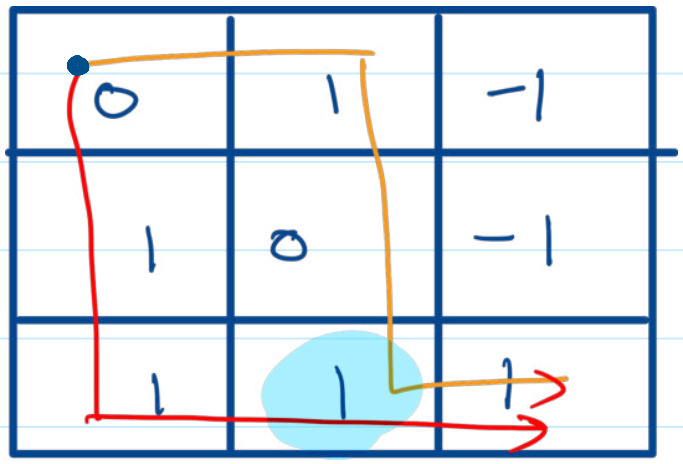
```
Input: grid = [[0,1,-1],[1,0,-1],[1,1,1]]
Output: 5
Explanation: The player started at (0, 0) and went down, down, right right to reach (2, 2).
4 cherries were picked up during this single trip, and the matrix becomes [[0,1,-1],[0,0,-1],[0,0,0]].
Then, the player went left, up, up, left to return 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 back to initial position, after collecting the cherris

→ OR we can say we can take two people who reach to end and collect the cherris



→ 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 once
Ex - at index (2, 1).

→ If both are at different index, collect cherries from Both 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] == -1)
      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 dp[50][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] == -1)
    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, $c_1 = c_1 + 1$ and another person move down, $r_2 = r_2 + 1$ 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][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] == -1)
      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%

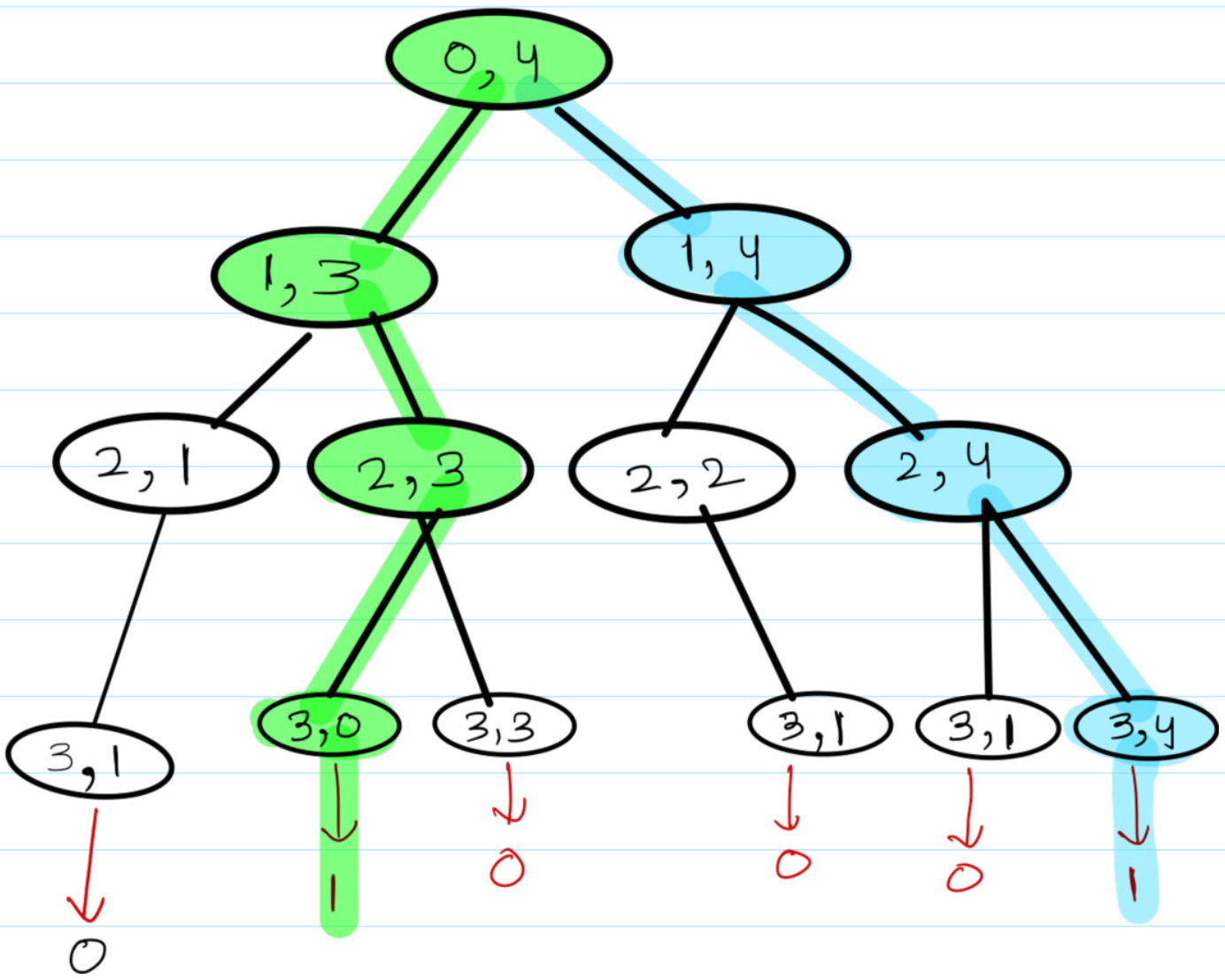
Problem Statement Suggest Edit

You are given an array/list ‘ARR’ of ‘N’ positive integers and an integer ‘K’. Your task is to 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 false.

For Example :

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



MEMOIZATION:

```cpp
bool findpartition(int currind, vector<int>& nums, vector<vector<int>> & 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[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--)
            { 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][k];
}
```

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, therefore for the target value equal to arr[n-1], only cell with that target will be true, so explicitly set dp[n-1][nums[n-1]] =true, (dp[n-1][nums[n-1]] means that we are considering the last element of the array with the target equal to the last element itself). Please note that it can happen that arr[n-1]>target, so we first check it: 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];
}
```

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

Or

- We already know that, if we need a target of 0 at any position, we can return 1, because that can be our one path. So, it's a base condition, we cam run our target loop till target>=1. and put curr[0]=1.

```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 partitioned into two 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].
```

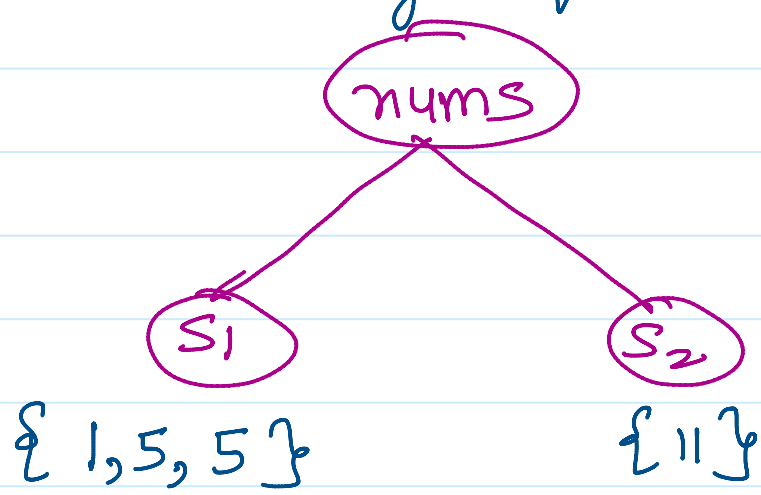
Example 2:

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

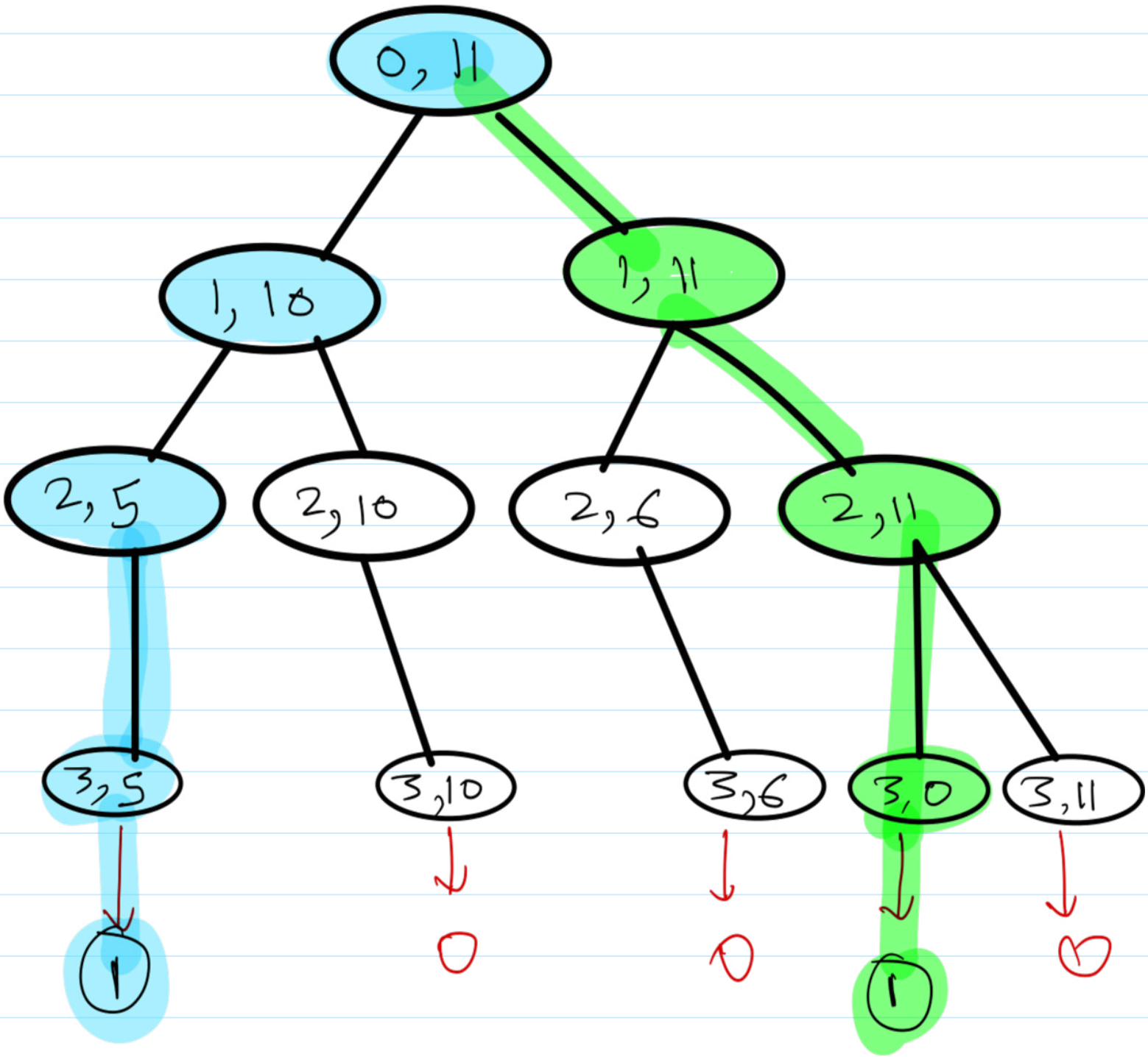
Same as Lecture 14. We just need to check if total sum is odd, we cannot partition it, return false. Else check for the sum of totalsum/2.

ex → {1, 5, 11, 5}

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



→ so we need to find, whether there is a subset of sum = totalsum/2 or not.


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

MEMOIZATION:

```cpp
bool findpartition(int currind, vector<int>& nums, vector<vector<int>>
& 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[currind]);
     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

LinkedIn/kapilyadav22

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

20 June 2022 13:31

Problem Statement Suggest Edit

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

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

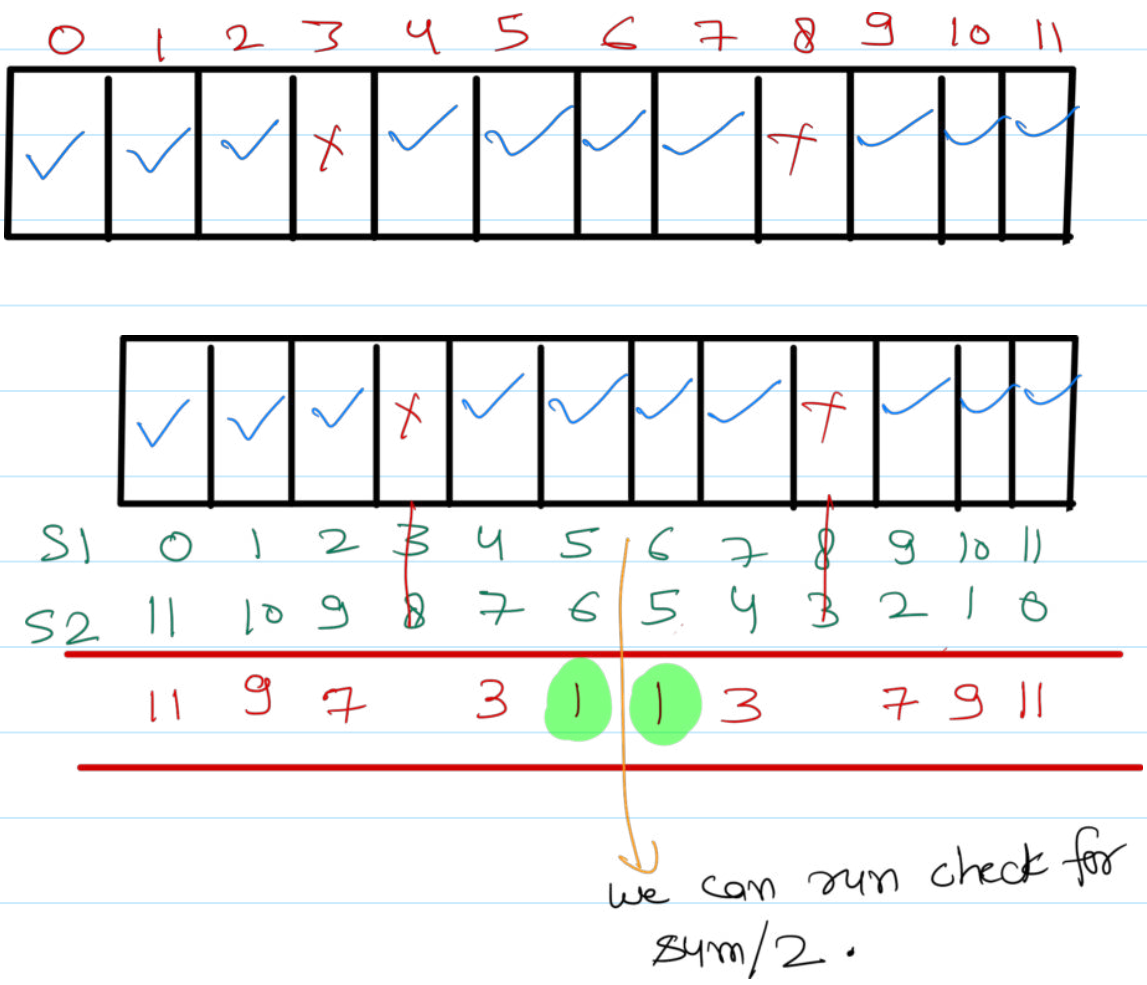
Note:

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 not possible.



```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[currind]];
                bool notconsider = dp[currind+1][target];
                dp[currind][target] = consider + notconsider;
            }
       }

 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:

```cpp
int minSubsetSumDifference(vector<int>& nums, int n)
{    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[currind]];
                 bool notconsider = next[target];
                 curr[target] = consider + notconsider;
             }
          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));
       }
  }
return mini;
 }
```

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

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.

- If our nums[n-1] ==0 and target ==0, then either we take it or we don't take it, we will get a path from each recursive call, so return 2.
- If our target is 0, but the nums[n-1] doesn't equal to 0, then return 1, because, the upper path is covering the target.
- If our target == nums[n-1] then also, return 1, because adding nums[n-1] will equal to our target.
- If target!= nums[n-1], then we don't get a path, return 0.

Memoization:

```cpp
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

LinkedIn/kapilyadav22

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:

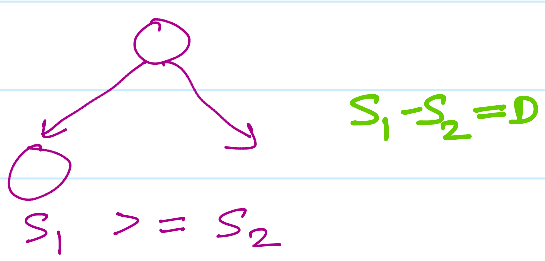
```
1) P1_S1 != P2_S1 i.e, at least one of the elements of P1_S1 is different from
P2_S2.
2) P1_S1 == P2_S2, but the indices set represented by P1_S1 is not equal to the
indices set of P2_S2. Here, the indices set of P1_S1 is formed by taking the
indices of the elements from which the subset is formed.
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
These two partitions are different because, in the 1st partition, S1 contains 5
from index 0, and in the 2nd partition, S1 contains 5 from index 2.
```

It is same as No of Subsets problem with constraint 0<=nums[i]<=50, so use Lecture-17 updated code(last code)



$S_1 > S_2$,

$S_1 = totalsum - S_2$

$S_1 - S_2 = D$

$totalsum - S_2 - S_2 = D$

$2S_2 = totalsum - D$

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

→ 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);
}
```

LinkedIn/kapilyadav22

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

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

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)
        return 0;
    int tar = (totalsum-d)/2;
     return countways(tar,arr);
}
```

*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 capacity **W** to get the maximum total value in the knapsack. Note that we have only **one quantity of each item**. In other words, given two integer arrays **val[0..N-1]** and **wt[0..N-1]** which represent values and weights associated with **N** items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of **val[]** such that sum of the weights of this subset is smaller than or equal to **W.** You cannot break an item, **either pick the complete 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
```

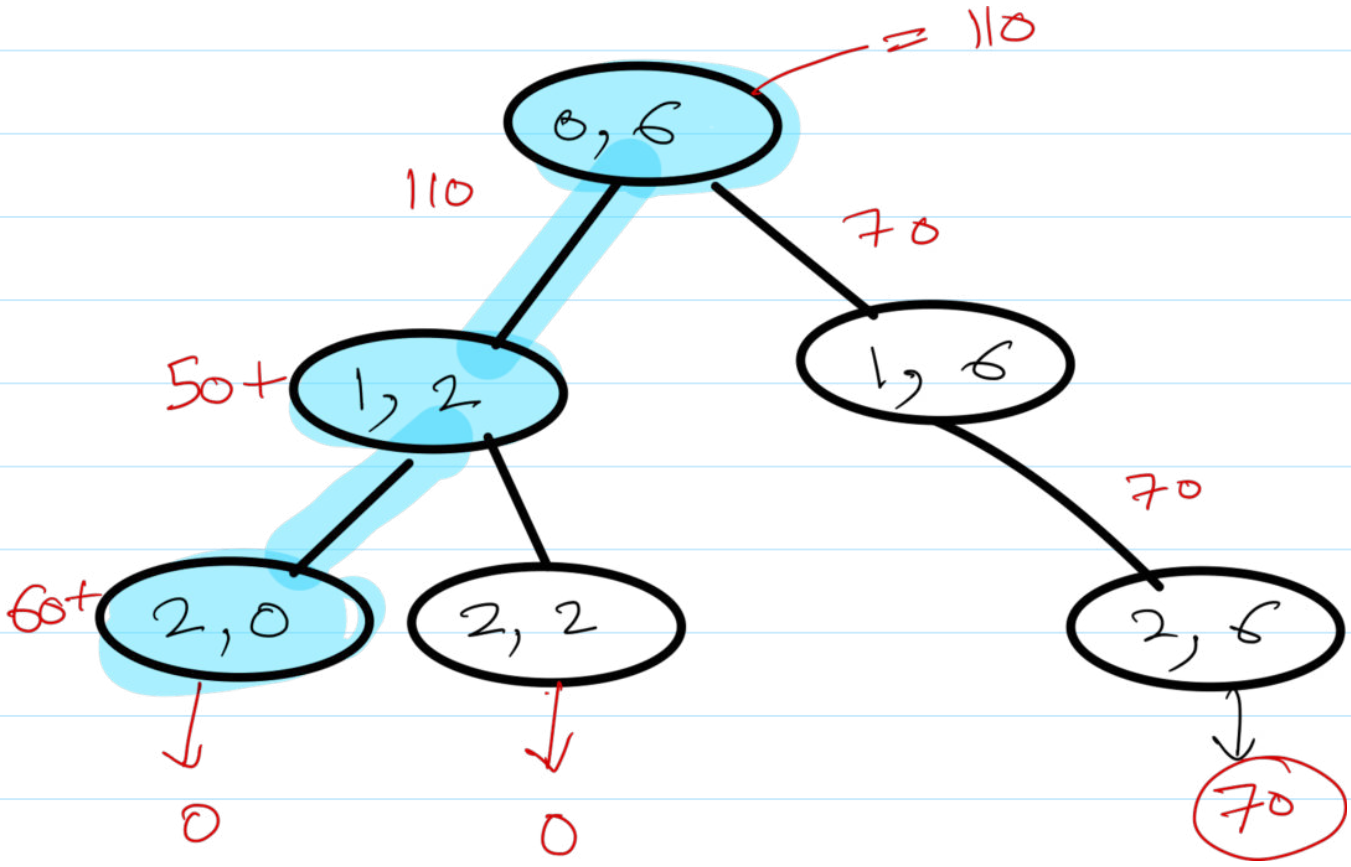
Can we use greedy ?

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 possibilities.
optimal solution will be → 50 + 60 = 110.

→ At every index, either we can consider it or we don't consider it.



→ These can be overlapping subproblems.

