

DYNAMIC PROGRAMMING

KAPIL YADAV



Kapilyadav22



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

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

Tabulation :-

i.) Write base cases.
 $dp[0] = 0, dp[1] = 1.$

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

for(i= 2 ; i <= n ; i++)

{ $dp[i] = dp[i-1] + dp[i-2];$
 }
 action $dp[n];$

Optimization in Tabulation;

$\cancel{p_{n+2}} \rightarrow \cancel{p_n} \rightarrow i$

$\cancel{p_{n+2}} \rightarrow \cancel{p_n} \rightarrow i$ $\cancel{p_{n+2}} \rightarrow \cancel{p_n} \rightarrow i$

at last, i will exceed n , and
 p_{n+2} will be n , so return p_{n+2} .

p_{2^k} will be n , so return p_{2^k} .

for($i=2$; $i \leq N$; $i++$)
{

 int curr = $p_{2^k} + p_{2^{k-1}}$; $p_{2^k} = p_{2^k}$
 $p_{2^k} = curr$;

}

return p_{2^k} ;

$\boxed{TC = O(N)}$
 $\boxed{SC = O(1)}$

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) +
15        fibdp(currentindex-2,v) ;
16    v[currentkey]= fib;
17    return v[currentkey];
18 }
```

Tabulation:



```
1 int fib(int n) {  
2     if(n==0 || n==1)  
3         return n;  
4  
5     vector<int> dp(n+1,0);  
6     dp[0]=0;  
7     dp[1]=1;  
8  
9     for( int i =2;i<=n;i++)  
10    {dp[i]= dp[i-1] + dp[i-2];  
11    }  
12    return dp[n];  
13 }
```

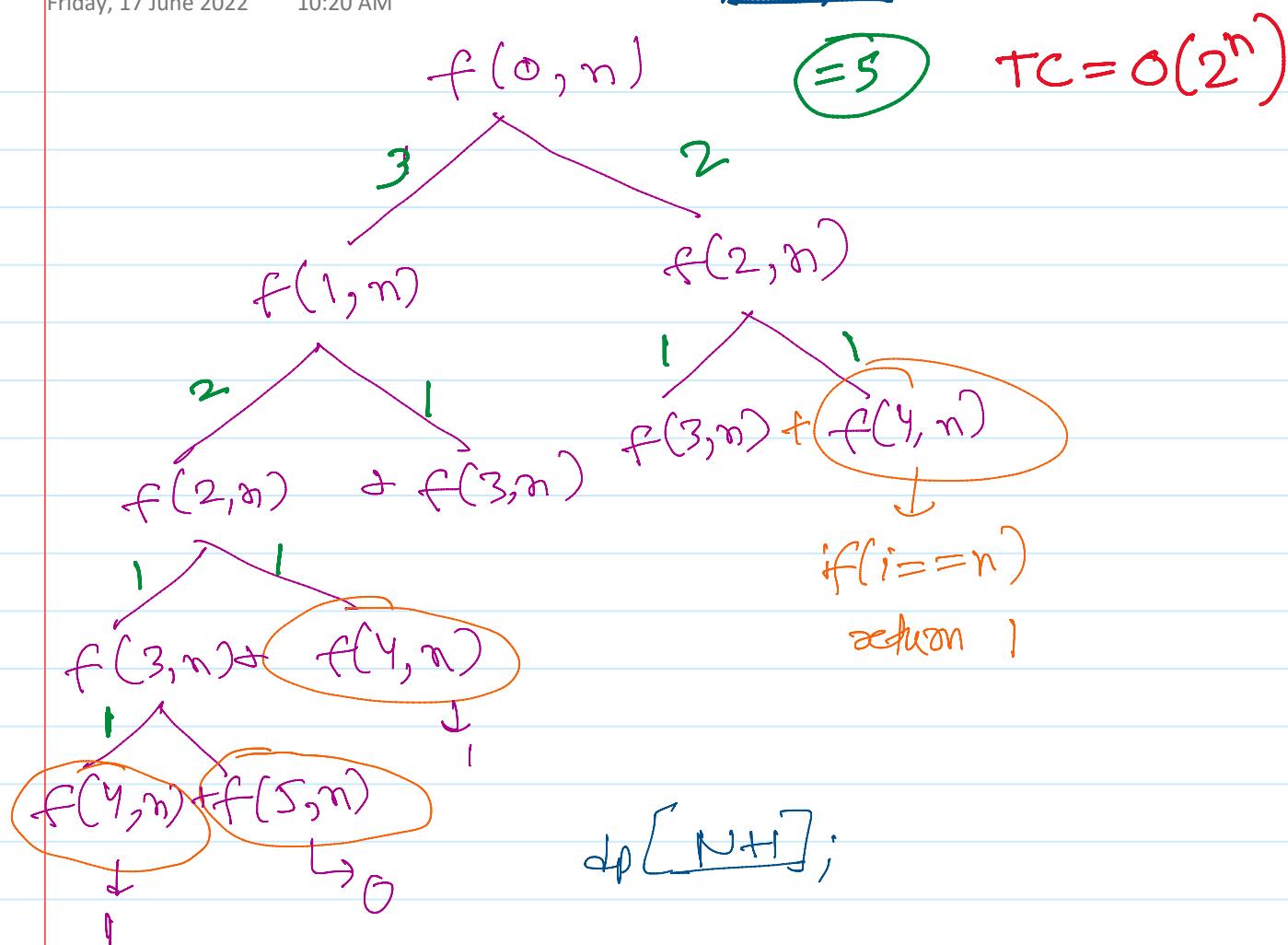
Space Optimization:



```
1 int fib(int n) {  
2     if(n==0 || n==1)  
3         return n;  
4  
5     int prev = 1;  
6     int prev2 = 0;  
7  
8     for( int i =2;i<=n;i++)  
9     {int curr= prev + prev2;  
10      prev2= prev;  
11      prev = curr;  
12  
13    }  
14    return prev;  
15 }  
16 }
```

LEC - 2, CLIMB STAIRS

Friday, 17 June 2022 10:20 AM



$dp[N+1]$

```

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

```

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

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

$return dp[currentIND] = f(currentIND+1, target, dp) + f(currentIND+2, target, dp);$

Q

Memoization:



```
1 int climbStairs(int n) {
2     vector<int> v(n+1,-1);
3     return noofways(0,n,v);
4 }
5
6     int noofways(int currentindex, int n, vector<int>&v)
7     {
8         if(currentindex==n)
9             return 1;
10        if(currentindex>n)
11            return 0;
12        int currentkey = currentindex;
13        if(v[currentkey]!=-1)
14            return v[currentkey];
15        else
16        {
17            int onestep = noofways(currentindex+1,n,v);
18            int twosteps = noofways(currentindex+2,n,v);
19            v[currentkey] = onestep+twosteps;
20            return onestep+twosteps;
21        }
22    }
```

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



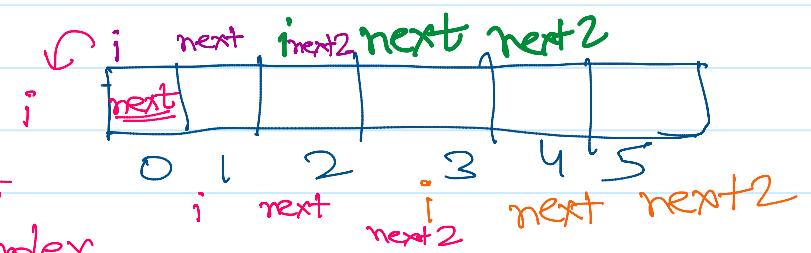
$TC = O(N)$

$SC = O(N)$