MEMOIZATION:

```
1 int findknapsack(int currind, int capacity,vector<int>& wt,vector<int>& profit, int n,
  vector<vector<int>>& dp)
2     {   if(capacity==0)
3             return 0;
4         if(currind>n-1)
5             return 0;
6
7         if(dp[currind][capacity]!=-1)
8          return dp[currind][capacity];
9
10         int notconsider = findknapsack(currind+1,capacity,wt,profit,n,dp);
11         int consider = INT_MIN;
12         if(wt[currind]<=capacity)
13         consider = profit[currind] + findknapsack(currind+1,capacity-wt[currind],wt,profit,n,dp);
14        return dp[currind][capacity] = max(consider,notconsider);
15    }
```

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

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[currind]];
        dp[currind][capacity] = max(consider,notconsider);
   }
 }
   return dp[0][W];
}
```

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

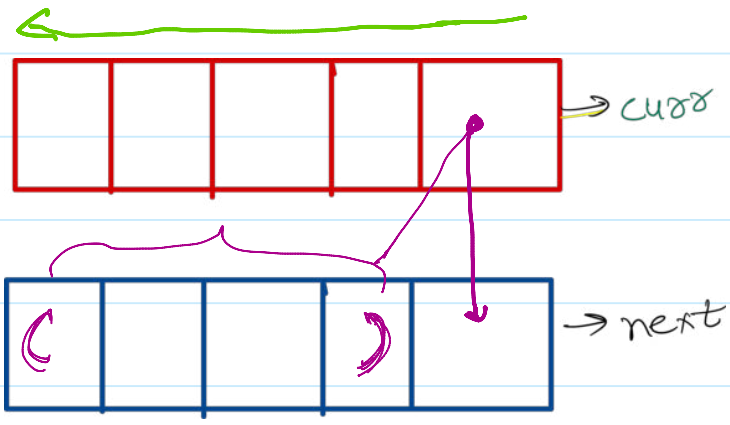
Space Optimization:

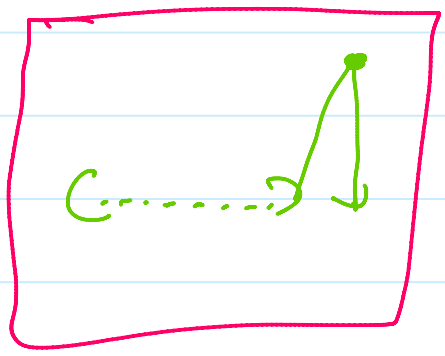
```cpp
int knapsack(vector<int>& wt, vector<int>& profit, int n, int W)
{   vector<int> next(W+1,0);
   for(int i=wt[n-1];i<=W;i++)
   {  next[i]= profit[n-1];
   }
 for(int currind=n-2;currind>=0;currind--)
 {  vector<int> curr(W+1,0);
     for(int capacity=W;capacity>=0;capacity--)
     {   int notconsider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

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

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 from left side, so we can start our loop from right side to access the unupdated values from next array, and use the right part of the array as updated part 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 unupdated values.

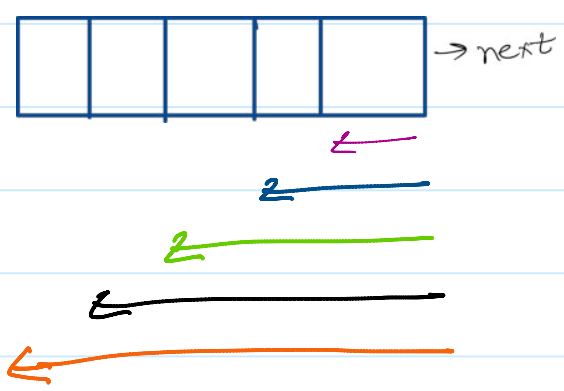




```cpp
int knapsack(vector<int>& wt, vector<int>& profit, int n, int W)
{   vector<int> next(W+1,0);
   for(int i=wt[n-1];i<=W;i++)
   {  next[i]= profit[n-1];
   }
 for(int currind=n-2;currind>=0;currind--)
 {   for(int capacity=W;capacity>=0;capacity--)
     {   int notconsider =next[capacity];
  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

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

LinkedIn/kapilyadav22



*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

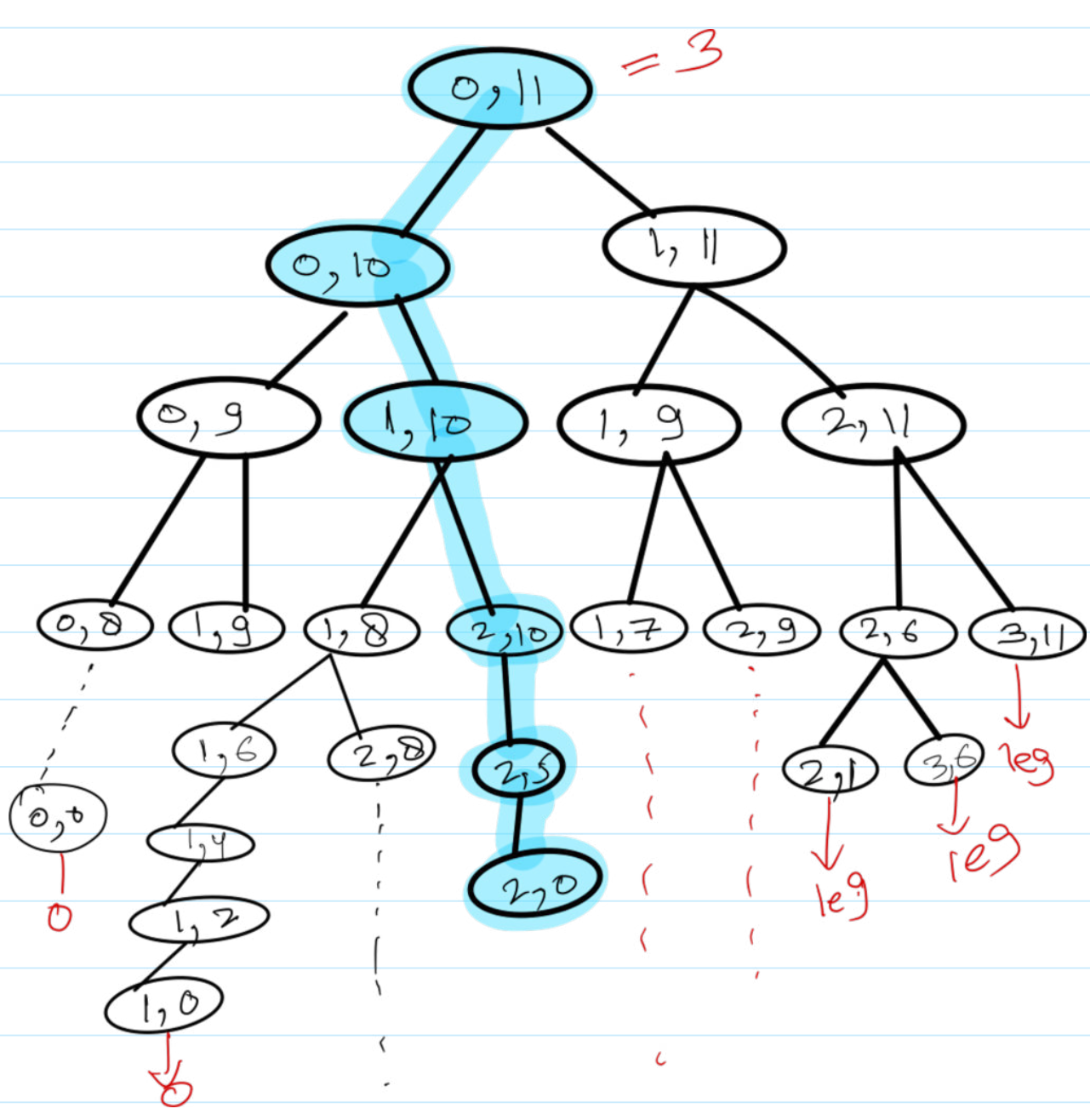
You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any 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 need 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],coins,dp);
    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[currind]];
            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)

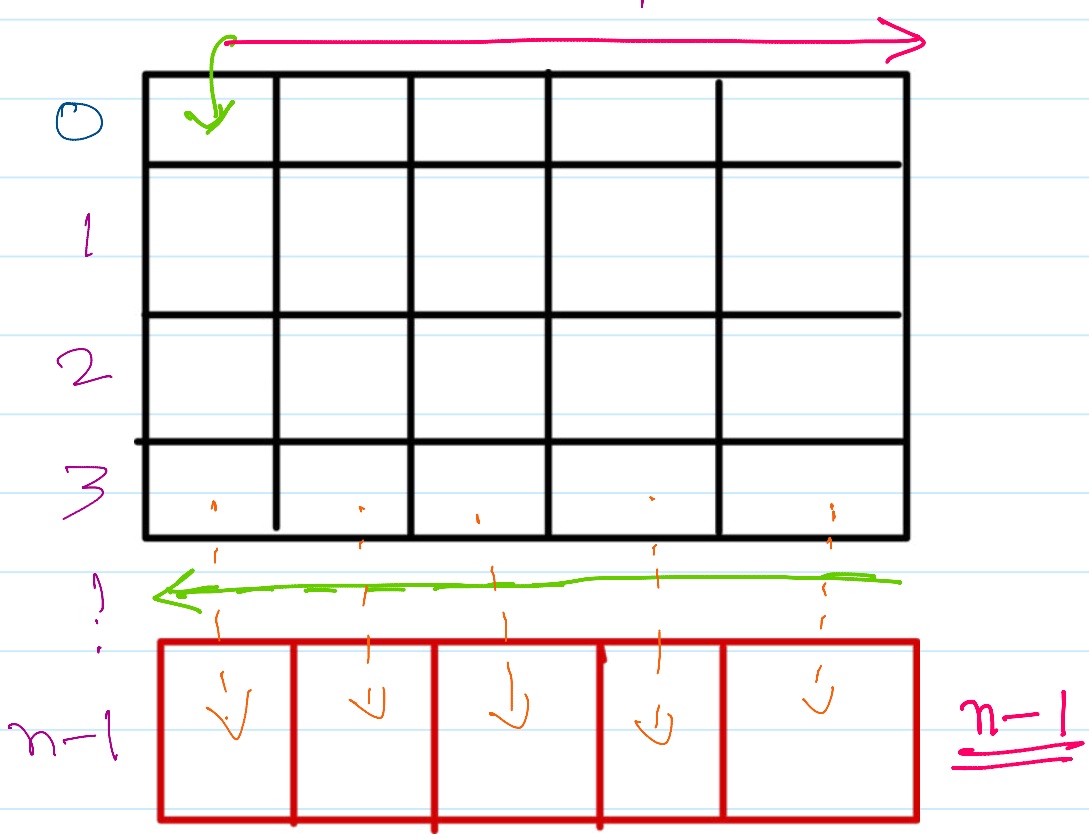
Space Optimization:

```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:


```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--)
        {
            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 concatenate all the integers.

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

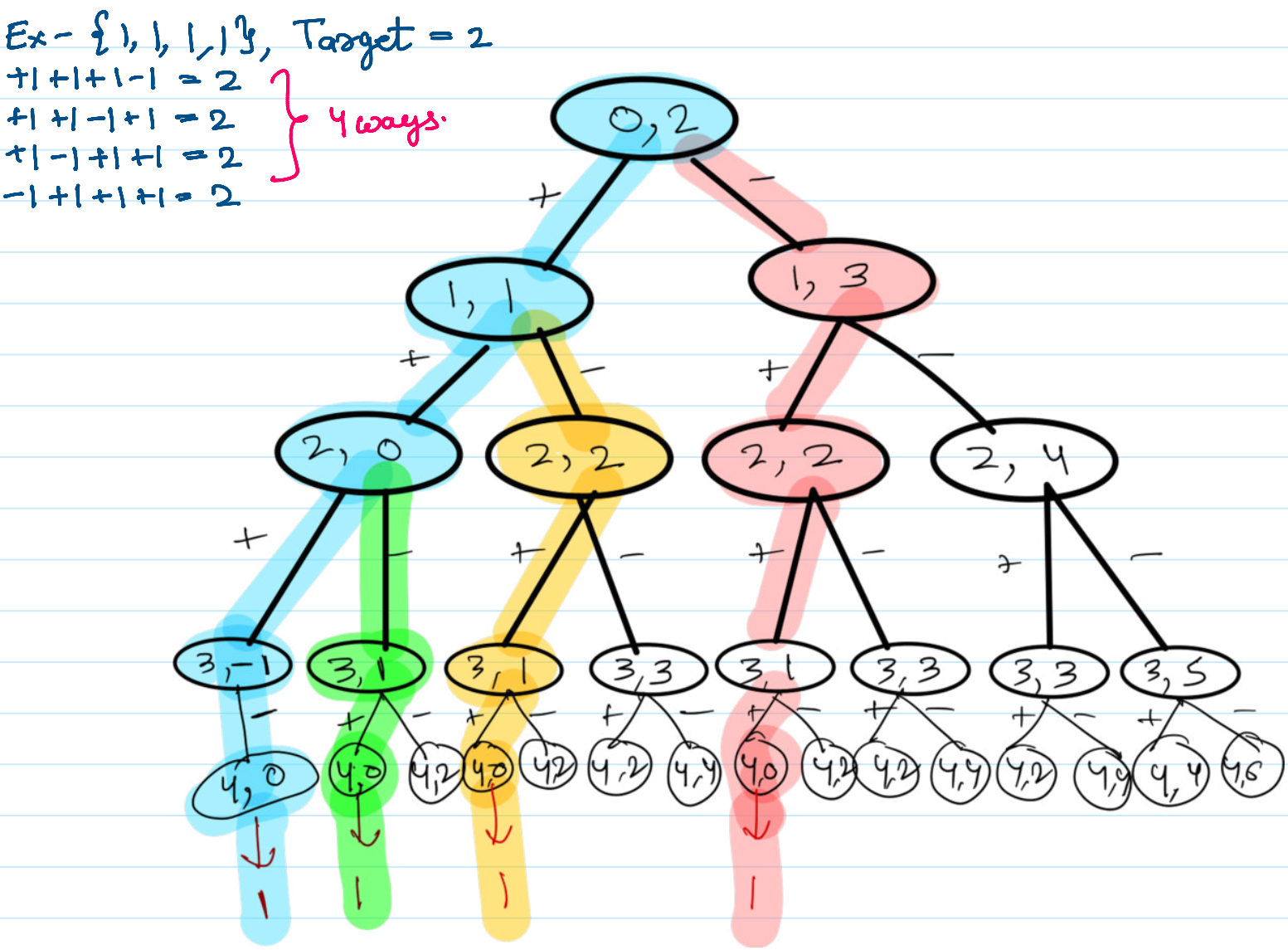
Return the number of different **expressions** that you can build, which evaluates to `target`.

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 will eventually 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>&mp)
 {    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

$\{1+1+1\}_{S1} - \{1\}_{S2} = 2_{D}$

→ It can be solved using count partition with given difference

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;
     return countways(tar,arr);
}

```

LinkedIn/kapilyadav22

TC: O(N*K), **SC:** O(N*K), where K = 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`.

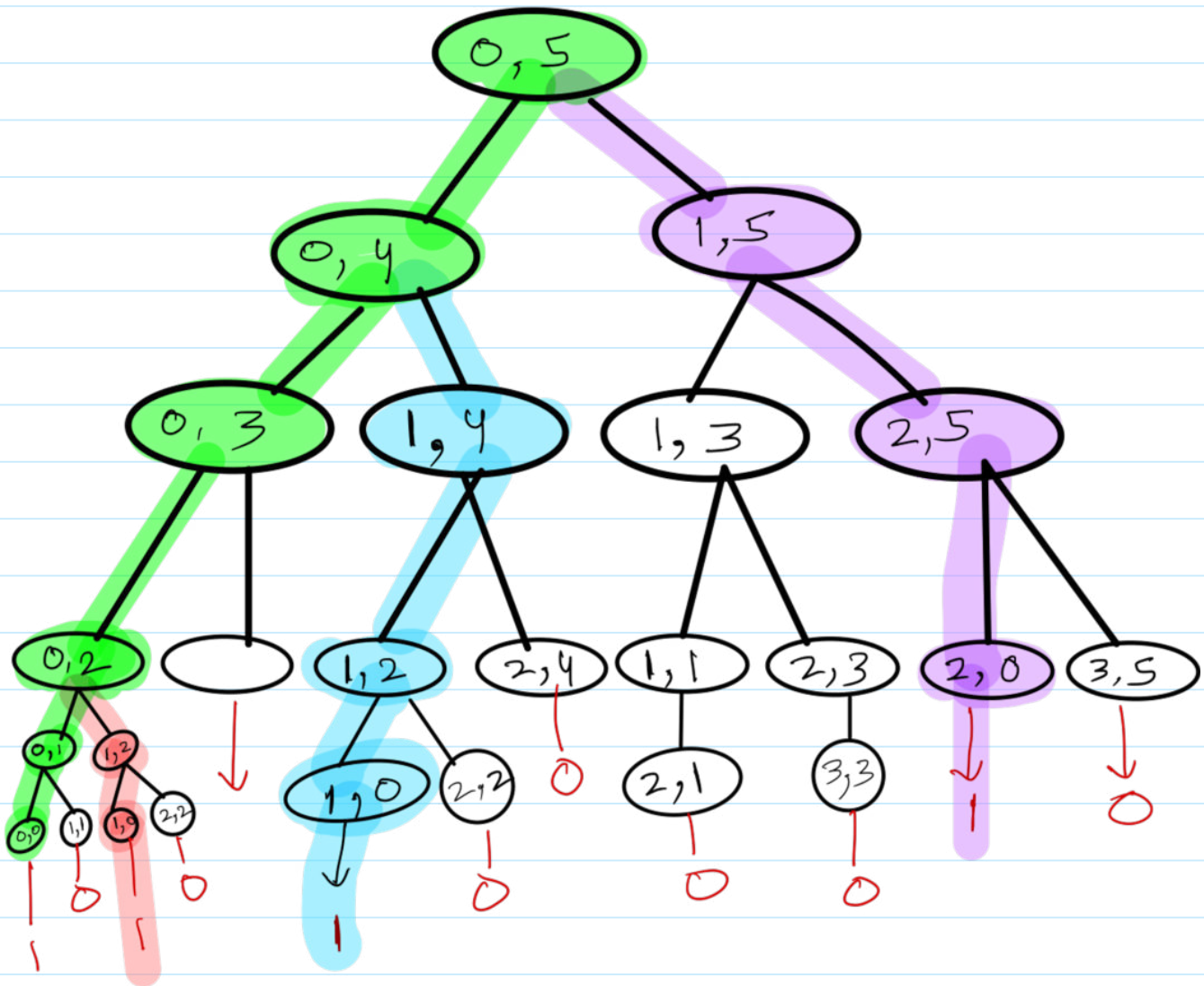
You may assume that you have an infinite number of each kind of coin.

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];
}
```

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

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)
        pick = next[price-denominations[currind]];
      next[price]= pick + notpick;
     }
    }
 return next[value];
}
```

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

LinkedIn/kapilyadav22

LECTURE-23 Unbounded Knapsack

Tuesday, 21 June 2022 6:50 PM

Problem Statement Suggest Edit

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

For Example

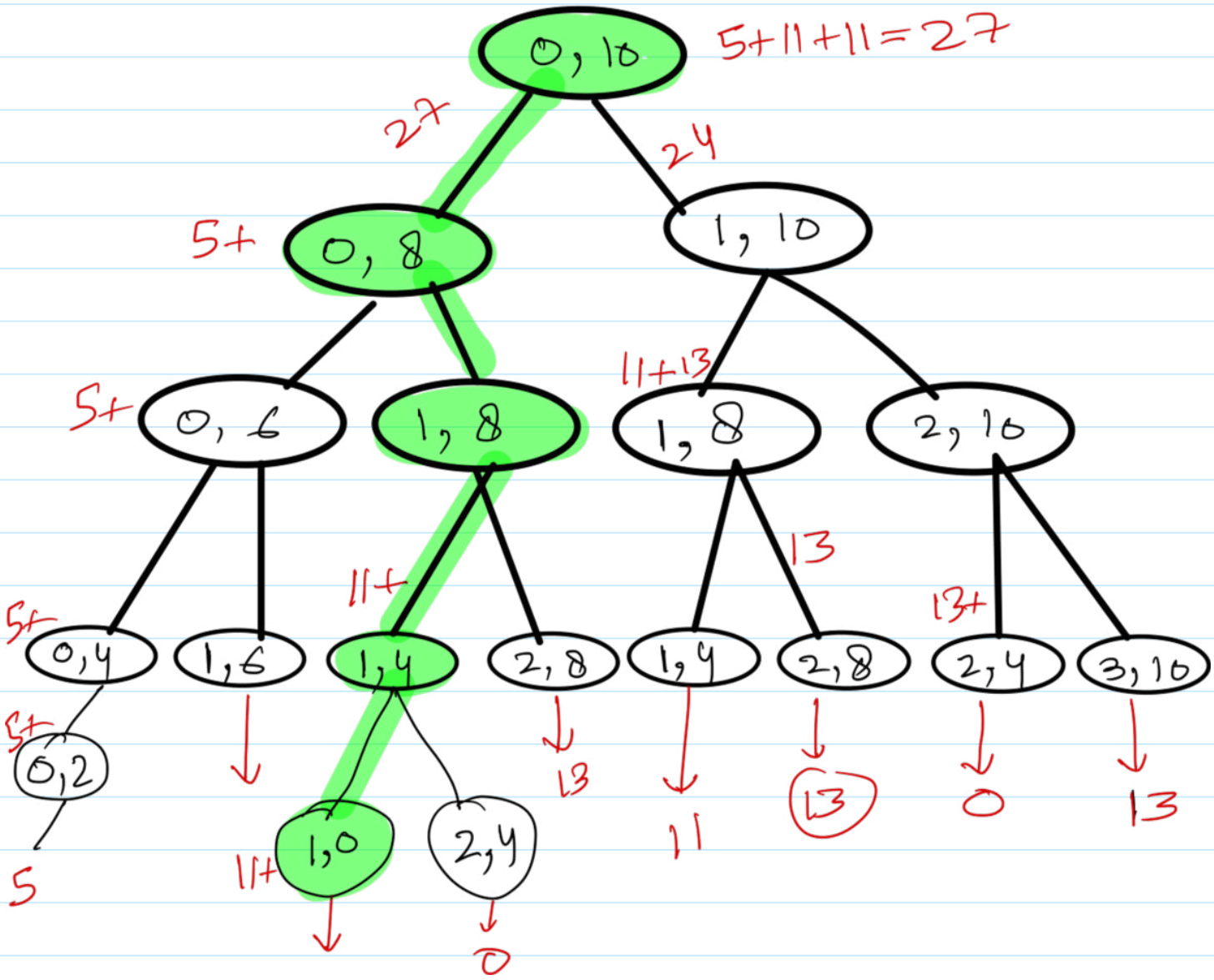
```
Let us say we have 'N' = 3 items and a knapsack of capacity 'W' =  10
'PROFIT' = { 5, 11, 13 }
'WEIGHT' = { 2, 4, 6 }

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:

```cpp
int findknap(int currind, int lastind, int capacity, vector<int> &profit, vector<int> &weight,
             vector<vector<int>> & dp)
{   if(currind==lastind)
    { return ((int)(capacity/weight[lastind])) * profit[lastind];
    }
     if(dp[currind][capacity]!=-1)
         return dp[currind][capacity];
     int notpick = findknap(currind+1,lastind,capacity,profit,weight,dp);
     int pick=-1e9;
     if(weight[currind]<=capacity)
         pick = profit[currind] + findknap(currind,lastind,capacity-weight[currind],profit,weight,dp);
 return dp[currind][capacity]= max(pick,notpick);
}

int unboundedKnapsack(int n, int w, vector<int> &profit, vector<int> &weight)
{  vector<vector<int>> dp(n,vector<int> (w+1,-1));
   return findknap(0,n-1,w,profit,weight,dp);
}
```

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++)
          { int notpick = next[capacity];
     int pick=0;
     if(weight[currind]<=capacity)
         pick = profit[currind] + curr[capacity-weight[currind]];
        curr[capacity]= max(pick,notpick);
          }
       next=curr;
     }
   return next[w];
}
```

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

Further Space Optimization: Using one 1D array

```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--)
     {  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];
}
```

LinkedIn/kapilyadav22

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

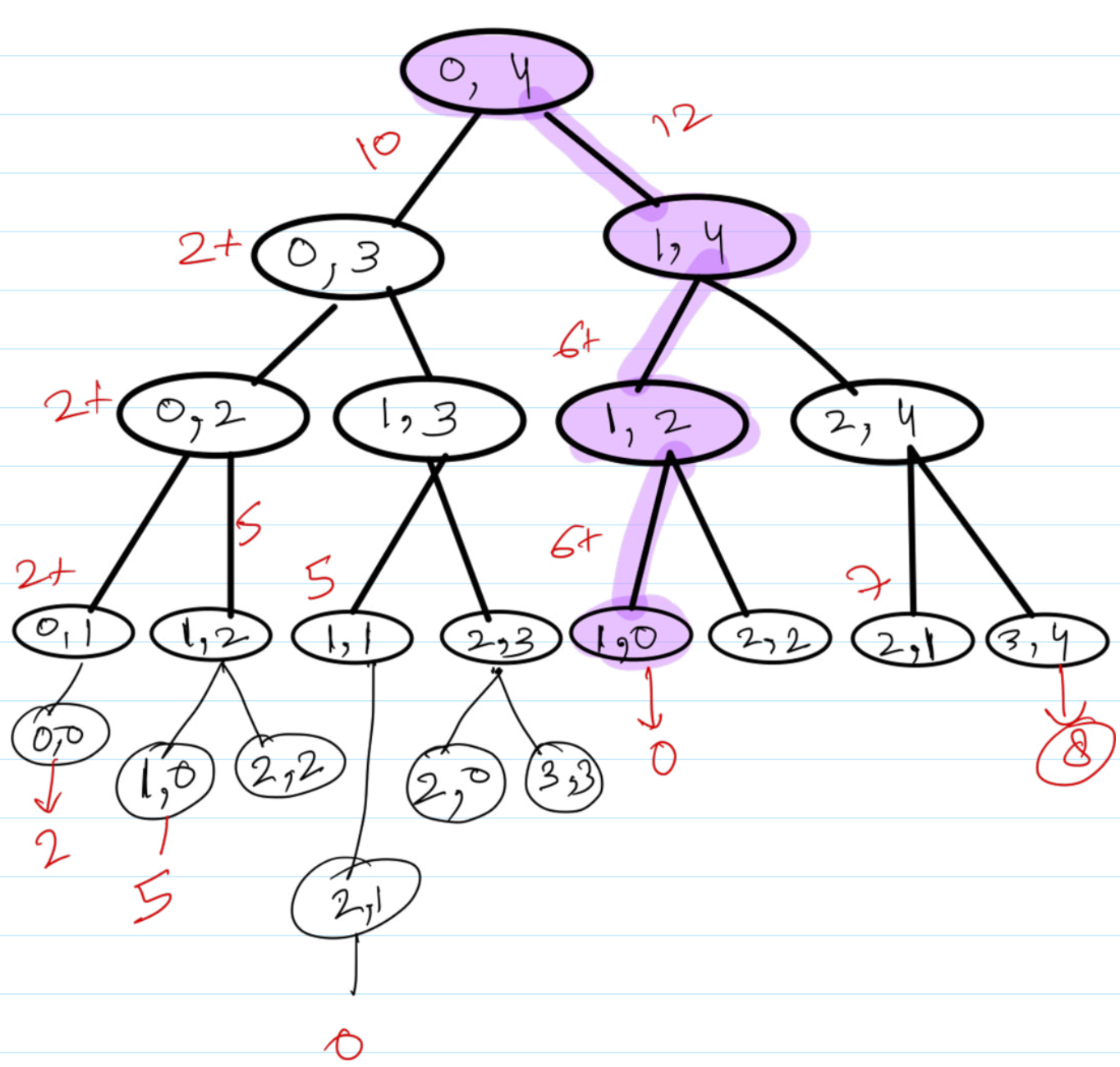
Suggest Edit

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

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:

```cpp
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>>& v )
    {
        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-currentlength,v);
        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:

```cpp
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 - rodlength];
 }    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];
}
```

LinkedIn/kapilyadav22

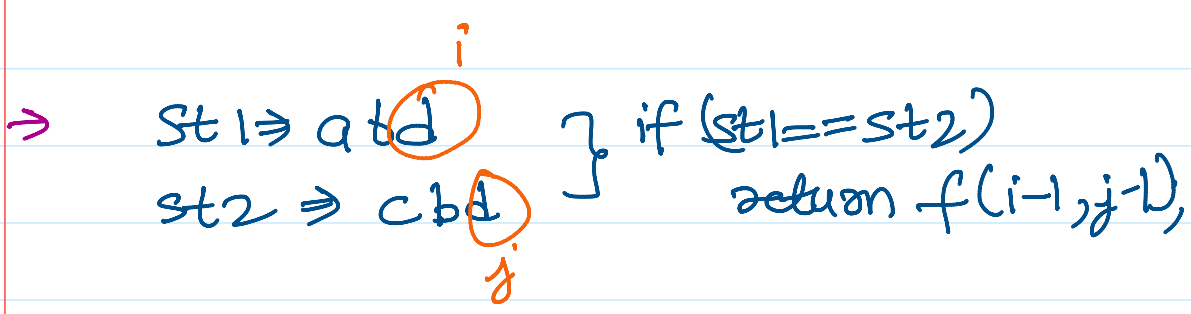
TC: O(N*N), where N is the length of the Rod
SC: O(N)

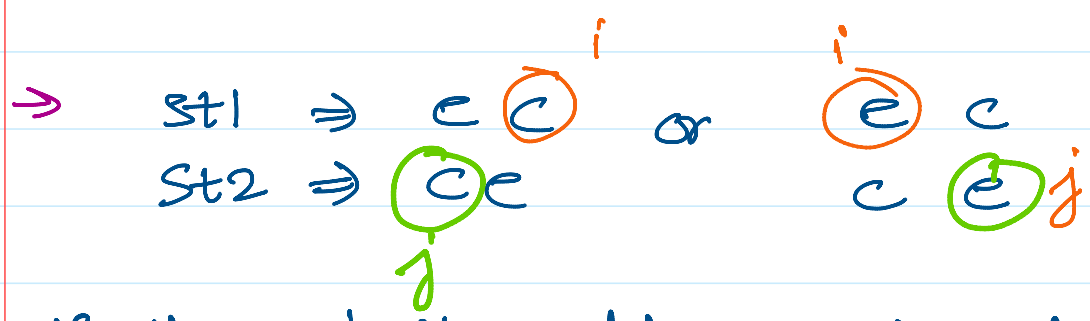
*LECTURE - 25 LONGEST COMMON SUBSEQUENCE (DP ON STRINGS)

22 June 2022 12:13

- A **subarray is a contiguous part of array and maintains relative ordering of elements**. For an array/string of size n, there are 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 array**. For a sequence of 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 set of size n, we can have (2^n) sub-sets in total.

Abc = a, b, c, ab, bc, ac, abc, ''





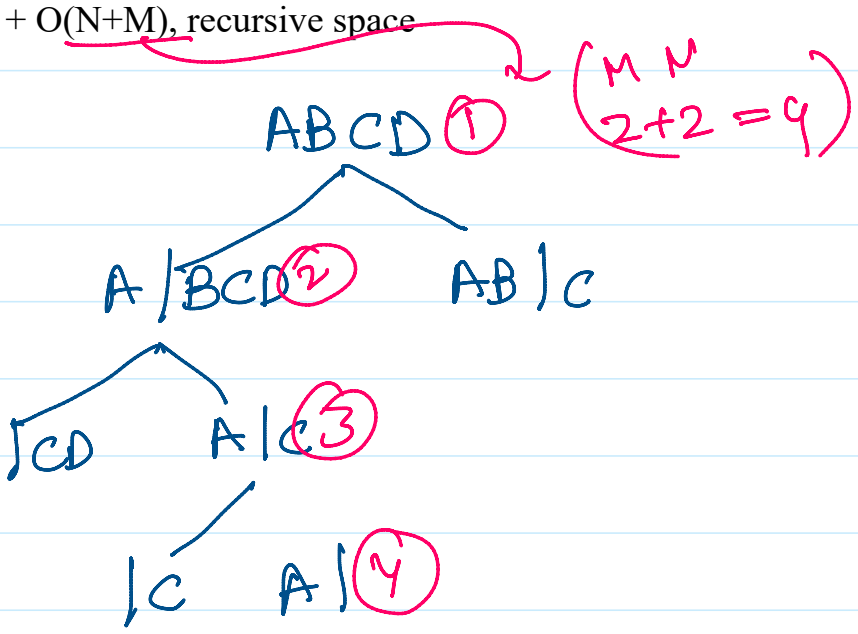
if they don't match, we will explore all other possibilities.

$$\left.\begin{matrix} f(i-1, j) \\ f(i, j-1) \end{matrix}\right\} \max$$

MEMOIZATION:

```cpp
int findlcs(int ind1, int ind2, string s1, string s2, vector<vector<int>> &dp)
    {   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:

```cpp
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][ind2+1]);
            }
        }
        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)

LinkedIn/kapilyadav22

Space Optimization:

```cpp
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];
    }
```

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

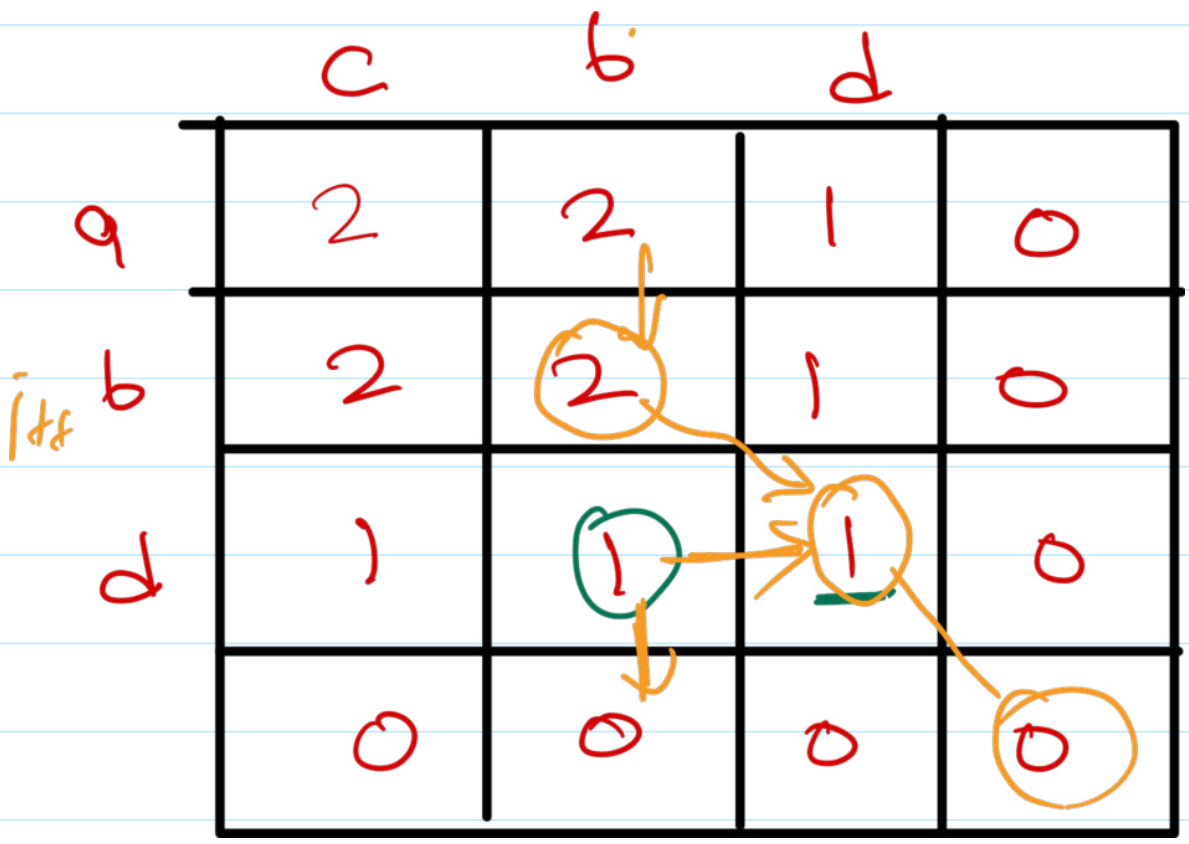
LECTURE - 26 PRINT LONGEST COMMON SUBSEQUENCE

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 Suggest Edit

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

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

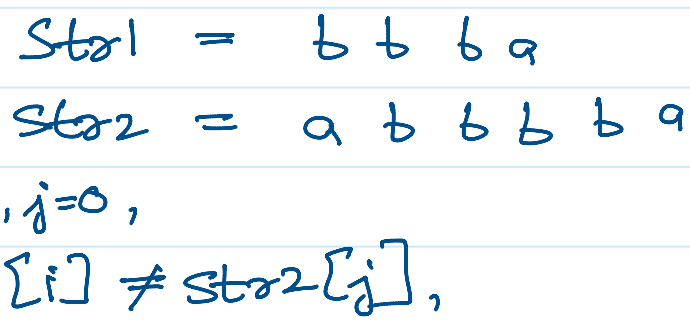
Same as Longest CommonSubsequence, But here If the strings are not matching, just pass the countlcs =0, and find the Longest common string from left and right.

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 (i,j+1) even if 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,s2,0));
9        return max(countlcs,notmatch);
10   }
11
12  int lcs(string &s1, string &s2){
13    int m = s1.length();
14    int n = s2.length();
15    int countlcs=0;
16     return findlcs(0,0,s1,s2,countlcs);
17  }
```

TC: Exponential 3^N
SC:O(1)

MEMOIZATION:

```
  int findlcs(int ind1, int ind2,int countlcs, string s1, string s2,vector<vector<vector<int>>>& dp)
   {   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]);
            }
        }
        return ans;
}
```

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

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 → " b b a a b a b b " → S1

Take reverse of string

" b b a b a a b b " → S2

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

⇒ b b a a a b b. or b b a b a b b

Tabulation:

```
1 int longestCommonSubsequence(string s1, string s2) {
2         int m = s1.length();
3         int n = s2.length();
4
5         vector<int> next(n+1,0);
6
7         for(int ind1 = m-1;ind1>=0;ind1--)
8         { vector<int> curr(n+1,0);
9             for(int ind2 = n-1;ind2>=0;ind2--)
10            { if(s1[ind1]==s2[ind2])
11             curr[ind2]=1 + next[ind2+1];
12            else
13            curr[ind2]=max(next[ind2],curr[ind2+1]);
14            }
15         next = curr;
16        }
17        return next[0];
18    }
19 int longestPalindromeSubsequence(string s1)
20 {string s2 =s1;
21        reverse(s2.begin(),s2.end());
22        return longestCommonSubsequence(s1,s2);
23 }
```

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

OOPs in C++

The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except this function.

Class : It is a user defined data types, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

Object : When a class is defined no memory is allocated but when it is instantiated (i.e., object is created) memory is allocated.

Encapsulation : In OOP, Encapsulation is defined as binding together the data and the functions that manipulates them

Abstraction : Abstraction means displaying only essential information and hiding the details.

- Abstraction using classes
- Abstraction using Header files (math.h → pow())

Polymorphism : In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

- Operator overloading
- Function overloading
  - ↳ int sum(10, 20, 30)
    int sum(10, 20)

Inheritance : The capability of a class to derive properties and characteristics from another class is called Inheritance.

- Subclass
- Superclass
- Reusability

Dynamic Binding : In dynamic binding, the code to be executed in response to function call is decided at run time.

Constructors : A constructor is a member function of a class which initializes objects of a class. In C++ constructor is automatically called when the object creates.

It has same name as class itself.
Constructor don't have a return type.

1. Default Constructor (No parameter passed)
2. Parametrized Constructor
3. Copy Constructor

Destructor in C++ : Derived class destructor will be invoked first, then the base class destructor will be invoked.

Access Modifier : Public - can be accessed by any class.
Private :- can be accessed only by a function in a class (inaccesible outside the class).
Protected :- It is also inaccessible outside the class but can be accessed by subclass of that class.

Note: If we do not specify any access modifier inside the class then by default the access modifier for the member will be private.

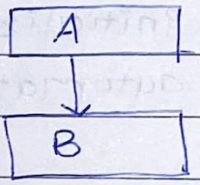
Friend class: A friend class can access private and protected members of other class in which it is declared as friend.

Ex-: friend class B;

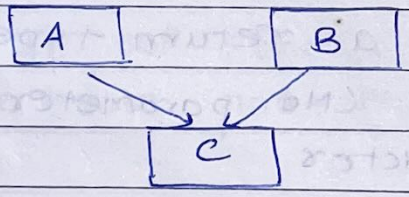
- Inheritance

```
class subclass : accessmode baseclass
{
}
```

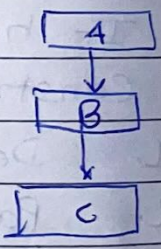
1. Single inheritance



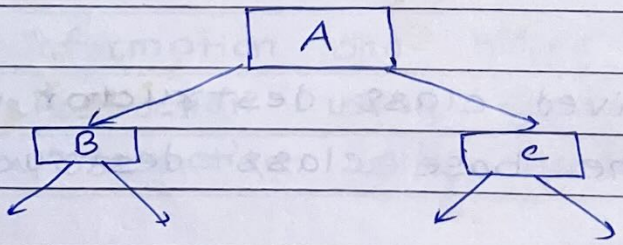
2. Multiple inheritance



3. Multilevel



4. Heerarchical inheritance



5. Hybrid

Combination of one or more type.

- Polymorphism

-> Compile time Poly → Operator overloading
                      → Function overloading.

-> Run time Poly

↳ function overriding occurs when a derived class has a definition of one or more members of base class.

Advantages of Data Abstraction

- Avoid code duplication and inc. reusability.
- can change internal implementation of class independently.

Structure Vs class : Most important difference is security.
A structure is not secure and cannot hide its member function and variable while class is secure and can hide its programming & designing details.

Local Classes in C++ : A class declared inside a function becomes local to that function and is called local class.
All the methods of local class must be defined inside the class only.

Virtual function and Runtime Polymorphism :
A virtual function is a member function which is declared within a base class and redefined (overriden) by derived class.
Functions are declared with Virtual Keyword in base class.

Exception Handling in C++ :
try : represent a block of code that can throw an exception.
catch : represent a block of code that get executed when error is thrown
throw : Used to throw an exception.

There is a special catch block → catch(...)
It catches all types of error.

- Inline Function

→ inline is a request not command.
It is function that is expanded in line when it is called. When the inline function is called, whole code get inserted or substituted at the point of inline function call.

```
inline return-type fun( )
{
}
```

- Function Overloading is a feature in C++ where two or more functions can have same name but different parameters.

```
void print(int i)
{
    cout << "Here is int " << i << endl;
}

void print(float i)
{
    cout << " Here is float" << i << endl;
}

int main
{
    print(10);
    print(10.12);
}
```

Differences b/w C and C++

	C	C++
1.	C supports procedural prog.	• C++ is known as hybrid language, because it support both procedural and object oriented programming.
2.	As C does not support the OOPs concept so it has no support for polymorphism, encapsulation and inheritance.	• C++ has support for polymorphism, encapsulation and inheritance as it is an OOPs language.
3.	C is a subset of C++	• C++ is superset of C
4.	C contains 32 keywords	• C++ contain 52 keywords (public, private, protected, try, catch, throw ...)
5.	C is a function driven language	• C++ is an object driven language.
6.	Function and operator overloading is not support in C.	• C++ supports function & operator overloading.
7.	C does not support exception handling	• C++ supports exception handing using try and catch

- Structure is a collection of dissimilar elements

- Static Members in C++

  - Static variable in a function : When a variable is declared as static, space for it gets allocated for the lifetime of the program. (defaut intialized to 0)
  Even if the function is called multiple times, the space for it is allocated once.

  - Static variable in a class :
    → Declared inside the class body.
    → Also known as class member variable.
    → They must be defined outside the class.
    → Static variable doesn't belong to any object, but to the whole class.
    → There will be only one copy of static member variable for the whole class.

Ex:

```
class Account
{
private:
    int balance;
    static float roi;
public:
    void setBalance(int b)
    { balance = b; }
};

// intialised outside class
float Account :: roi = 3.5f;
void main
{
    Account a1;
}
```

- Object can also be declared as static.

```
static Account a1;
```

- Static function in a Class

Static member functions are allowed to access only the static data members or other static member functions.

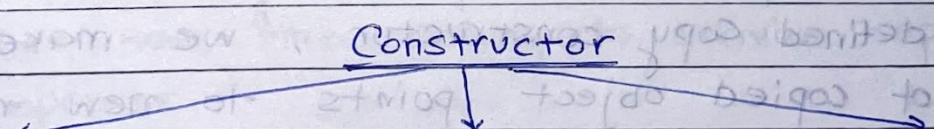
- Constructors :

→ Constructors is an special member function of the class. It is automatically invoked when an object is created.

→ It has no return type.

→ Constructor has same name as class itself.

→ If we do not specify, then C++ compiler generates a default constructor for us.



Class_name();	Class_name(parameters)	Class_name(const Class_name &obj);
update()	update(int x, int y)	update(const update &p2)
{	{	{ a=p2.a;
a=10;	a=x;	b=p2.b;
b=20;	b=y;	}
}	}	

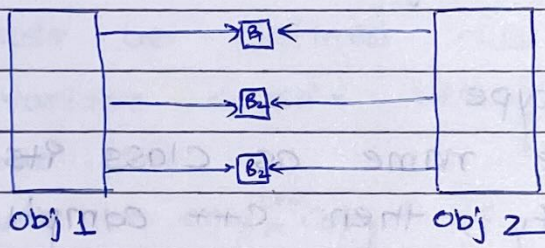
Compiler generates two constructor by itself.