```
1 int climbStairs(int n) {
2     vector<int> dp(n+2,0);
3     dp[n+1]=0;
4     dp[n]=1;
5
6     for(int CurrInd = n-1; CurrInd>=0; CurrInd--)
7     {   dp[CurrInd] = dp[CurrInd+1] + dp[CurrInd+2];
8     }
9     return dp[0];
10 }
```

Space Optimization :-

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



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

$$TC = O(N)$$

$$SC = O(1)$$

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/kapilyadav22)

Lecture-3 Frog JUMP

17 June 2022 11:14

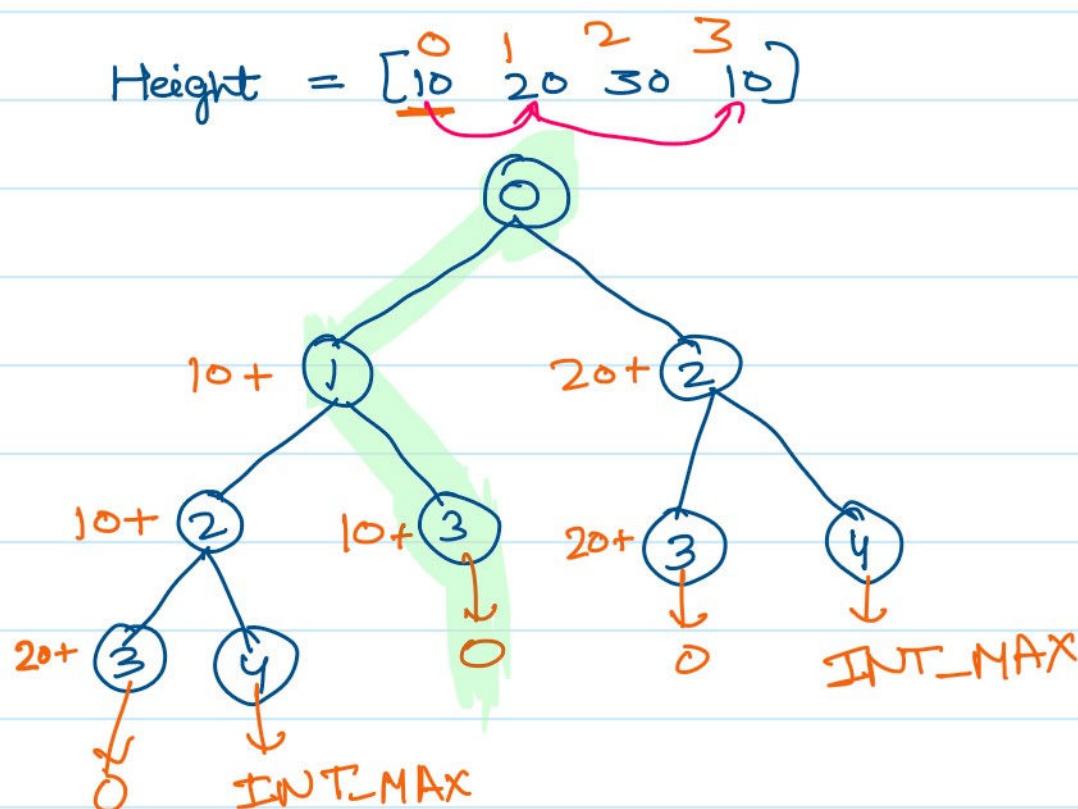
It is same as Min cost to climb the stairs (Leetcode), just the cost is added as
 $\text{abs}(\text{heights}[\text{currindex}] - \text{height}[\text{currindex}+1])$ in case of jumping one step.
 $\text{abs}(\text{heights}[\text{currindex}] - \text{height}[\text{currindex}+2])$ in case of jumping two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



$$\text{minimum total energy} = 20 \cdot 0 \rightarrow (1 \rightarrow 3)$$

Memoization:

```
● ● ●
```

```
1 int recursive(int CurrInd, int targetInd, vector<int>
    &heights, vector<int>& dp)
2 { if(CurrInd>targetInd)
3     return 10000000;
4     if(CurrInd==targetInd)
5         return 0;
6     if(dp[CurrInd]!=-1)
7         return dp[CurrInd];
8     int stepone = abs(heights[CurrInd+1]-
    heights[CurrInd]) +
    recursive(CurrInd+1,targetInd,heights,dp);
9     int steptwo = abs(heights[CurrInd+2]-
    heights[CurrInd]) +
    recursive(CurrInd+2,targetInd,heights,dp);
10    return dp[CurrInd] = min(stepone,steptwo);
11 }
```

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