1. Default Constructor
2. Copy Constructor

But if any of the constructor is created by user, then default constructor will not be created by compiler.

Construction overloading can be done just like function overloading.

Default (Compiler's) Copy constructor can done only shallow copy.



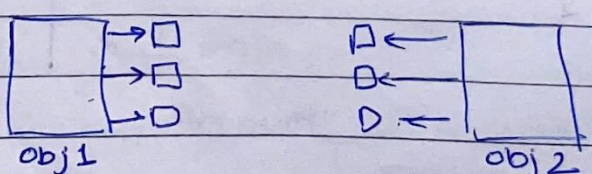
(Points to same memory location)

Deep Copy is possible only with user defined constructors. In user defined copy constructor, we make sure that pointers of copied object points to new memory location.

Can we make Copy Constructor private? Yes

Why argument to Copy constructor must be passed as a reference?

Because if we pass value, then it would made to call copy constructor which becomes non-terminating.



Deep Copy

- Destructor

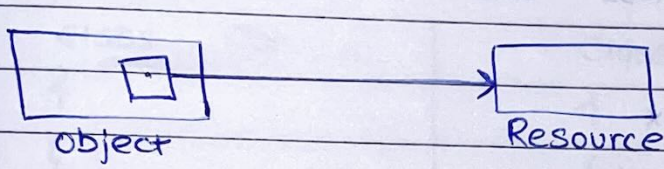
→ Destructor is a member function which destructs or deletes an object.

→ Destructor don't take any argument and don't have any return type.

→ Only one destructor is possible

→ Destructor cannot be static.

→ Actually destructor doesn't destroy object, it is the last function that invoked before object destroy.



Destructor is used, so that before deletion of obj we can free space allocated for this resource. B/c if obj gets deleted then space allocated for obj will be free but resource doesn't.

- Operator Overloading

C++ have the ability to provide special meaning to the operator.

```
class Complex
{
    .
    .
    .
    Complex operator + (Complex &c1)
    {
        Complex res;
        res.a = c1.a;
        res.b = c2.b;
    }
};
int main()
{
    c = c1 + c2
}
```

As '+' can't add complex no's directly. So we can define a function with name + but we need write operator keyword before it. So, we can use all operator like this.

Friend Class

A friend class can access the private and protected members of other class in which it is declared as friend.

There can be friend class and friend function.

Ex:

```
class Box
{   private;
        double width;
    public;
        friend void printWidth(Box box);
        void setWidth(double Wid);
};

void Box::setWidth(Box double Wid)
{       width = Wid;    }

void printWidth(Box box)
{   cout << box.width;  }

int main()
{   Box box;
    box.setWidth(4);
    printWidth(box);
}
```

Inheritance

It is a process of inherting properties and behaviour of existing class into a new class.

```
class Base_class
{
};
```

```
class der_class : Visibility_Mode Base class
{
};
```

Ex:

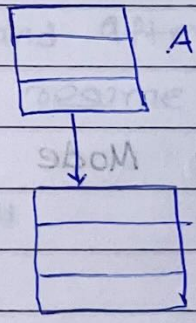
```
class Car
{
};
```

```
class Sports_Car : public Car
{
};
```

Types of Inheritance :

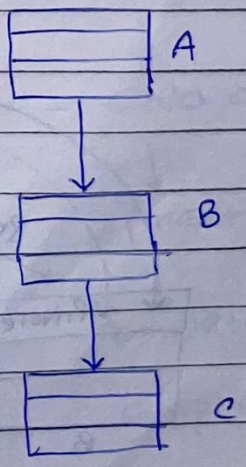
a). Single Inheritance :

```
class B : public A
{

};
```

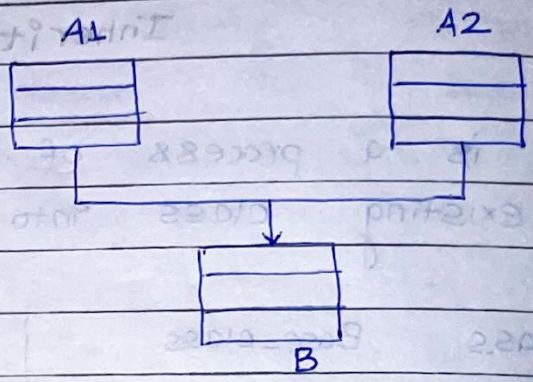


b). Multilevel Inheritance :

```
Class B : public A
{
};
class C : public B
{
};
```

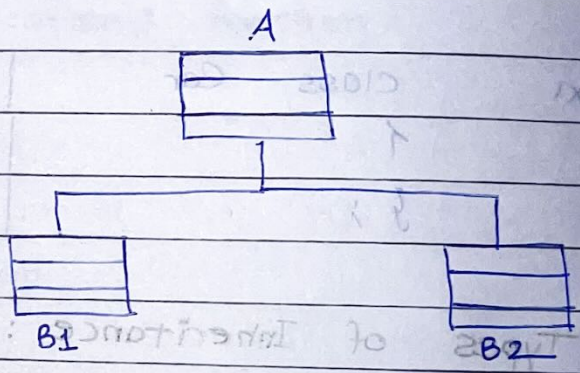


c). Multiple Inheritance



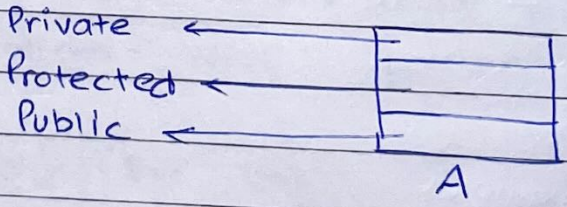

```
class A1          class A2
{                 {
};                };

class B : public A1, public A2
{
};
```

d). Heirarchial Inheritance



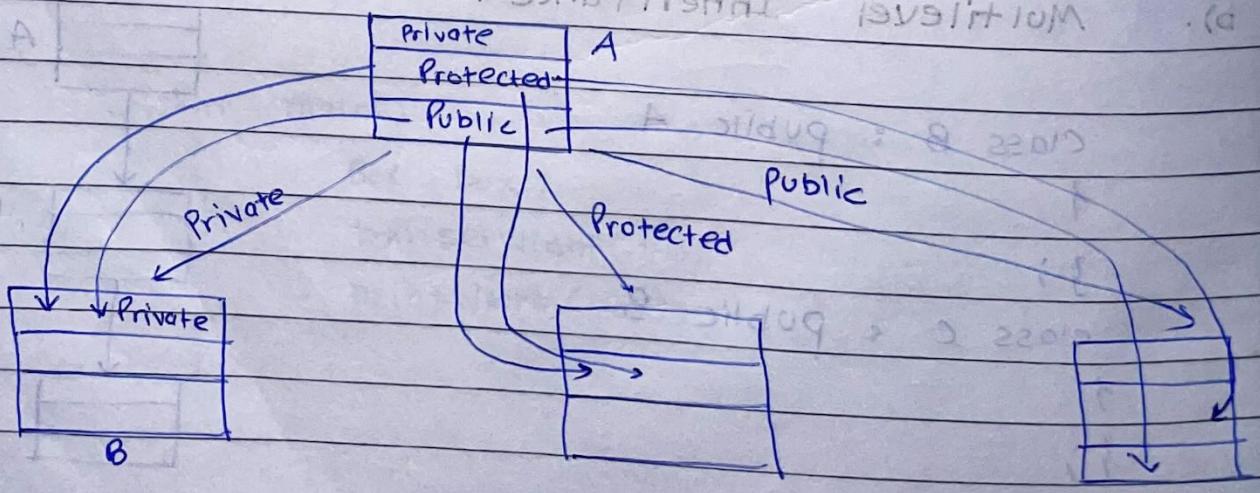

```
class B1 : public A
{
};
class B2 : public A
{
};
```

→ Visibility Mode :



A - base class

B - Sub class



If B is subclass and visibility Mode is public.

```
class A : public B
{
};
```

then public member of A will be public in B, and protected will protected.

If visibility mode is private then both protected and public member of A will be private member of B

- Is a Relationship is always implemented as a public inheritance.

- Constructor and Destructor in Inheritance

First child class constructor will run during creation of object of child class, but as soon as obj is created child class constructor run and it will call constructor of it's parent class and after the execution of parent class constructor it will resume it constructor execution.

Child → B() : A() → parent constructor call

```
{
{
```

constructor exec.

obj

↓

Child const

↓

parent const

↓

complete parent

↓

complete child

While in case of destructor, first child destructor exec, then parent desc. executed.

this pointer

Every object in C++ has access to its own address through an important pointer called this pointer.

Friend function doesn't have a 'this' pointer, b/c friends are not members of a class. Only member function have this pointer.

```
Class Box
{
    private:
        int l, b, h;
    public:
        void set (int l, int b, int h)
        {
            this->l = l;
            this->b = b;
            this->h = h;
        }
};
int main ()
{
    Box b;
    b.set (5, 10, 4);
}
```

Method Over Riding
(achieved at run time)

It is the redefinition of base class function in its derived class, with same return type and same parameters.

While method Overloading is achieved at Compile time.

Ex:

```
Class Car
{
    private:
        int gearno;
    public:
        void change_gear(int gear)
        {
            gear++;
        }
};

Class SportsCar : public Car
{
    void change_gear(int gear)
    {
        if (gear > 5)
            gear++;
    }
};

int main
{
    SportsCar sc;
    sc.change_gear(4);
}
```

function of sportsCar Class will be called.

While calling change_gear(), first it check if any fun with this name exist in calling class, otherwise it goes to baseClass.

Useful: like we have change_gear for all except one car which have unique method of gearChange.

Virtual Function

A virtual function is a member function which is declared with a 'virtual keyword' in the base class and redeclared (overridden) in a derived class. When you refer to a object of derived class using pointer to a base class, you can call a virtual function of that object and execute the derived class's version of the function.

- They are used to achieve Run time Polymorphism.
- Virtual Function cannot be static and also cannot be friend function of another class.

Compile-time (Early binding) Vs Run-time (Late Binding)

```
class base
{       public:
            virtual void print()
            {       cout<< " This is base print" <<endl;
            }
            void show()
            {       Cout << " Base show fun" << endl;
            }
};
class derived
{       public:
            void print()
            { cout << " derived Print" <<endl; }
            void show()
            { cout << " derived show fun" << endl; }
};
```

```
int main ()
{       base *bptr ;
        derived der;
        bptr = &der;

        bptr -> print() ;        // Run time
        bptr -> show() ;         // Compile time
}
```

Output: derived print // Late Binding
This Base show fun // Early binding

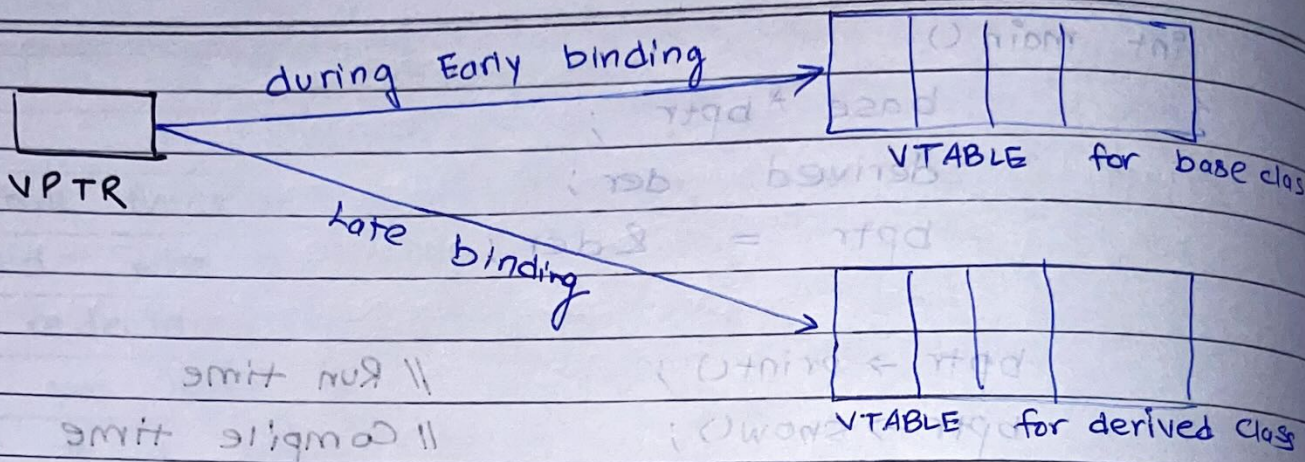
As during compiler time bptr behaviour judged on the basis of which class it belong, so bptr represent base class.

If the function is not virtual then it will allow binding at compiler time and print fun of base class will get binded b/c bptr represent base class.
But at run time bptr points to the object of class derived, so it will bind function of derived at run time.

Working of Virtual Function (VTable & VPtr)

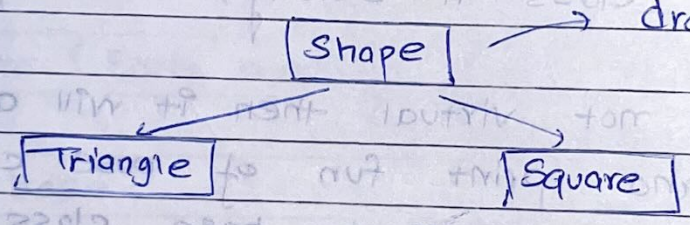
If a class contains virtual function then compiler itself does two things:

1. A virtual pointer (VPTR) is created every time obj is created for that class which contains virtual function.
2. Irrespective of object is created or not, static array of pointer called VTABLE where each cell point to each virtual function is created, in base class and derived class.



Pure Virtual Function and abstract Class

Sometimes implementation of all function cannot be provi in the base class. Such a class is called abstract class.



A pure virtual function in C++ is a virtual function for which we don't have any implementation, we only declare it.

```
// Abstract Class
class Test
{
    public:
        virtual void fun() = 0;
};
```

Pure Virtual function

1. A class is abstract if it has at least one pure virtual function.

We cannot declare object of abstract class.
Ex: Test t; will show error

2. We can have pointer or reference of abstrac class.
3. We can access the other functions except virtual by object of its derived class.
4. If we dont override the pure virtual function in derived class then it becomes abstract.
5. An abstract class can have constructors.

(Read from GfG)

Template in C++

```
template <class X> int check (int a, X b)
                   X         X
{
    if (a > b)
        return a;
    else return b;
}
```

It doe just help in data type. So that we can write generic function that can be used for different data type.

Dynamic Constructor

When allocation of memory is done dynamically using dynamic memory allocator 'new' in constructor.

```
class geeks
{   public:
        void fun() { p = new char[6]; }
};
int main()
{   geeks g = new geeks();
}
```

1. Introduction

What is Standard Template Library in C++ :-

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators.

It mainly has 3 components -

1. Containers
2. Algorithms
3. Iterators

So, let's see them one by one…

Note – The 'bits/stdc++.h>' is the universal header file for every C++ STL element. Include it in your every C++ program to avoid writing the header files for specific or separate STL elements.

2. Containers

A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows great flexibility in the types supported as elements.

The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).

In C++ STL, we have 4 types of Containers –

1. Sequence Containers
   - 1.1 Vector
   - 1.2 Deque
   - 1.3 List
   - 1.4 Arrays
   - 1.5 Forward Lists
2. Container Adaptors
   - 2.1 Stack
   - 2.2 Queue
   - 2.3 Priority Queue
3. Associative Containers
   - 3.1 Set
   - 3.2 Map
   - 3.3 Multiset
   - 3.4 Multimap
4. Unordered Associative Containers
   - 4.1 Unordered Set
   - 4.2 Unordered Map
   - 4.3 Unordered Multiset
   - 4.4 Unordered Multimap

Let’s see the most widely and most commonly used STL containers

2.1 Sequence Containers

2.1.1 Vectors

Vectors are dynamic arrays with the ability to resize themselves automatically when an element is inserted or deleted, with their storage being handled automatically by the container.

Operations on vectors -

1. push_back() - It push the elements into a vector from the back.
2. pop_back() - It is used to pop or remove elements from a vector from the back.
3. size() - Returns the number of elements in the vector.
4. empty() - Returns whether the vector is empty or not.
5. erase() - It is used to remove elements from a container from the specified position or range.
6. clear() - It is used to remove all the elements of the vector container.
7. front() - Returns a reference to the first element in the vector.
8. back() - Returns a reference to the last element in the vector.
9. sort(v.begin(), v.end()) - Sorts the elements in the vector.
10. reverse(v.begin(), v.end()) - Reverses the elements in the vector.

Syntax -

```
vector<data_type> vector_name;
```

Operation	Time Complexity	Space Complexity
push_back()	O(1)	O(1)
pop_back()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
erase()	O(n)	O(1)
clear()	O(n)	O(1)
front()	O(1)	O(1)
back()	O(1)	O(1)
sort()	O(nlog(n))	O(1)
reverse()	O(n)	O(1)

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    vector<int> v;              // Creating a vector
    vector<int> v2(5);          // Creating a vector with 5 elements
    vector<int> v3(5, 0);       // Creating a vector with 5 elements having value 0

    v.push_back(5);             // Adding 5
    v.push_back(11);            // Adding 11
    v.push_back(2);             // Adding 2
    v.push_back(7);             // Adding 7

    cout<<v.size()<<endl;       // Outputs 4

    v.pop_back();               // Remove last element i.e. 7

    cout<<v.empty()<<endl;      // Outputs 0 (v is not empty)

    sort(v.begin(), v.end());   // Sorting the vector

    for(int i=0;i<v.size();i++){
        cout<<v[i]<<" ";            // Outputs 2 5 11
    }
    return 0;
}

```

2.1.2 Deque

Double-ended queues are sequence containers with the feature of expansion and contraction on both ends. Double-ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.

Operations on deque -

1. push_back() - It pushes the elements into deque from the back.
2. push_front() - It pushes the elements into deque from the front.
3. pop_back() - It is used to pop or remove elements from deque from the back.
4. pop_front() - It is used to pop or remove elements from deque from the front.
5. size() - Returns the number of elements in the deque.
6. empty() - Returns whether the deque is empty or not.
7. erase() - It is used to remove elements from a container from the specified position or range.
8. clear() - It is used to remove all the elements of the deque container.
9. front() - Returns a reference to the first element in the deque.
10. back() - Returns a reference to the last element in the deque.

Syntax -

```
deque<data_type> deque_name;
```

Operation	Time Complexity	Space Complexity
push_back()	O(1)	O(1)
pop_back()	O(1)	O(1)
push_front()	O(1)	O(1)
pop_front()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
erase()	O(n)	O(1)
clear()	O(n)	O(1)
front()	O(1)	O(1)
back()	O(1)	O(1)

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    deque<int> d;              // Creating a deque
    deque<int> d2(5);          // Creating a deque with 5 elements
    deque<int> d3(5, 0);       // Creating a deque with 5 elements having value 0

    d.push_back(5);             // Adding 5 from back
    d.push_back(11);            // Adding 11 from back
    d.push_back(2);             // Adding 2 from back
    d.push_front(7);             // Adding 7 from front
    d.push_front(14);              // Adding 14 from front

    cout<<d.size()<<endl;       // Outputs 5

    d.pop_back();               // Remove element from back i.e. 2
    d.pop_front();               // Remove element from front i.e. 14

    cout<<d.empty()<<endl;      // Outputs 0 (d is not empty)

    reverse(d.begin(), d.end());   // Reversing the deque

    for(int i=0;i<d.size();i++){
        cout<<d[i]<<" ";            // Outputs 11 5 7
    }
    return 0;
}

```

2.1.3 Lists

Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, the list has slow traversal, but once a position has been found, insertion and deletion are quick.

This list is implemented using a doubly-linked list. For implementing a singly linked list, we use a forward list.

Operations on list -

1. push_back() - Adds a new element at the end of the list.
2. push_front() – Adds a new element at the beginning of the list.
3. pop_back() – Removes the last element of the list.
4. pop_front() - Removes the first element of the list.
5. size() - Returns the number of elements in the list.
6. empty() - Returns whether the list is empty or not.
7. erase() - It is used to remove elements from a container from the specified position or range.
8. clear() - It is used to remove all the elements of the list container.
9. front() - Returns the value of the first element in the list.
10. back() - Returns the value of the last element in the list.

Syntax -

```
list<data_type> list_name;
```

Operation	Time Complexity	Space Complexity
push_back()	O(1)	O(1)
pop_back()	O(1)	O(1)
push_front()	O(1)	O(1)
pop_front()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
erase()	O(n)	O(1)
clear()	O(n)	O(1)
front()	O(1)	O(1)
back()	O(1)	O(1)

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    list<int> l;              // Creating a list
    list<int> l2(5);          // Creating a list with 5 elements
    list<int> l3(5, 0);       // Creating a list with 5 elements having value 0

    l.push_back(5);             // Adding 5 from back
    l.push_back(11);            // Adding 11 from back
    l.push_back(2);             // Adding 2 from back
    l.push_front(7);             // Adding 7 from front
    l.push_front(14);             // Adding 14 from front

    cout<<l.size()<<endl;       // Outputs 5

    l.pop_back();               // Remove element from back i.e. 2
    l.pop_front();               // Remove element from front i.e. 14

    cout<<l.empty()<<endl;      // Outputs 0 (l is not empty)

    reverse(l.begin(), l.end());   // Reversing the list

    for(auto i: l){
        cout<<i<<" ";            // Outputs 11 5 7
    }
    return 0;
}
```

2.2 Container Adaptors

2.2.1 Stack

Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end (top) and an element is removed from that end only.

Operations on stack -

1. push() - It adds an element to the top of the stack.
2. pop() - It deletes the top most element of the stack.
3. size() - Returns the number of elements in the stack.
4. empty() - Returns whether the stack is empty or not.
5. top() - Returns a reference to the top most element of the stack.

Syntax -

```
stack<data_type> stack_name;
```

Operation	Time Complexity	Space Complexity
push()	O(1)	O(1)
pop()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
top()	O(1)	O(1)

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    stack<int> s;              // Creating a stack

    s.push(5);             // Adding 5
    s.push(11);            // Adding 11
    s.push(2);             // Adding 2
    s.push(7);             // Adding 7

    cout<<s.size()<<endl;       // Outputs 4
    cout<<s.top()<<endl;       // Outputs 7

    s.pop();               // Remove topmost element i.e. 7

    cout<<s.empty()<<endl;      // Outputs 0 (s is not empty)

    while(!s.empty()){
        cout<<s.top()<<" ";            // Outputs 2 11 5
        s.pop();               // Pops/Removes every top most element
    }
    return 0;
}

```

2.2.2 Queue

Queues are a type of container adaptors that operate in a first in first out (FIFO) type of arrangement. Elements are inserted at the back (end) and are deleted from the front.

Operations on stack -

1. push() - It adds an element to the end of the queue.
2. pop() - It deletes the first element of the queue.
3. size() - Returns the number of elements in the queue.
4. empty() - Returns whether the queue is empty or not.
5. top() - Returns a reference to the top most element of the stack.
6. front() - Returns a reference to the first element in the queue.
7. back() - Returns a reference to the last element in the queue.

Syntax -

```
queue<data_type> queue_name;
```

Operation	Time Complexity	Space Complexity
push()	O(1)	O(1)
pop()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
front()	O(1)	O(1)
back()	O(1)	O(1)

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    queue<int> q;              // Creating a queue

    q.push(5);             // Adding 5
    q.push(11);            // Adding 11
    q.push(2);             // Adding 2
    q.push(7);             // Adding 7

    cout<<q.size()<<endl;       // Outputs 4
    cout<<q.front()<<endl;       // Outputs 5
    cout<<q.back()<<endl;       // Outputs 7

    q.pop();               // Remove the front element i.e. 5

    cout<<q.empty()<<endl;      // Outputs 0 (q is not empty)

    while(!q.empty()){
        cout<<q.front()<<" ";            // Outputs 11 2 7
        q.pop();               // Pops/Removes every front element
    }
    return 0;
}

```

2.2.3 Priority Queue

Priority queues are a type of container adapters, specifically designed such that the first element of the queue is either the greatest or the smallest of all elements in the queue. Priority queues are built on the top to the max heap by default. Hence, the elements are stored in decreasing order by default.

Operations on priority queue -

1. push() - It adds an element to the end of the queue.
2. pop() - It deletes the first element of the queue.
3. size() - Returns the number of elements in the queue.
4. empty() - Returns whether the queue is empty or not.
5. top() - Returns a reference to the top most element of the queue.

Syntax -

For max-heap implementation(default) i.e. elements are stored in decreasing order -

```
priority_queue<data_type> queue_name;
```

For min-heap implementation i.e. elements are stored in increasing order -

```
priority_queue<data_type, vector<data_type>, greater<data_type>> queue_name;
```

Operation	Time Complexity	Space Complexity
push()	O(log(n))	O(1)
pop()	O(log(n))	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
top()	O(1)	O(1)

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    priority_queue<int> pq;             // Creating a priority_queue for max-heap (default)
    priority_queue<int, vector<int>, greater<int>> pq2;    // Creating a priority_queue for min-heap

    pq.push(5);             // Adding 5
    pq2.push(5);
    pq.push(11);            // Adding 11
    pq2.push(11);
    pq.push(2);             // Adding 2
    pq2.push(2);
    pq.push(7);             // Adding 7
    pq2.push(7);

    cout<<pq.size()<<endl;       // Outputs 4
    cout<<pq.top()<<endl;       // Outputs 11

    pq.pop();               // Remove topmost element i.e. 11

    cout<<pq.empty()<<endl;      // Outputs 0 (pq is not empty)

    while(!pq.empty()){
        cout<<pq.top()<<" ";            // Outputs 7 5 2
        pq.pop();                // Pops/Removes every top/first element of pq
    }
    cout<<endl;
    while(!pq2.empty()){
        cout<<pq2.top()<<" ";            // Outputs 2 5 7 11
        pq2.pop();               // Pops/Removes every top/first element of pq2
    }
    return 0;
}
```

2.3 Associative Containers

2.3.1 Set

Sets are a type of associative containers in which each element has to be unique because the value of the element identifies it. Here, the values are stored in a specific or sorted order. It doesn't allow duplicate values i.e. values are unique. The values are immutable i.e. The value of the element cannot be modified once it is added to the set, though it is possible to remove and then add the modified value of that element.

It follows binary search tree implementation. Also, the values stored in it are unindexed.

Operations on set -

1. insert() - It adds a new element to the set.
2. erase() - It removes the specified value from the set.
3. clear() - It removes all the elements from the set.
4. size() - Returns the number of elements in the set.
5. empty() - Returns whether the set is empty or not.
6. find() - Returns an iterator to the specified element in the set if found, else returns the iterator to end.

Syntax -

For storing values in ascending order -

```
set<data_type> set_name;
```

For storing values in descending order -

```
set<data_type, greater<data_type>> set_name;
```

Operation	Time Complexity	Space Complexity
insert()	O(log(n))	O(1)
erase()	O(log(n))	O(1)
clear()	O(log(N))	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
find()	O(log(n))	O(1)

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    set<int> s;                     // Creating a set in which values are stored in ascending order
    set<int, greater<int>> s2;      // Creating a set in which values are stored in descending order

    s.insert(5);           // Adding 5
    s.insert(11);          // Adding 11
    s.insert(2);           // Adding 2
    s.insert(7);           // Adding 7
    s2.insert(5);
    s2.insert(11);
    s2.insert(2);
    s2.insert(7);

    s.erase(5);             // Removes 5 from s
    cout<<s.size()<<endl;       // Outputs 3

    cout<<s.empty()<<endl;      // Outputs 0

    for(auto i: s){
        cout<<i<<" ";            // Outputs 2 7 11
    }
    cout<<endl;
    for(auto i: s2){
        cout<<i<<" ";            // Outputs 11 7 5 2
    }
    return 0;
}
```

2.3.2 Map

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have the same key values.

Operations on map -

6. insert(pair<dt, dt> (key, value)) – It inserts elements with a particular key in the map container.
7. erase() - It removes the specified key-value from the map.
8. clear() - It removes all the elements from the map.
9. size() - Returns the number of elements in the map.
10. empty() - Returns whether the map is empty or not.
11. find() - Returns an iterator to the specified element in the map if found, else returns the iterator to end.
12. count() - Returns the number of matches to element with the specified key-value in the map.

Syntax -

```
map<data_type, data_type> map_name;
```

Operation	Time Complexity	Space Complexity
insert()	O(log(n))	O(1)
erase()	O(log(n))	O(1)
clear()	O(log(N))	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
find()	O(log(n))	O(1)
count()	O(log(n))	O(1)

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    map<string, int> m;                     // Creating a map

    m.insert(pair<string, int> ("English", 45));     // Adding "English" as a key and "45" as a value in map
    m.insert(pair<string, int> ("Maths", 48));       // Adding "Maths" as a key and "48" as a value in map
    m["Science"] = 50;                              // Another way of adding elements into the map
    m["History"] = 41;

    m.erase("Science");         // Removes "Science" key and it's value from map
    cout<<m.size()<<endl;       // Outputs 3

    cout<<m.empty()<<endl;      // Outputs 0

    for(auto i: m){
        cout<<i.first<<" = "<<i.second<<endl;
    }

    //      Outputs
    //      English = 45
    //      History = 41
    //      Maths = 48
    return 0;
}

```

2.4 Unordered Associative Containers

2.4.1 Unordered Set

An unordered set is implemented using a hash table where keys are hashed into indices of a hash table so that the insertion is always randomized. All operations on the unordered_set takes constant time O(1).

The unordered_set allows only unique keys, for duplicate keys unordered_multiset should be used.

Operations on unordered_set -

1. insert() - It adds a new element to the unordered_set.
2. erase() - It removes the specified value from the unordered_set.
3. clear() - It removes all the elements from the unordered_set.
4. size() - Returns the number of elements in the unordered_set.
5. empty() - Returns whether the unordered_set is empty or not.
6. find() - Returns an iterator to the specified element in the unordered_set if found, else returns the iterator to end.

Syntax -

```
unordered_set<data_type> set_name;
```

Operation	Time Complexity	Space Complexity
insert()	O(1) or O(n) in worst case	O(1)
erase()	O(1)	O(1)
clear()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
find()	O(1) or O(n) in worst case	O(1)

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    unordered_set<int> us;           // Creating an unordered_set

    us.insert(5);           // Adding 5
    us.insert(11);          // Adding 11
    us.insert(2);           // Adding 2
    us.insert(7);           // Adding 7

    us.erase(5);             // Removes 5 from s
    cout<<us.size()<<endl;       // Outputs 3

    cout<<us.empty()<<endl;      // Outputs 0

    for(auto i: us){
        cout<<i<<" ";            // Outputs 7 2 11
    }
    return 0;
}

```

2.4.2 Unordered Map

The unordered_map is an associated container that stores elements formed by the combination of key-value and a mapped value. The key value is used to uniquely identify the element and the mapped value is the content associated with the key.

Internally unordered_map is implemented using Hash Table. All the operations on the unordered_map takes constant time O(1).

Operations on unordered_map -

1. insert(pair<dt, dt> (key, value)) – It inserts elements with a particular key in the unordered_map container.
2. erase() - It removes the specified key-value from the unordered_map.
3. clear() - It removes all the elements from the unordered_map.
4. size() - Returns the number of elements in the unordered_map.
5. empty() - Returns whether the unordered_map is empty or not.
6. find() - Returns an iterator to the specified element in the unordered_map if found, else returns the iterator to end.
7. count() - Returns the number of matches to element with the specified key-value in the unordered_map.

Syntax -

```
unordered_map<data_type, data_type> map_name;
```

Operation	Time Complexity	Space Complexity
insert()	O(1) or O(n) in worst case	O(1)
erase()	O(1)	O(1)
clear()	O(1)	O(1)
size()	O(1)	O(1)
empty()	O(1)	O(1)
find()	O(1) or O(n) in worst case	O(1)
count()	O(1)	O(1)

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    unordered_map<string, int> um;                    // Creating an unordered_map

    um.insert(pair<string, int> ("English", 45));     // Adding "English" as a key and "45" as a value in map
    um.insert(pair<string, int> ("Maths", 48));       // Adding "Maths" as a key and "48" as a value in map
    um["Science"] = 50;                               // Another way of adding elements into the map
    um["History"] = 41;

    um.erase("Science");         // Removes "Science" key and it's value from map
    cout<<um.size()<<endl;       // Outputs 3

    cout<<um.empty()<<endl;      // Outputs 0

    for(auto i: um){
        cout<<i.first<<" = "<<i.second<<endl;
    }

    //      Outputs
    //      History = 41
    //      Maths = 48
    //      English = 45
    return 0;
}
```

3. Algorithms

3.1 Binary Search

Syntax -

```
binary_search (start_address, end_address, value_to_find);
```

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    vector<int> v = {4, 13, 27, 55, 71, 98};

    cout<<binary_search(v.begin(), v.end(), 27)<<endl;    // Outputs 1 because 27 is present in v
    cout<<binary_search(v.begin(), v.end(), 44)<<endl;    // Outputs 0 because 44 is not present in v
    return 0;
}

```

It’s Binary Search, so make sure your array/vector/container has sorted elements.

Time Complexity - O(nlog(n))
Space Complexity - O(1)

3.2 Sort

Syntax -

For sorting in ascending(default) order -

```
sort (start_address, end_address);
```

For sorting in descending order -

```
sort (start_address, end_address, greater<data_type>());
```

For sorting in your own way/order -

```
sort (start_address, end_address, your_comparision_function);
```

```cpp
#include<bits/stdc++.h>
using namespace std;

bool compare(string s1, string s2){
    return s1.length() < s2.length();
}

int main(){
    vector<int> v = {4, 55, 98, 27, 13, 71};
    vector<int> v2 = {4, 55, 98, 27, 13, 71};
    vector<string> v3 = {"Apple", "Banana", "Pineapple", "Kiwi"};

    sort(v.begin(), v.end());               // Sorting in ascending order
    sort(v2.begin(), v2.end(), greater<int>());     // Sorting in descending order
    sort(v3.begin(), v3.end(), compare);     // Sorting by the lengths of the strings using our own compare function
    for(int i: v){
        cout<<i<<" ";           // Outputs 4 13 27 55 71 98
    }
    cout<<endl;
    for(int j: v2){
        cout<<j<<" ";           // Outputs 98 71 55 27 13 4
    }
    cout<<endl;
    for(string k: v3){
        cout<<k<<" ";           // Outputs Kiwi Apple Banana Pineapple
    }
    return 0;
}

```

Time Complexity - O(nlog(n))
Space Complexity - O(log(n))

3.3 Swap

Syntax -

```
swap (value1, value2);
```

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    int a = 15;
    int b = 27;

    cout<<a<<" "<<b<<endl;      // Outputs 15 27
    swap(a, b);
    cout<<a<<" "<<b<<endl;      // Outputs 27 15
    return 0;
}

```

Time Complexity - O(1)
Space Complexity - O(1)

3.4 Reverse

Syntax -

```
reverse (start_address, end_address);
```

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    vector<int> v = {4, 55, 98, 27, 13, 71};

    reverse(v.begin(), v.end());
    for(int i: v){
        cout<<i<<" ";           // Outputs 71 13 27 98 55 4
    }
    return 0;
}

```

Time Complexity - O(n)
Space Complexity - O(1)

3.5 Min and Max

Syntax -

For default comparison -

```
min (value1. value2);
max (value1. value2);
```

For comparison in your own way -

```
min (value1. value2, your_comparision_function);
max (value1. value2, your_comparision_function);
```

For comparison on a whole container or on a range of values -

```
*min_element (start_address, end_address);
*max_element (start_address, end_address);
*min_element (start_address, end_address, your_comparision_function);
*max_element (start_address, end_address, your_comparision_function);
```

```cpp
#include<bits/stdc++.h>
using namespace std;

bool compare(string s1, string s2){
    return s1.length() < s2.length();
}

int main(){
    int a = 15, b = 27;
    cout<<min(a, b)<<endl;      // Outputs 15
    cout<<max(a, b)<<endl;      // Outputs 27

    string s1 = "Orange", s2 = "Mango";
    cout<<min(s1, s2, compare)<<endl;       // Outputs Mango
    cout<<max(s1, s2, compare)<<endl;       // Outputs Orange

    vector<int> v = {4, 55, 98, 27, 13, 71};
    cout<<*min_element(v.begin(), v.end())<<endl;       // Outputs 4
    cout<<*max_element(v.begin(), v.end())<<endl;       // Outputs 98

    return 0;
}
```

Time Complexity - O(1), O(n) for range of values

Space Complexity - O(1)

4. Iterators

Iterators are used to point at the memory addresses of STL containers. They are primarily used in sequences of numbers, characters etc. They reduce the complexity and execution time of the program.

Syntax -

```
container_type<data_type>:: iterator iterator_name;
```

```cpp
#include<bits/stdc++.h>
using namespace std;

int main(){
    vector<int> v = {4, 55, 98, 27, 13, 71};

    vector<int>:: iterator ir;      // Declaring an iterator to a vector

    for(ir = v.begin();ir!=v.end();ir++){
        cout<<*ir<<" ";             // Outputs 4 55 98 27 13 71
    }
    return 0;
}

```

Trees

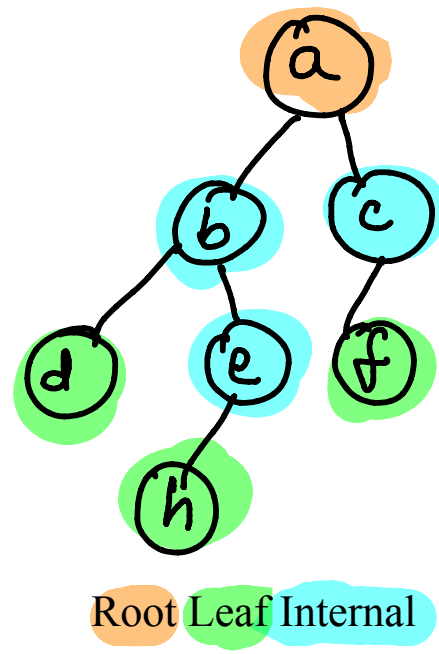
Tree - collection of tree-nodes

why trees?

1. Hiraichy
2. Computer system. (UNIX)

① Class Treenode
- data
- list<Treenode> children

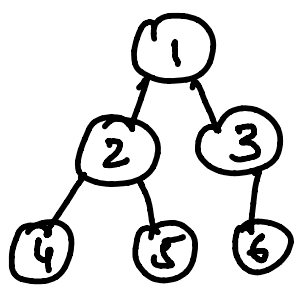
② Binary Tree → atmost 2 children (0,1,2)
- data
- leftchild
- rightchild



③ Types →

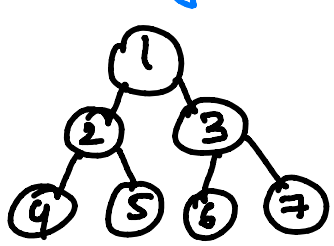
Ⓐ Complete Binary Tree
↳ all levels are completely filled except last one

Eg



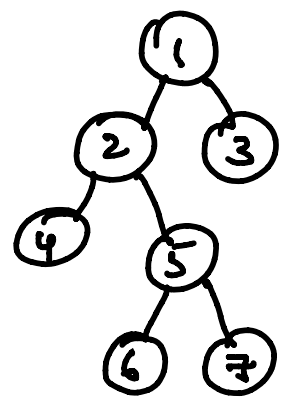
Ⓑ Perfect Binary Tree
↳ every internal node has exactly 2 children

Eg



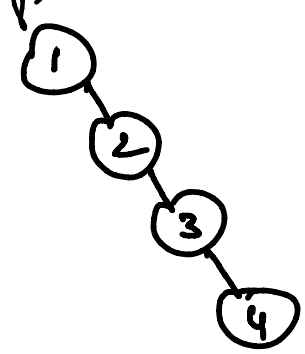
Ⓒ Full Binary tree
↳ if every node has 0 or 2 children

Eg



Ⓓ Skewed Binary Tree
(* used for finding complexity)
↳ all nodes have either one or no child.

Eg

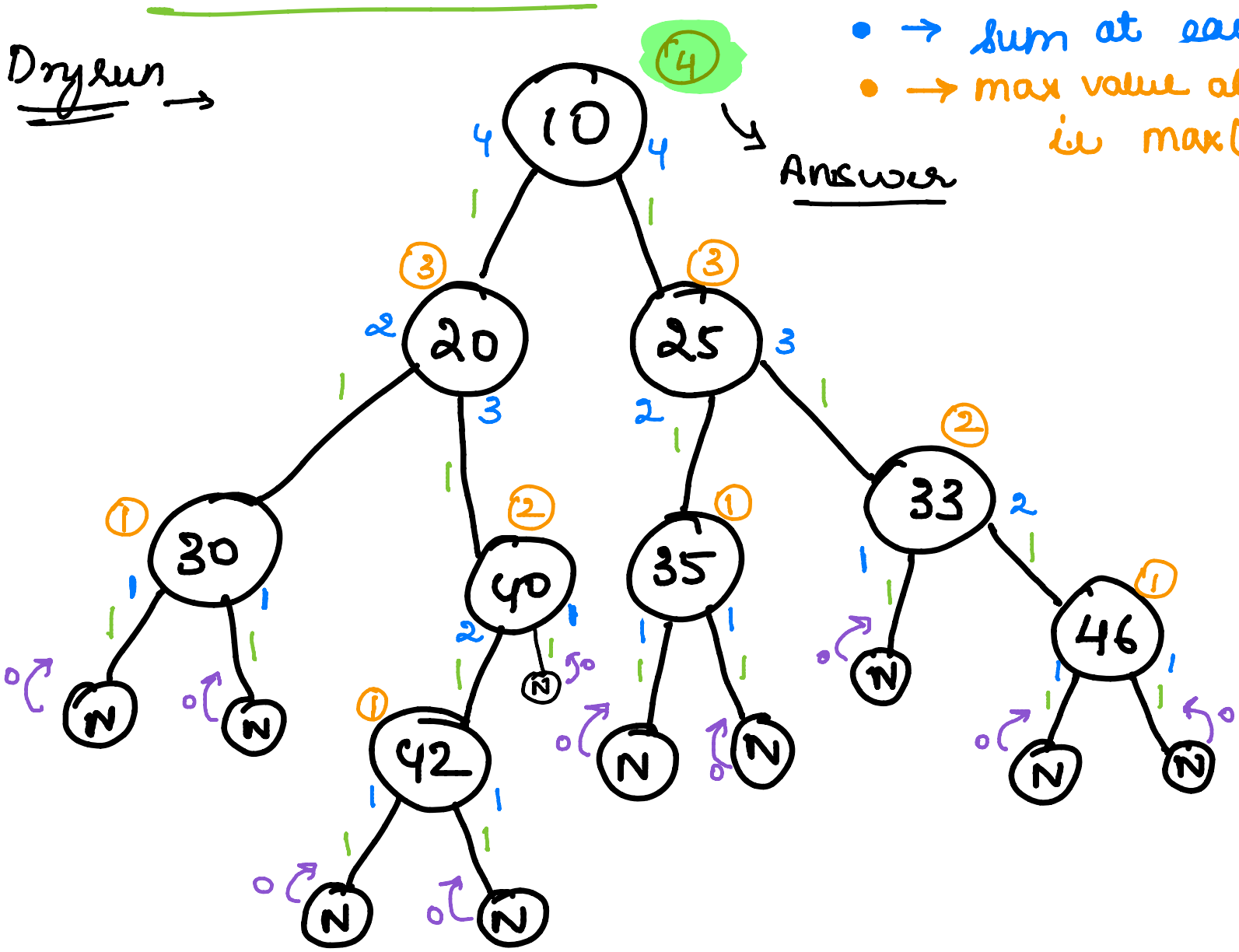


D1

① Depth of a binary tree (Max depth)

- → 1 added while returning.
- → sum at each node.
- → max value at node is max(left, right).

Dryrun →



if null
then ht = 0

- consider max at a node ie either left or right

TC → O(n)
SC → O(1)
AUX → O(h)
h → height

Code →

```C++
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nu
 *     TreeNode(int x) : val(x), left(nullptr), rig
 *     TreeNode(int x, TreeNode *left, TreeNode *r
 * };
 */
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(root == NULL) return 0;

        int lefth= 1+ maxDepth(root->left);
        int righth = 1+maxDepth(root->right);
        return max(lefth,righth);
    }
};
```

② Maximum depth of n-ary tree

Idea is same as previous problem, only implementation changes

Code →

C++

```cpp
/*
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> children;

    Node() {}

    Node(int _val) {
        val = _val;
    }

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};
*/

class Solution {
public:
    int maxDepth(Node* root) {
        if(root==NULL) return 0;
        int ans=0;
        for(int i=0;i<root->children.size();i++)
        {
            int tempans = maxDepth(root->children[i]);
            ans = max(ans,tempans);
        }
        return ans+1;
    }
};
```

D2

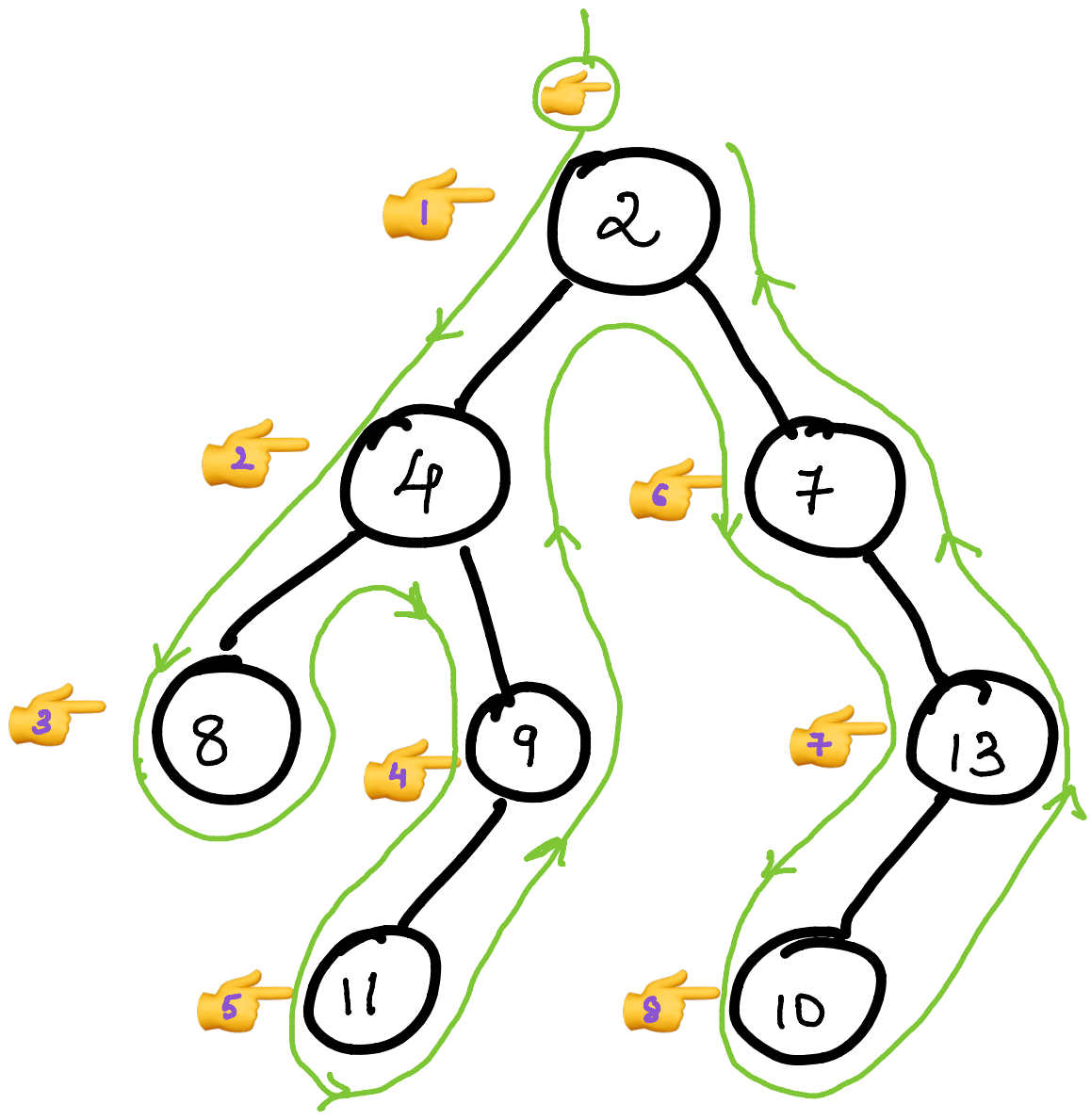
Traversals → DFS → Preorder
Inorder
Postorder

→ BFS → Level order

(A) Preorder →

processing order → node
left child
right child

Eg



* Point fingers as shown and traverse the tree starting from Root

* Order of visiting is the preorder traversal.

[2, 4, 8, 9, 11, 6, 13, 10]

Tc → O(n)

Sc → O(n) Recursive stack space → O(h) h → height.

③ Pre-order traversal of Binary tree

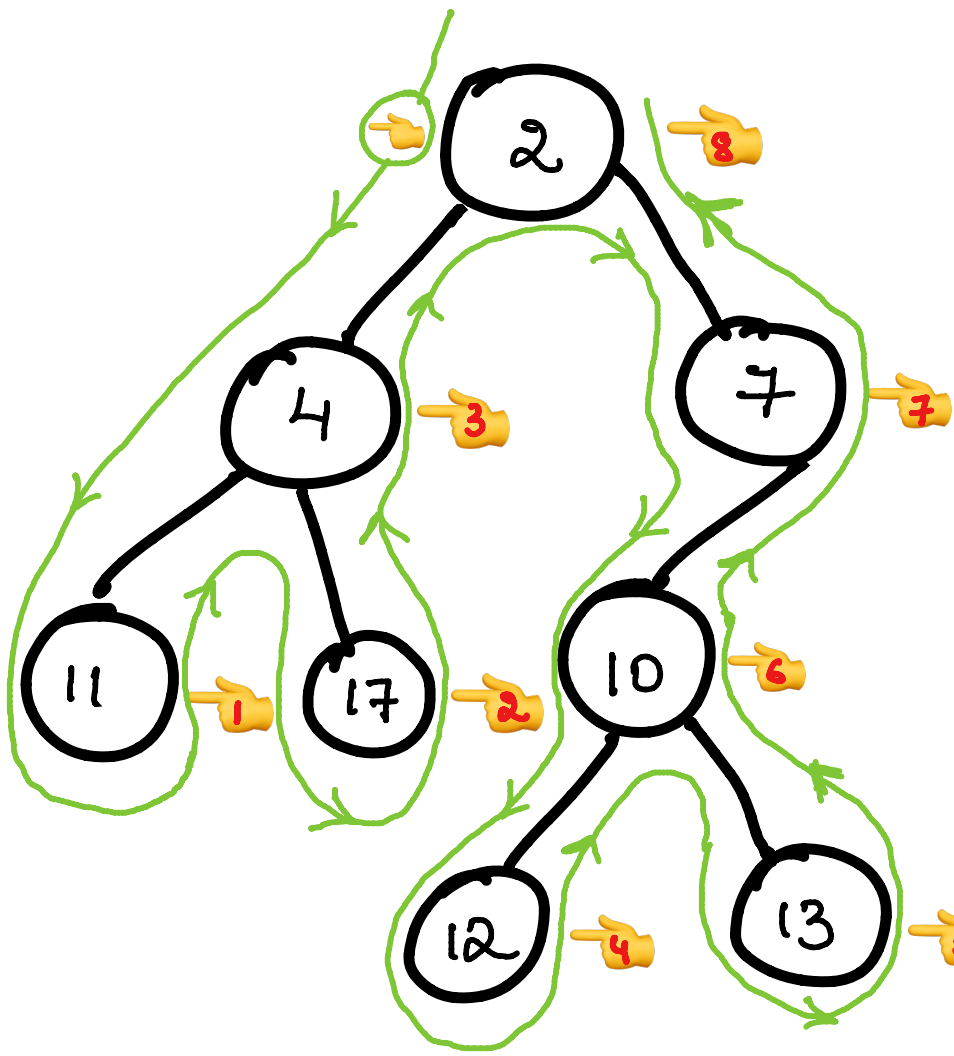
```cpp
class Solution {
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int>ans;
        Preorder(root,ans);
        return ans;
    }
    void Preorder(TreeNode* root,vector<int>&ans)
    {
        if(root == NULL) return;
        ans.push_back(root->val);
        Preorder(root->left,ans);
        Preorder(root->right,ans);
        return;
    }
};
```

④ Pre-order traversal of n-ary tree

```cpp
class Solution {
public:
    vector<int> preorder(Node* root) {
        vector<int>ans;
        Preorder(root,ans);
        return ans;
    }
    void Preorder(Node* root, vector<int>&ans)
    {
        if(root==NULL) return;
        ans.push_back(root->val);
        for(int i=0;i<root->children.size();i++)
        {
           Preorder(root->children[i],ans);

        }
        return;
    }
};
```

(B) Postorder → processing order → left child, right child, node

Eg



* Point fingers as shown and traverse the tree starting from Root

* Order of visiting is the postorder traversal.

[11, 17, 4, 12, 13, 10, 7, 2]

Tc → O(n)

Sc → O(n) Recursive stack space → O(h) h → height.

⑤ Postorder traversal of Binary tree

```
class Solution {
public:
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int>ans;
        Postorder(root,ans);
        return ans;
    }
    void Postorder(TreeNode* root,vector<int>&ans)
    {
        if(root == NULL) return;

        Postorder(root->left,ans);
        Postorder(root->right,ans);
         ans.push_back(root->val);
        return;
    }
};
```

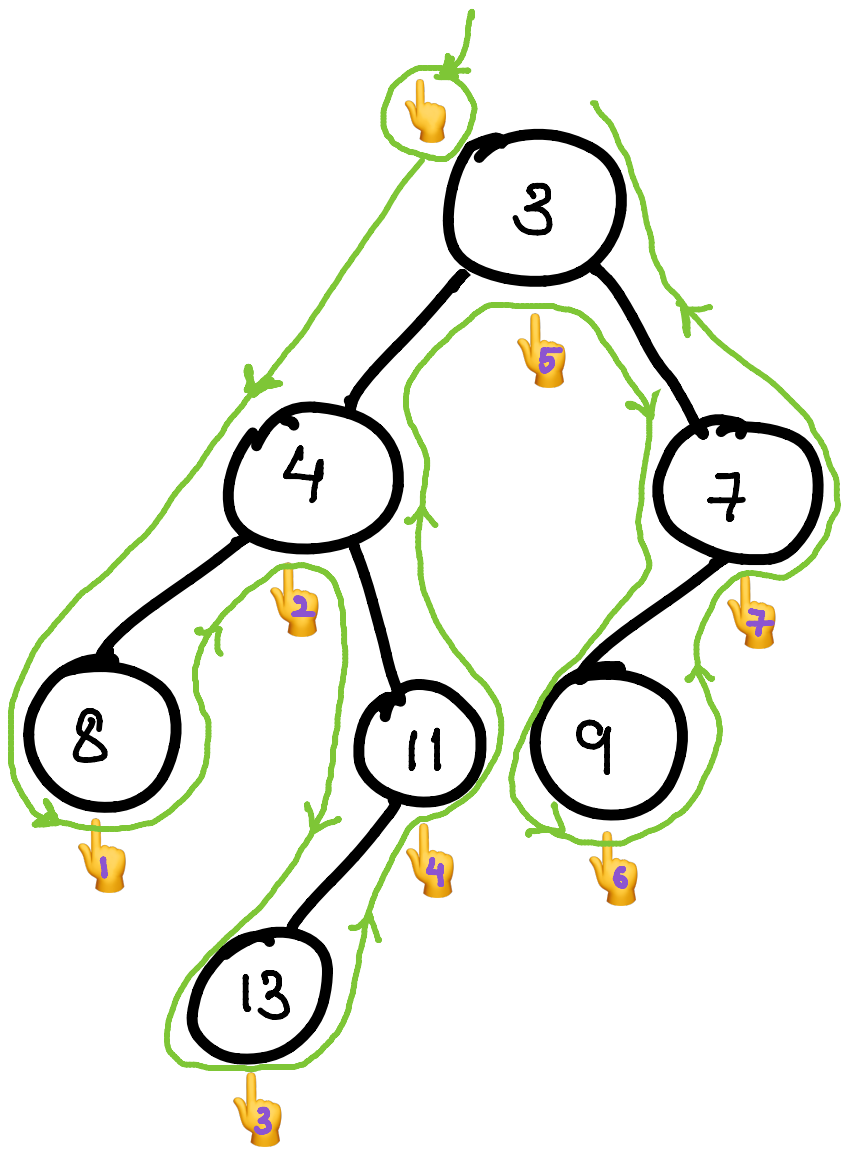
⑥ Postorder traversal of nary tree

```
class Solution {
public:
    vector<int> postorder(Node* root) {
        vector<int>ans;
        Postorder(root,ans);
        return ans;
    }
    void Postorder(Node* root, vector<int>&ans)
    {
        if(root == NULL) return;
        for(int i=0;i<root->children.size();i++)
        {
            Postorder(root->children[i],ans);
        }
        ans.push_back(root->val);
        return;
    }
};
```

(c) Inorder →

processing order → left child
node
right child

Eg



* Point fingers as shown and traverse the tree starting from Root

* Order of visiting is the Inorder traversal.

[8, 4, 13, 11, 3, 9, 7]

Tc → O(n)

Sc → O(n) Recursive stack space → O(h) h → height.

(7) In-order traversal of Binary tree

```cpp
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int>ans;
        Inorder(root,ans);
        return ans;
    }
    void Inorder(TreeNode* root, vector<int>&ans)
    {
        if(root==NULL) return;
        Inorder(root->left,ans);
        ans.push_back(root->val);
        Inorder(root->right,ans);
        return;
    }
};
```

Inorder traversal of n-ary tree

Approach:

The inorder traversal of an N-ary tree is defined as visiting all the children except the last then the root and finally the last child recursively.

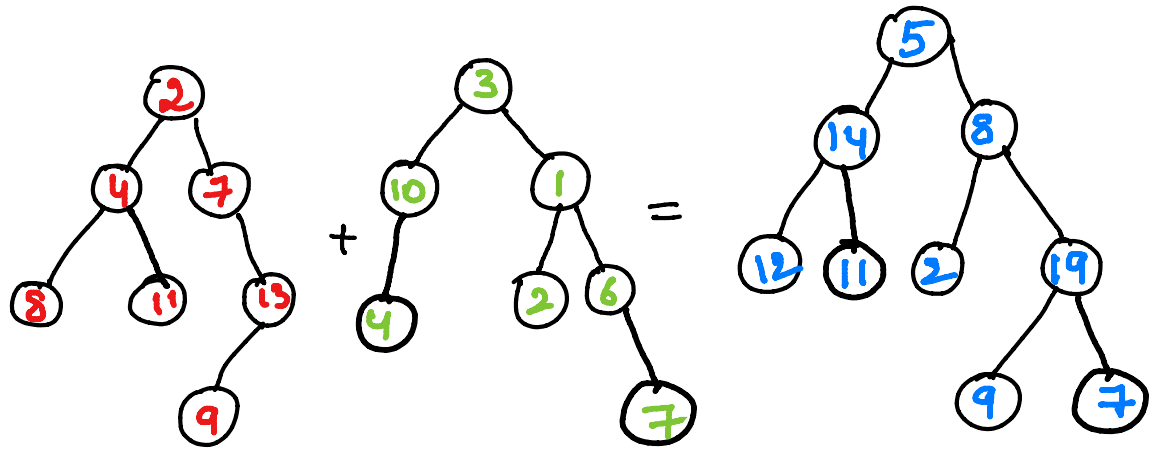
- Recursively visit the first child.
- Recursively visit the second child.
- .....
- Recursively visit the second last child.
- Print the data in the node.
- Recursively visit the last child.
- Repeat the above steps till all the nodes are visited.

```cpp
void inorder(Node *node)
{
    if (node == NULL)
        return;

    // Total children count
    int total = node->length;

    // All the children except the last
    for (int i = 0; i < total - 1; i++)
        inorder(node->children[i]);

    // Print the current node's data
    cout<< node->data << " ";

    // Last child
    inorder(node->children[total - 1]);
}
```

D3 (8) Merge two Binary trees →

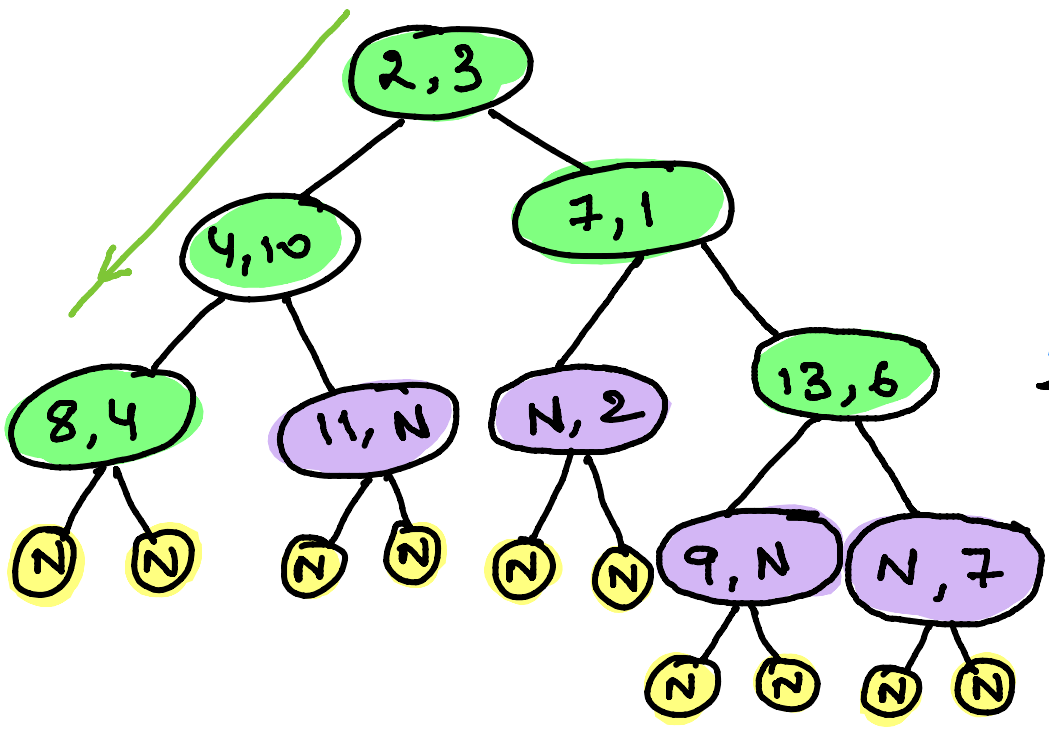
Given root nodes of 2 binary trees, return root of the sum tree

Eg

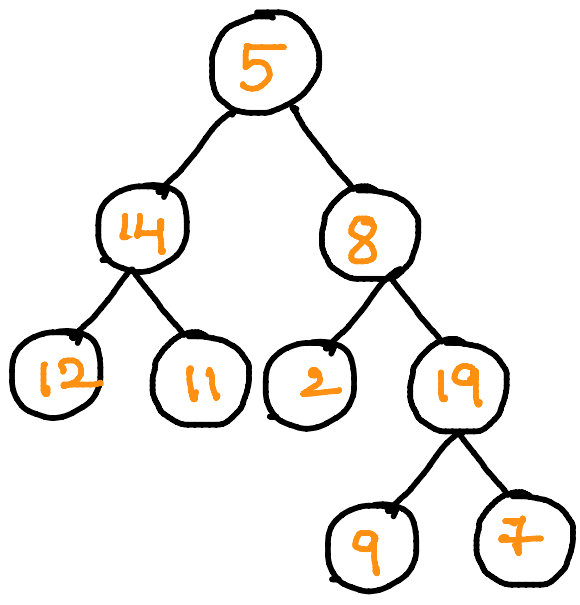


we will perform preorder traversal on the binary tree because the node/root needs to be processed first.

The recursive tree structure would be like :



condensed to



Resultant Tree

- NULL & NULL
- Node & NULL
- Node & Node

TC → O(n+m)

SC → O(max(n,m))

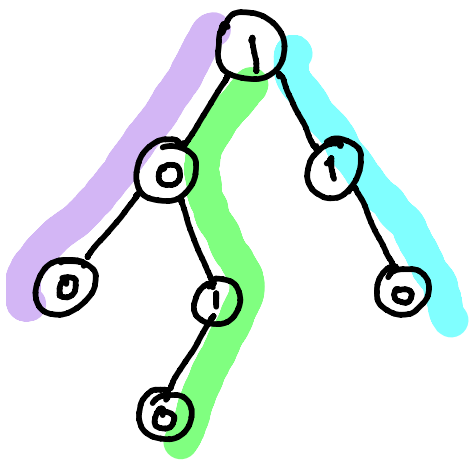
Recursive stack → O(max(h1,h2))

code →

```
class Solution {
public:
    TreeNode* merge(TreeNode* root1, TreeNode* root2){

        if(root1==NULL && root2==NULL)  return NULL;
        if(root1==NULL) return root2;
        if(root2==NULL) return root1;

        // Create new node to store sum
        TreeNode *newNode = new TreeNode(root1->val+root2->val);

        // Recursively call the left sub-trees and right sub-trees
        newNode->left = merge(root1->left, root2->left);
        newNode->right = merge(root1->right, root2->right);

        // return the new node
        return newNode;
    }

    TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
        return merge(root1, root2);
    }
};
```

(9) Sum of root to leaf paths →

Eg

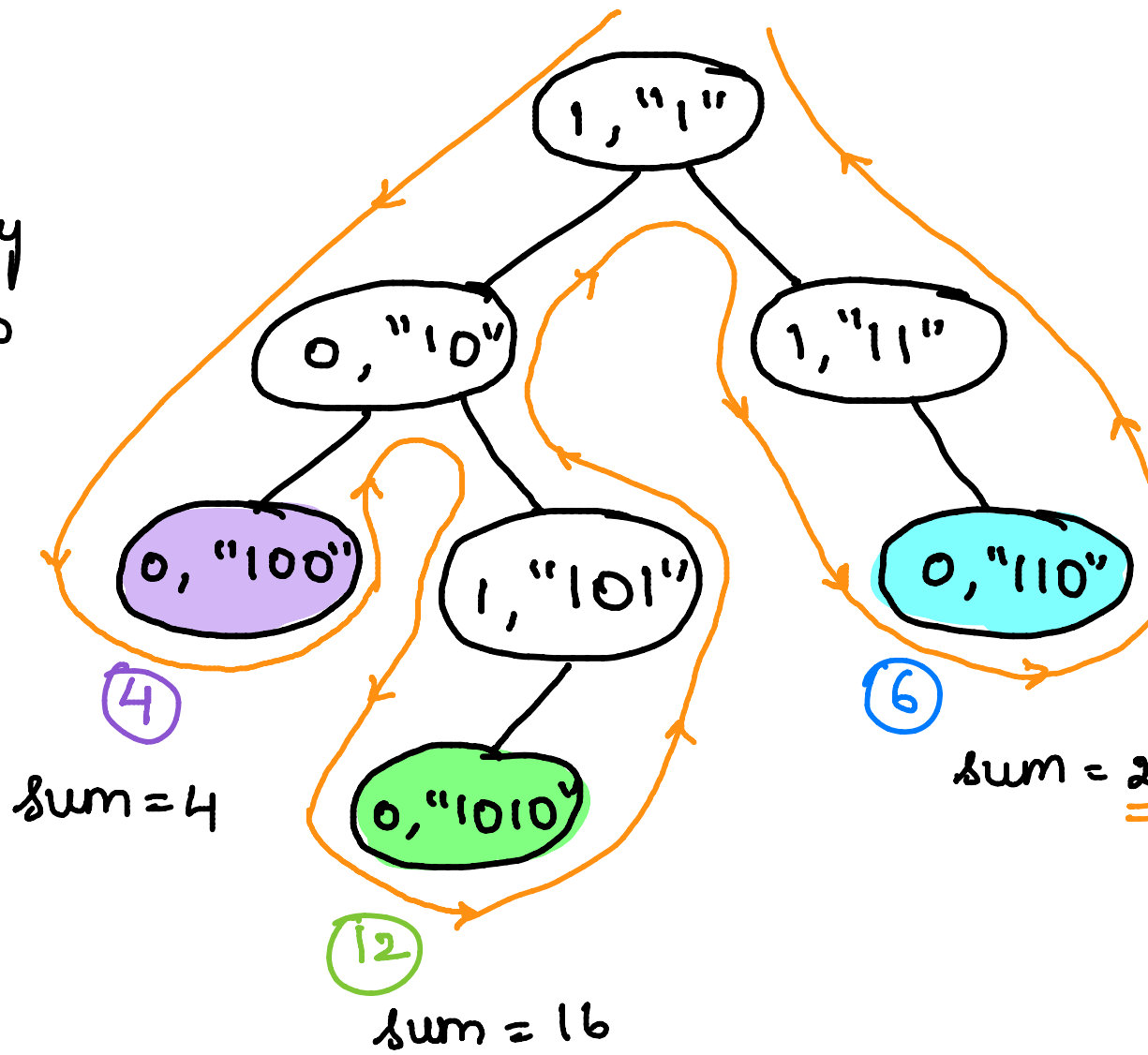


→ $(100)_2 + (1010)_2 + (110)_2$

$4 + 12 + 6$

$= 22$.

=



Initially

sum = 0

* If root becomes null convert string to integer & add to sum.

Time → O(n)

Space → O(n)

Recursive stack → O(h)

Code

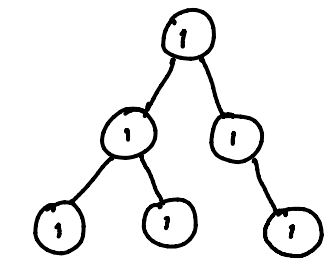
```
class Solution {
public:
    void rootToLeaf(TreeNode* root, string currentString,int* ans)
    {
        if(root->left== NULL && root->right==NULL)
        {
            currentString+=to_string(root->val);
            ans[0]+=stoi(currentString,0,2);
            return;
        }
        string curr=to_string(root->val);
        if(root->left!=NULL)
            rootToLeaf(root->left,currentString+curr,ans);
        if(root->right!=NULL)
            rootToLeaf(root->right,currentString+curr,ans);

    }
    int sumRootToLeaf(TreeNode* root) {
        int* ans=new int[1];
        ans[0]=0;
        rootToLeaf(root,"",ans);
        return ans[0];
    }
};
```

Note →

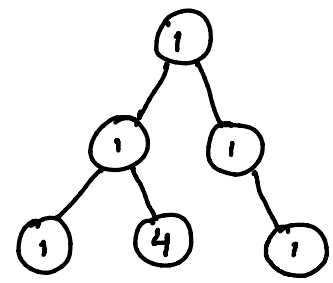
stoi() can take upto three parameters, the second parameter is for starting index and third parameter is for base of input number.

[to convert from binary to decimal we give it as 2]

(10) Univalued Binary Tree →

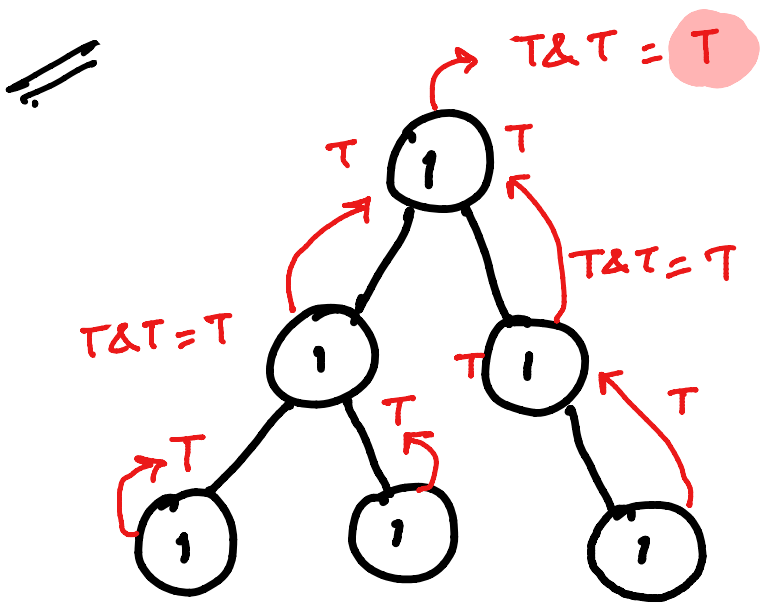


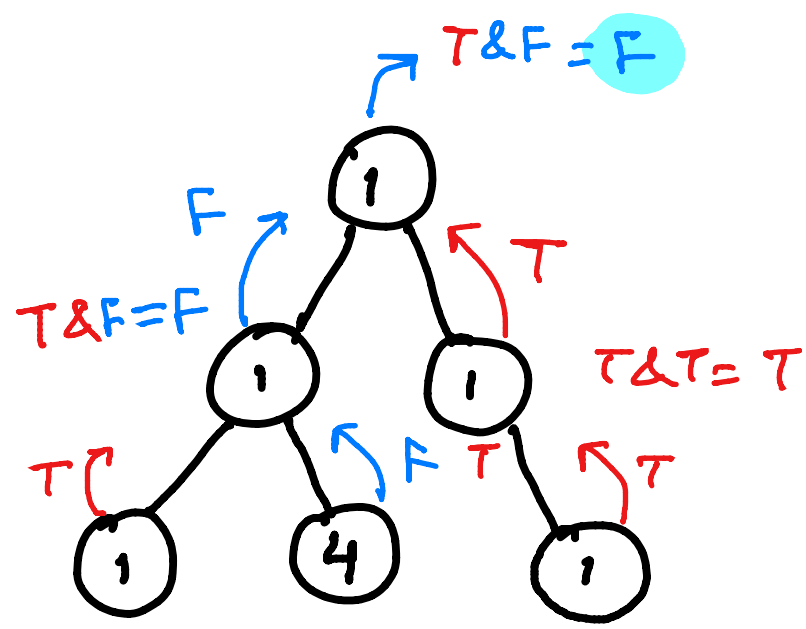
Is returns true



Is returns false

Eg





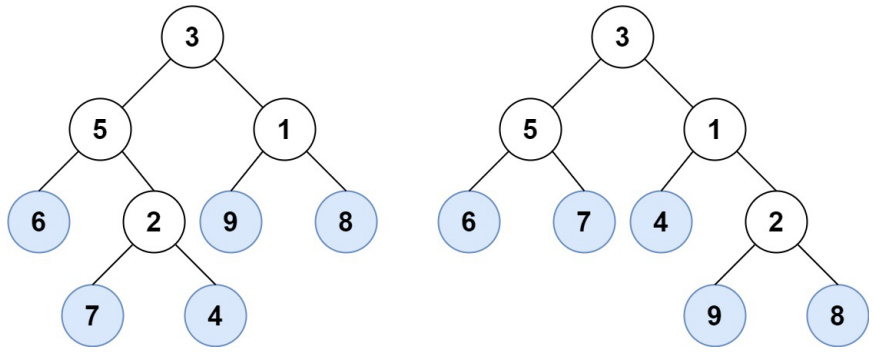
Code

```
class Solution {
public:
    bool isSame(TreeNode* root, int val){
        if(root==NULL)  return true;
        if(root->val!=val)  return false;

        bool left = isSame(root->left, val);
        bool right = isSame(root->right, val);

        return left && right;
    }

    bool isUnivalTree(TreeNode* root) {
        return isSame(root, root->val);
    }
};
```

(11) Leaf similar trees

↳ Return true if all leaves are in same order for both trees.

Eg



V_1 = 6, 7, 4, 9, 8 ⇒ $V_1 == V_2$

V_2 = 6, 7, 4, 9, 8 ↳ return true else false.

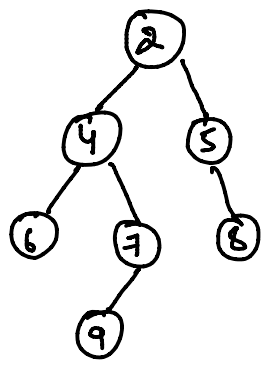
Code →

```
class Solution {
public:
    void traversal(TreeNode* root, vector<int>&v){
        if(root==NULL)
            return;

        if(root->left==NULL && root->right==NULL)
            v.push_back(root->val);

        if(root->left!=NULL)
            traversal(root->left, v);

        if(root->right!=NULL)
            traversal(root->right, v);
    }

    bool leafSimilar(TreeNode* root1, TreeNode* root2) {
        vector<int> a;
        vector<int> b;
        traversal(root1,a);
        traversal(root2,b);
        return a==b;
    }
};
```

DS (12) Binary Tree paths

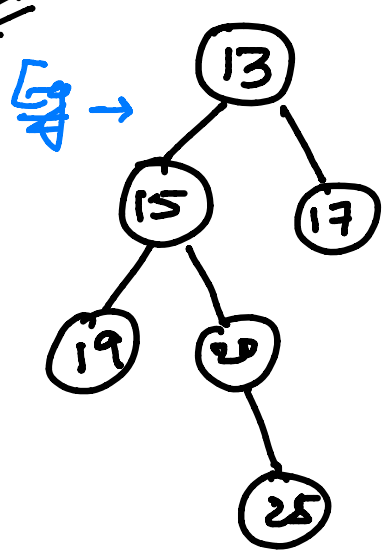
↳ given root print all the paths from root to leaf

Eg



⇒ ["2→4→6", "2→4→7→9", "2→5→8"]

Eg →



Result =

["13→15→19", "13→15→20→25", "13→17"]

Time complexity = O(n)

Space complexity = O(x) + O(h) → recursive stack.

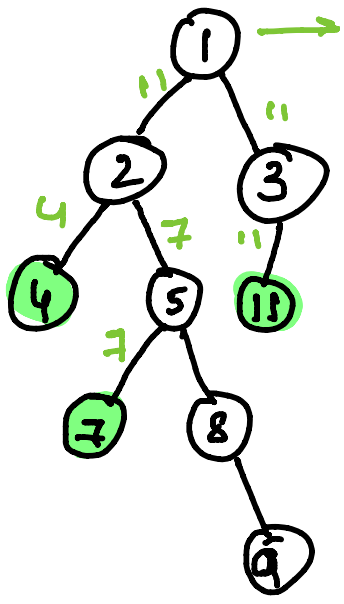
↳ Answer array

Code →

```
class Solution {
public:
    void pathFinder(TreeNode *root, vector<string> &res, string currPath){

        if(root==NULL)  return;

        // if leaf then add it's value to currentPath
        if(root->left == NULL && root->right==NULL){
            currPath += to_string(root->val);
            res.push_back(currPath);
            return;
        }
        // else add the node's value to path
        currPath += to_string(root->val)+"->";

        if(root->left)  pathFinder(root->left, res, currPath);
        if(root->right) pathFinder(root->right, res, currPath);
    }

    vector<string> binaryTreePaths(TreeNode* root) {
        vector<string> res;
        pathFinder(root, res, "");
        return res;
    }
};
```

(13) Sum of left leaves →

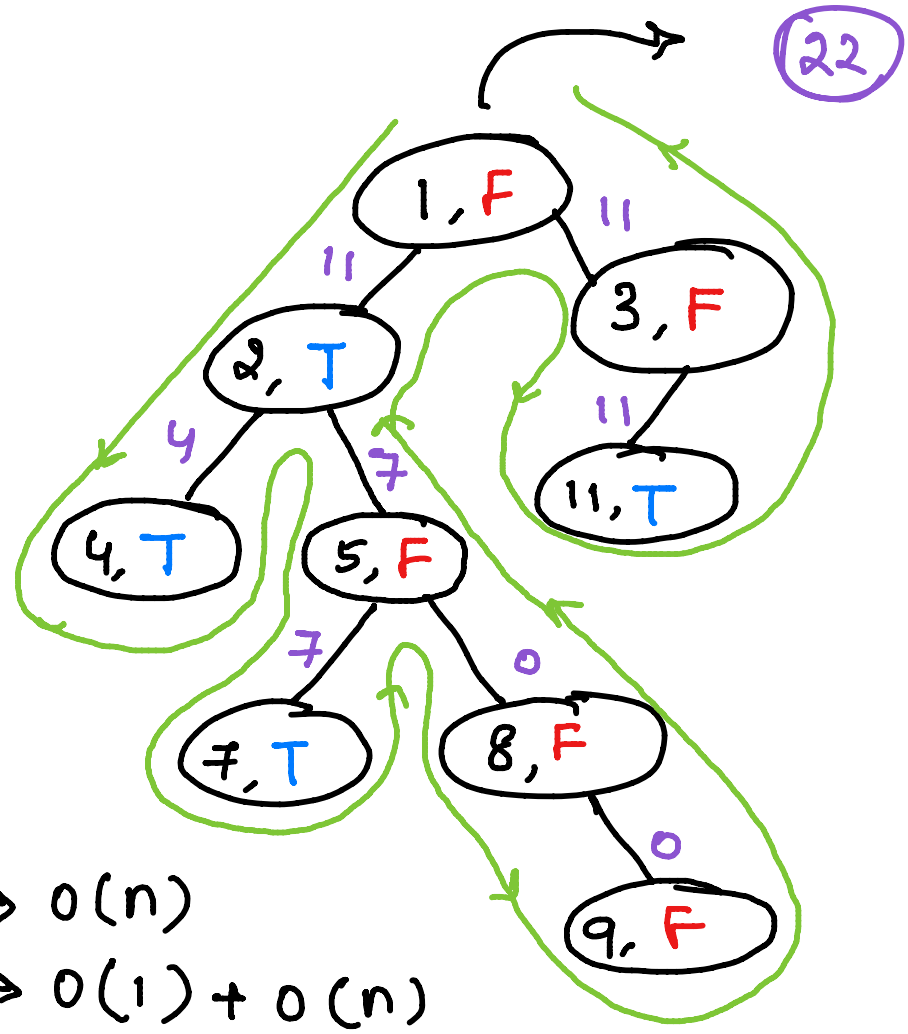
Eg



→ 22.

Result = 4 + 7 + 11

= 22.



Tc → O(n)

Sc → O(1) + O(n)

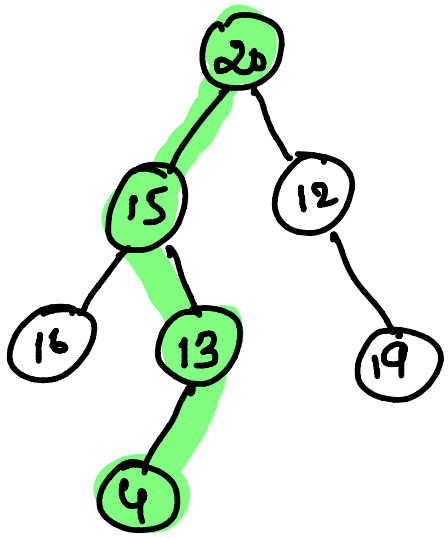
↳ stack

Code →

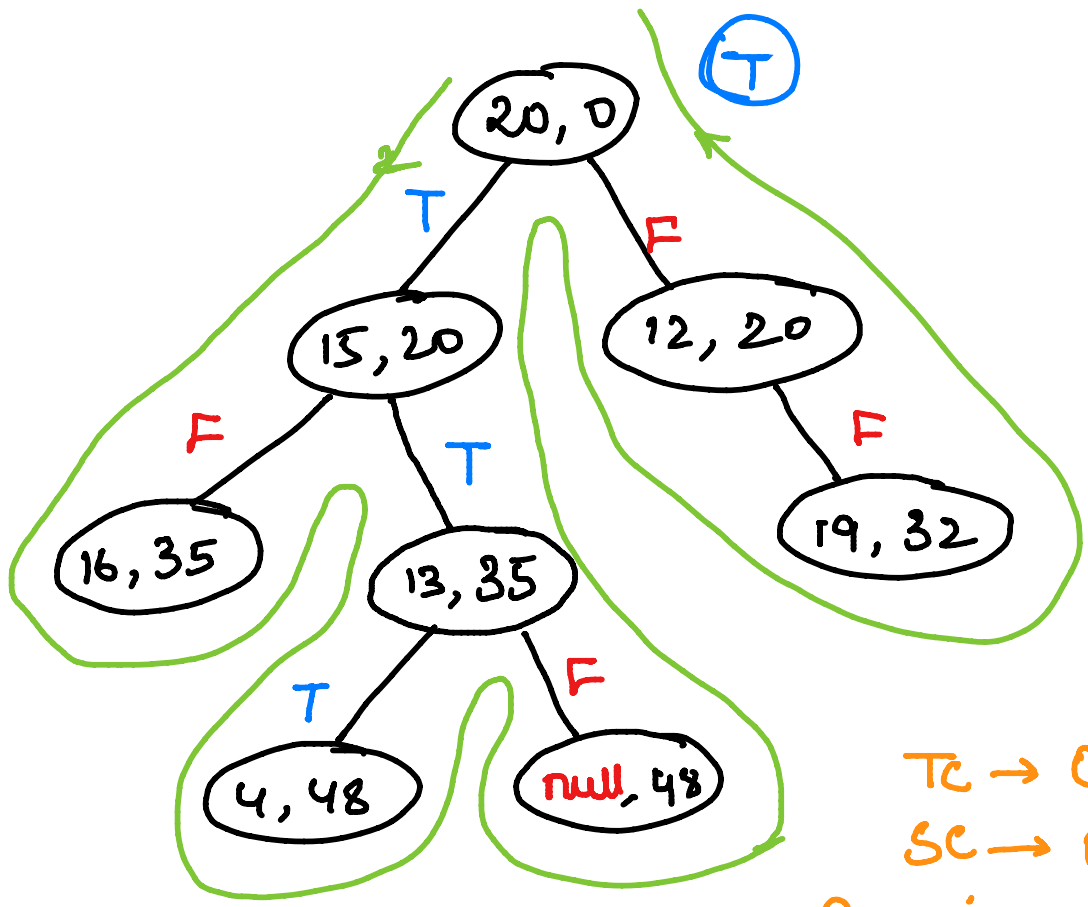
```
class Solution {
public:
    int leftLeafSum(TreeNode *root, bool leaf){
        if(root==NULL){
            return 0;
        }
        if(root->left==NULL && root->right==NULL && leaf){
            return root->val;
        }
        int ls = leftLeafSum(root->left, true);
        int rs = leftLeafSum(root->right, false);
        return ls+rs;
    }

    int sumOfLeftLeaves(TreeNode* root) {
        return leftLeafSum(root, false);
    }
};
```

(14) Path sum → Sum of all nodes from root to leaf is equal to target sum → then T else F.

Eg



target = 52



Tc → O(n)

Sc → O(1)

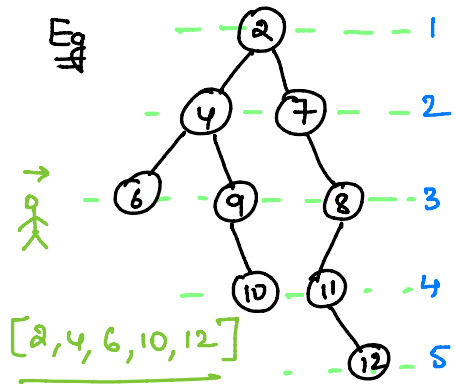
Recursive Stack → O(h)

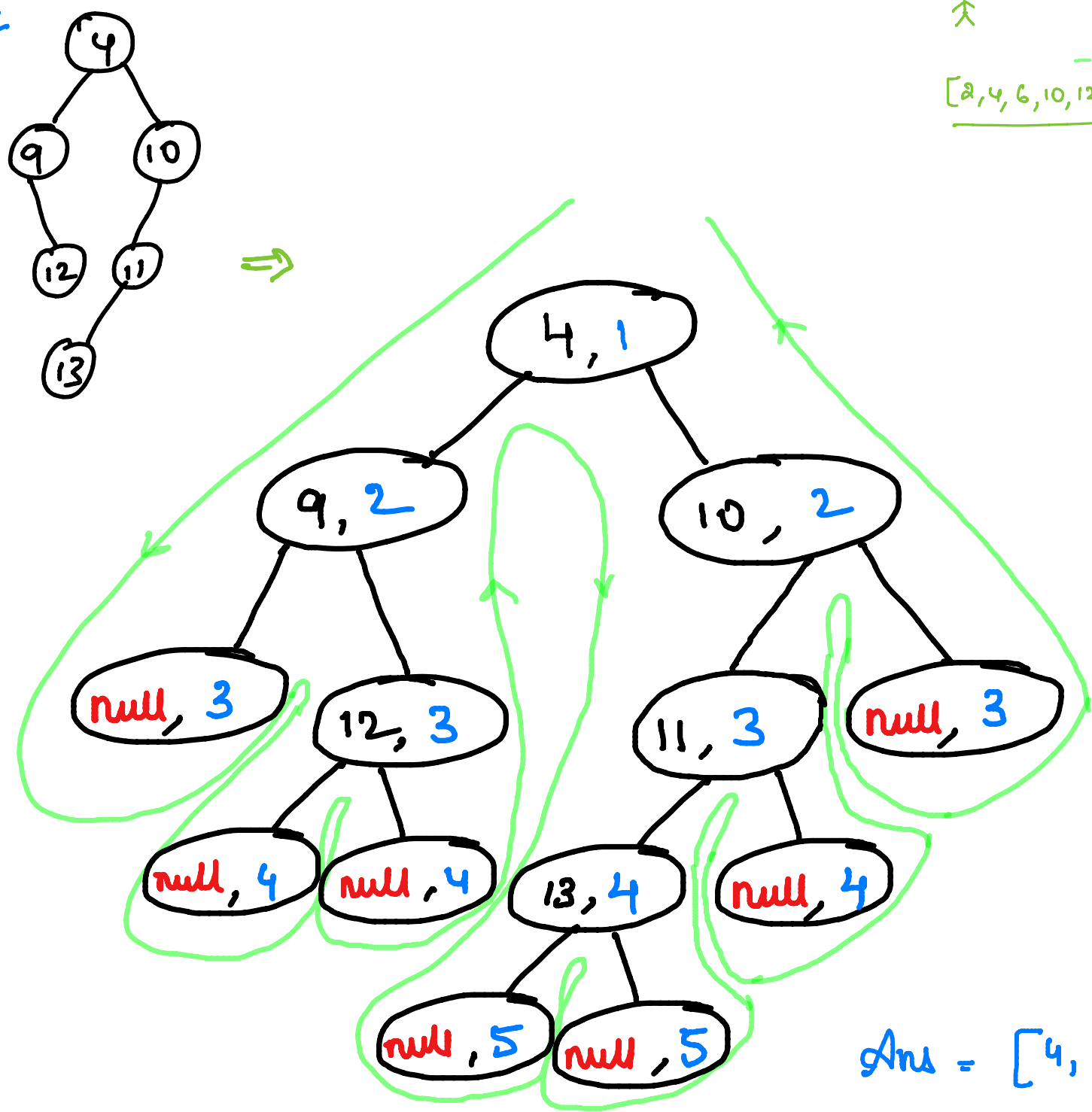
Code

```
class Solution {
public:
    bool pathSumUtil(TreeNode* root, int currSum, int targetSum){
        if(root==NULL)
            return false;

        if(root->left==NULL && root->right==NULL){
            return (currSum+root->val)==targetSum;
        }

        return pathSumUtil(root->left, currSum+root->val, targetSum)
            ||pathSumUtil(root->right, currSum+root->val, targetSum);
    }

    bool hasPathSum(TreeNode* root, int targetSum) {
        return pathSumUtil(root, 0, targetSum);
    }
};
```

D6 (15) Left view of a Binary Tree

Eg





Ans = [4, 9, 12, 13]

→ For every level traversed,
check if it already exist in the set,

if already exist then continue,
else add the node's value
to array & into the set

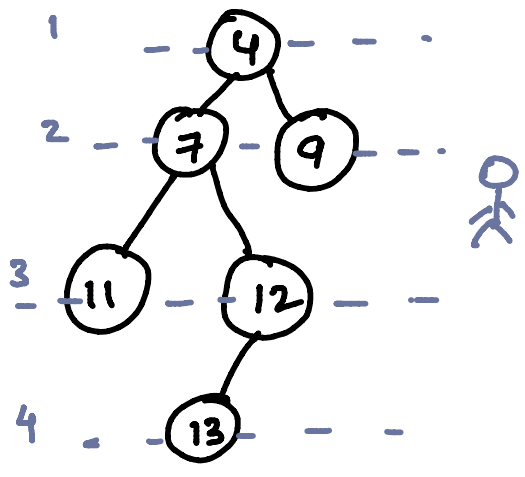
Tc → O(n)

Sc → O(n) + O(n) + O(h)
(result)

Code →

```
void viewGenerator(Node *root, vector<int> &res, set<int> &s, int currLevel){
    if(root==NULL) return;
    // if level is not reached, then add to result and the set
    if(s.find(currLevel)==s.end()){
        s.insert(currLevel);
        res.push_back(root->data);
    }
    // traverse the remaining branches
    viewGenerator(root->left, res, s, currLevel+1);
    viewGenerator(root->right, res, s, currLevel+1);
    return;
}

vector<int> leftView(Node *root)
{
    vector<int> res;
    set<int> s;
    viewGenerator(root, res, s, 0);
    return res;
}
```

(16) Right view of Binary Tree →



Result = [4, 9, 12, 13].

→ The entire approach to solve the problem is same as the left view of binary tree. Even the time complexities.

→ Only order of calling the branches change.

① right
② left

Code

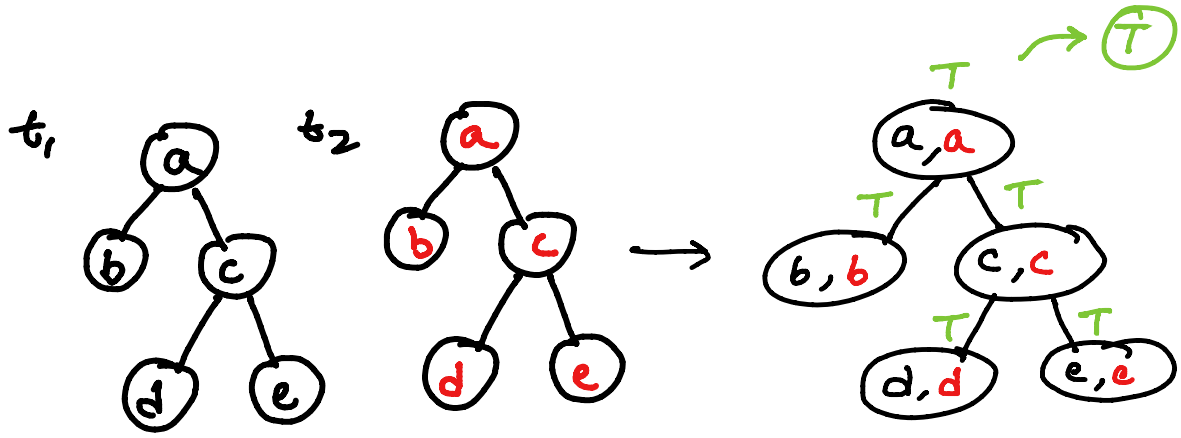
```
class Solution {
public:
    void viewGenerator (TreeNode* root, vector<int> &res, set<int> &s, int currLevel){
        if(root==NULL) return;
        // if level is not reached, then add to result and the set
        if(s.find(currLevel)==s.end()){
            s.insert(currLevel);
            res.push_back(root->val);
        }
        // traverse the remaining branch
        viewGenerator(root->right, res, s, currLevel+1);
        viewGenerator(root->left, res, s, currLevel+1);
        return;
    }
    vector<int> rightSideView(TreeNode* root) {
        vector<int> res;
        set<int> s;
        viewGenerator(root, res, s, 0);
        return res;
    }
};
```

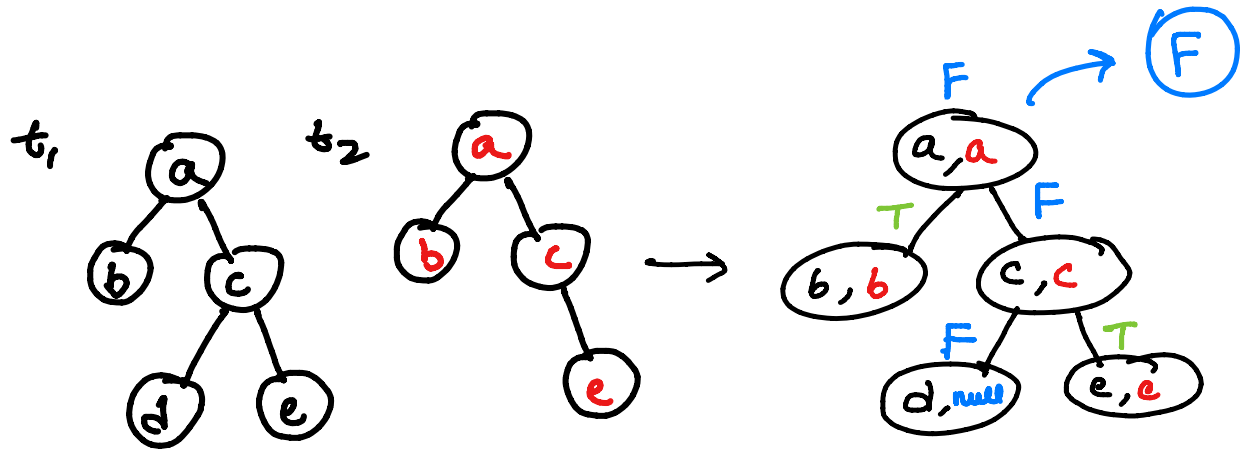
TC → O(n)

SC → O(n) + O(n) + O(h)

Result

(17) Same tree → return true if both tree are same else false





Tc → O(min(m, n))

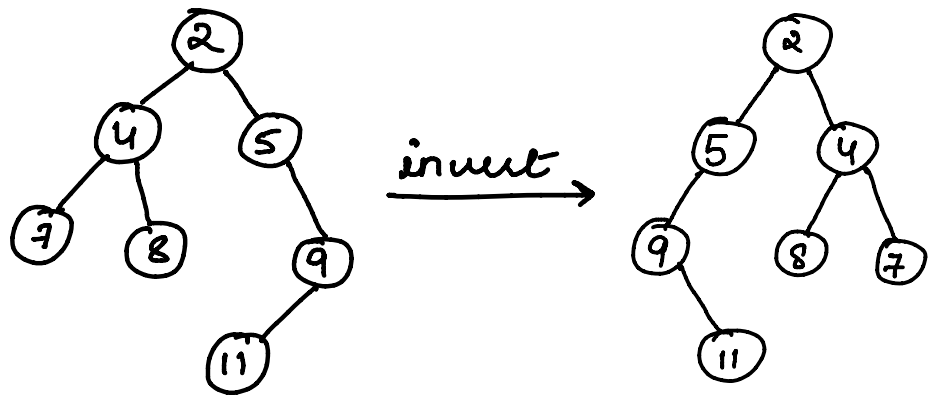
Sc → O(1) + O(min(h1, h2))

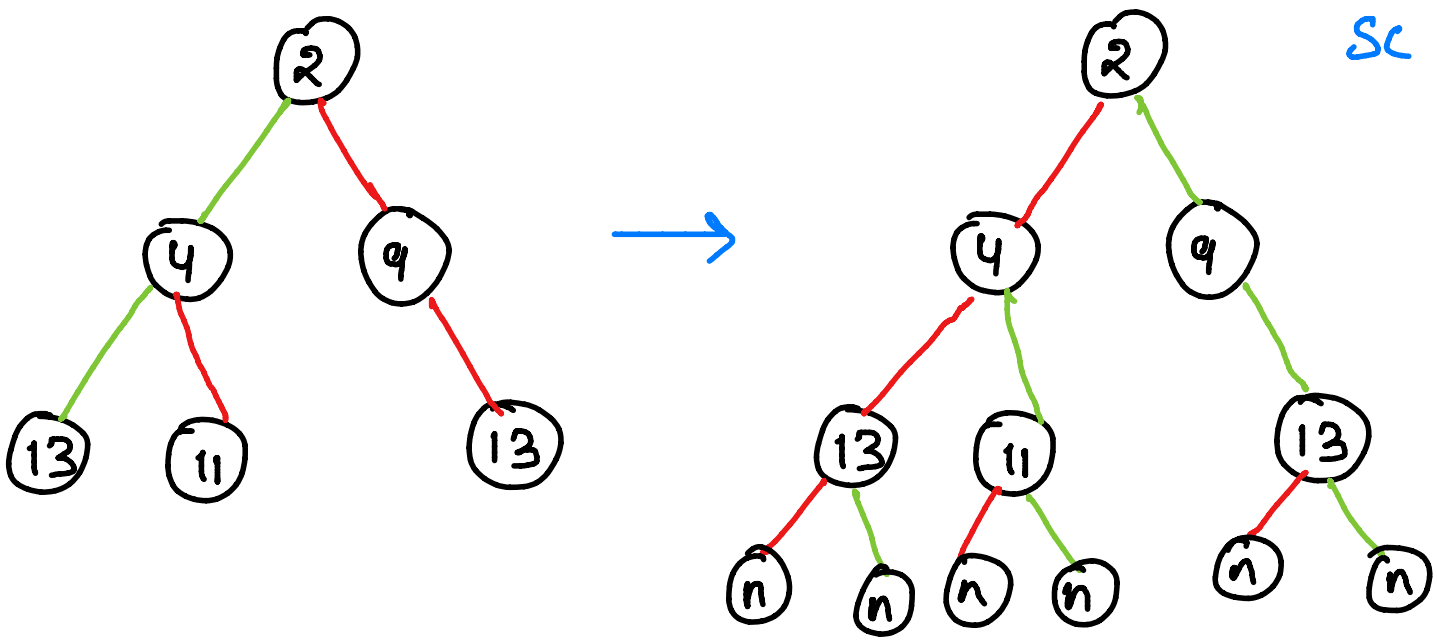
code →

```
class Solution {
public:

    bool isSameTree(TreeNode* p, TreeNode* q) {
        if(p==NULL && q==NULL) return true;

        if(p==NULL || q==NULL || p->val != q->val) return false;

        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);

    }
};
```

(8) Invert Binary Tree → given the root of BT, find its mirror img.

Eg





TC → O(n)

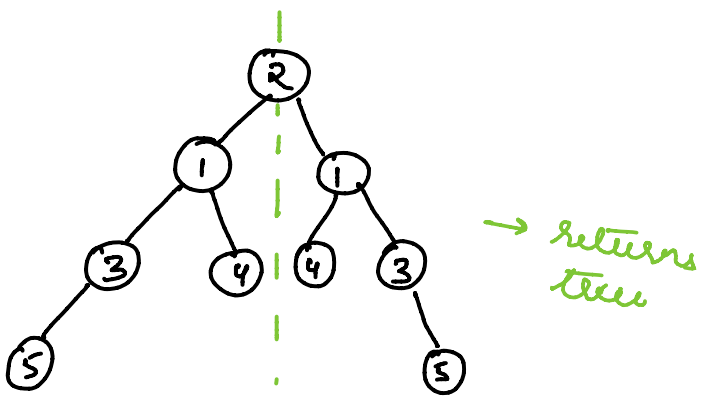
SC → O(n)+O(h)

Code →

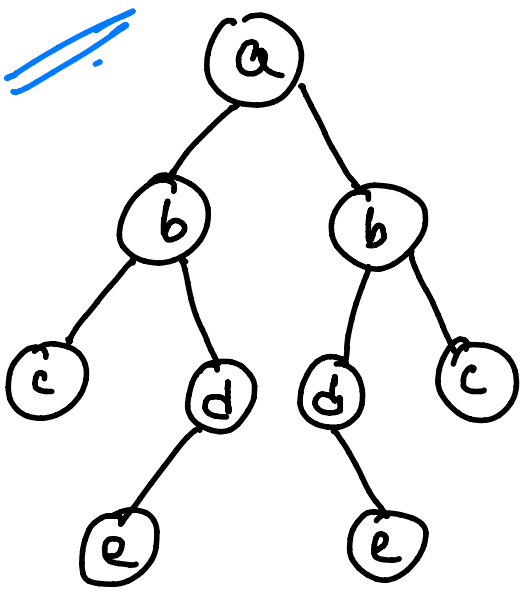
```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(root==NULL)  return root;

        /* invert the left and right sub-trees and store
            them separately */
        TreeNode *leftSub = invertTree(root->right);
        TreeNode *rightSub = invertTree(root->left);

        // attach the branches to root
        root->left = leftSub;
        root->right = rightSub;

        return root;
    }
};
```

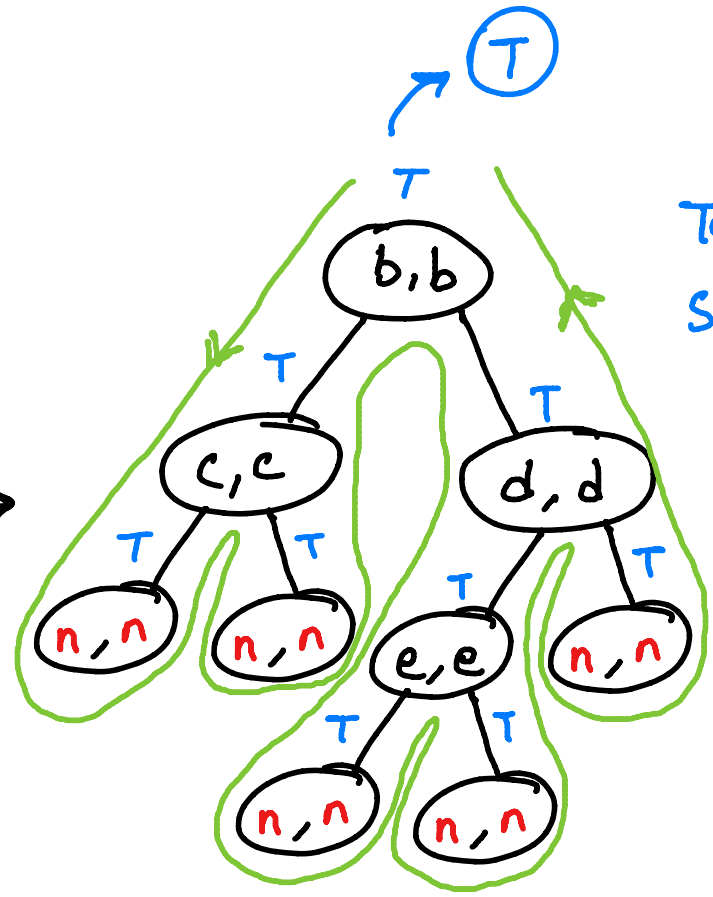
D7 (19) Symmetric Tree →

return true if left subtree is equal to right subtree, else return false



→ returns true





Tc → O(n)

Sc → O(1) + O(h)

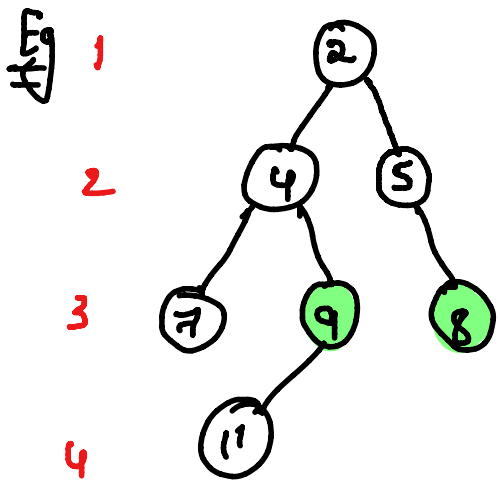
Code →

```
class Solution {
public:
    bool isMirror(TreeNode* l, TreeNode* r){

        if(l== NULL && r==NULL)
            return true;
        else if(l==NULL || r==NULL)
            return false;
        else if(l->val != r->val)
            return false;

        return isMirror(l->left,r->right) && isMirror(l->right, r->left);
    }
    bool isSymmetric(TreeNode* root) {
        if(root==NULL) return true;
        return isMirror(root->left, root->right);
    }
};
```

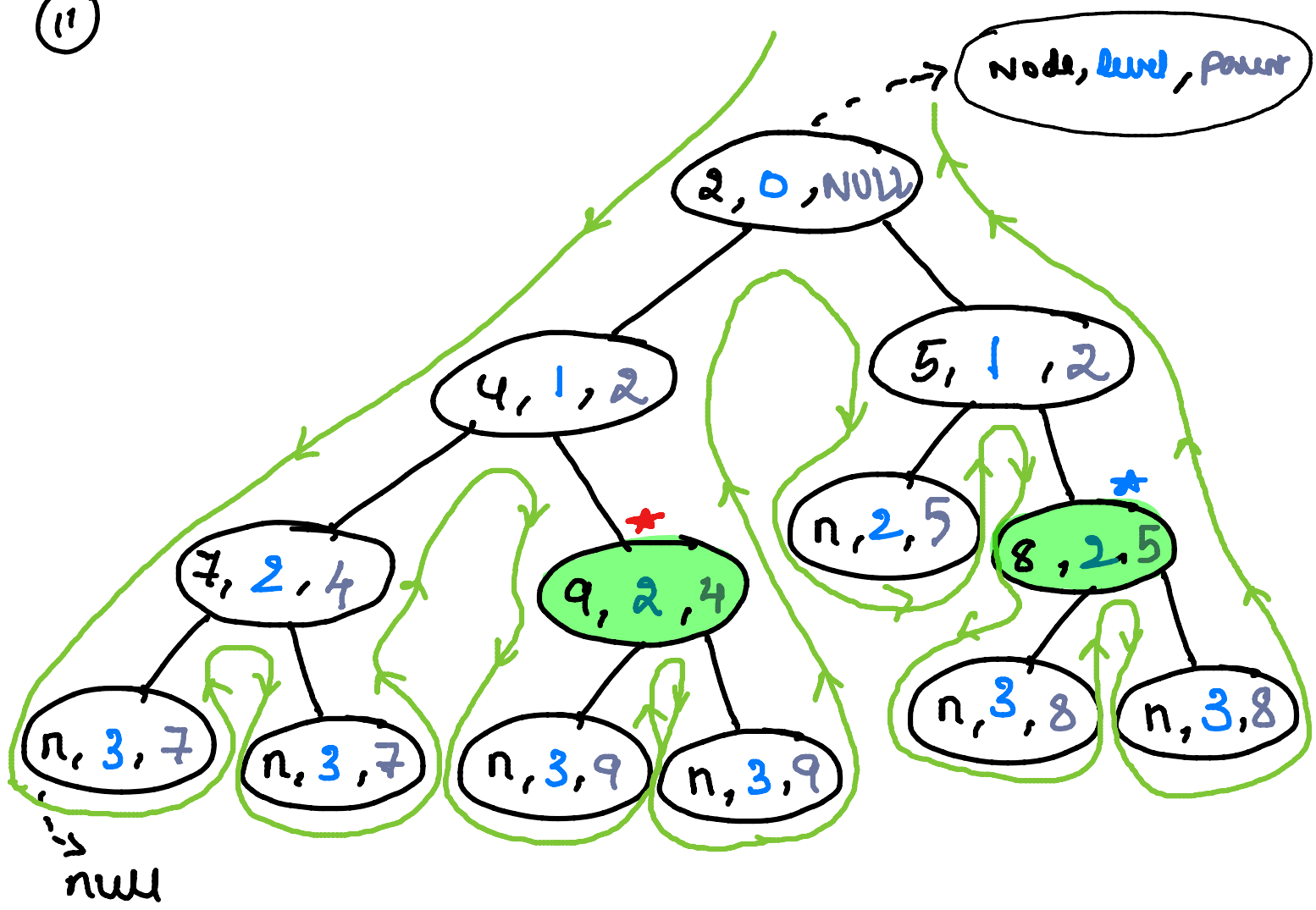
20 Cousins of a Binary Tree → given two nodes, find if they are cousins of each other.

↳ same level but diff parents.

Eg



x = 9
y = 8
} cousins



* at this step as value = 9 is
x is found store it's parent & level in separate variables
* later compare its value with other ocurance in Y
such that

1) x.parent != y.parent
2) x.level == y.level.

TC → O(n)

SC → O(1)

Recursive stack → O(n)

Code

```
class Solution {
public:
    void findNodes(TreeNode* root, int x, int y,int level[2],int parents[2],int currlevel,TreeNode* currparent)
    {
        if(root==NULL)return;
        if(root->val == x)
        {
            level[0]=currlevel;
            parents[0]=currparent->val;
        }
        if(root->val == y)
        {
            level[1]=currlevel;
            parents[1]=currparent->val;
        }
        findNodes(root->left, x, y, level, parents, currlevel+1, root);
        findNodes(root->right, x, y, level, parents, currlevel+1, root);
    }
    bool isCousins(TreeNode* root, int x, int y) {
        int level [2] = {-1,-1};
        int parents[2] = {-1,-1};
        findNodes(root, x, y, level, parents, 0, new TreeNode(-1));
        if(level[0]==level[1] && parents[0]!=parents[1])
            return true;
        return false;
    }
};
```