```
13 int frogJump(int n, vector<int> &heights)
14 {     vector<int> dp(n,-1);
15     return recursive(0,n-1,heights,dp);
16 }
```

Tabulation:

```
● ● ●
```

```
1 int frogJump(int n, vector<int> &heights)
2 {     vector<int> dp(n+1,0);
3     dp[n] = 100000;
4     dp[n-1] = 0;
5     for(int CurrInd =n-2;CurrInd>=0;CurrInd--)
6     {     int stepone = abs(heights[CurrInd+1]-heights[CurrInd]) + dp[CurrInd+1];
7         int steptwo = abs(heights[CurrInd+2]-heights[CurrInd]) + dp[CurrInd+2];
8         dp[CurrInd] = min(stepone,steptwo);
9     }
10    return dp[0];
11 }
```

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

Space Optimization:



```
1 int frogJump(int n, vector<int> &heights)
2 {    int next2 = 100000;
3     int next = 0;
4     for(int CurrInd =n-2;CurrInd>=0;CurrInd--)
5     {   int stepone = abs(heights[CurrInd+1]-heights[CurrInd]) + next;
6         int steptwo = abs(heights[CurrInd+2]-heights[CurrInd]) + next2;
7         next2 =next;
8         next = min(stepone,steptwo);
9     }
10    return next;
11 }
```

$TC = O(N)$

$SC = O(1)$

[LinkedIn](#)/kapilyadav22

LECTURE-4 FROG JUMP 2 (ATCODER)

Friday, 17 June 2022 12:36 PM

Problem Statement

There are N stones, numbered $1, 2, \dots, N$. 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 $i + 1, i + 2, \dots, i + K$. 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 codeforces Jump frog problem.

Memoization:

```
1 Frog Jump with distance K
2 //Take input cases from codestudio
3 #include<iostream>
4 #include<vector>
5 #include<climits>
6 using namespace std;
7
8 int recursive(int CurrInd, int targetInd,int k,vector<int>& heights,vector<int>& dp)
9 { if(CurrInd>targetInd)
10     return 10000000;
11     if(CurrInd==targetInd)
12         return 0;
13     if(dp[CurrInd]!=-1)
14         return dp[CurrInd];
15     int mincost = INT_MAX;
16     for(int i=1;i<=k && i<=targetInd;i++)
17     { int stepk = abs(heights[CurrInd+i]-heights[CurrInd]) +
      recursive(CurrInd+i,targetInd,k,heights,dp);
18         mincost= min(stepk,mincost);
19     }
20     dp[CurrInd] = mincost;
21     return dp[CurrInd];
22 }
23
24 int frogJump(int k,int n,vector<int>& heights)
25 {     vector<int> dp(n,-1);
26     return recursive(0,n-1,k,heights,dp);
27 }
28
29 int main() {
30     int t,n;
31     cin>>t;
32     while(t--) {
33         cin>>n;
34         vector<int> heights(n);
35         for(int i=0;i<n;i++)
36             cin>>heights[i];
37         cout<<frogJump(2,n,heights)<<endl;
38     }
39     return 0;
40 }
```

$$TC = O(N) + O(k)$$

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

Tabulation:

```
● ○ ●
1 int frogJump(int k,int n,vector<int>& heights)
2 {
3     vector<int> dp(n+1,0);
4     dp[n] = 100000;
5     dp[n-1] = 0;
6     for(int CurrInd = n-2; CurrInd >= 0; CurrInd--)
7     {
8         int mincost = INT_MAX;
9         for(int i=1; i <= k && i < n; i++)
10        {
11            int stepk = abs(heights[CurrInd+i]-heights[CurrInd]) + dp[CurrInd+i];
12            mincost = min(stepk, mincost);
13        }
14    }
15    dp[CurrInd] = mincost;
16 }
17 return dp[0];
18 }
```

$TC = O(N) \times O(K)$
 $SC = O(N)$

- 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 optimize the space, by taking only K next elements, but in WC \Rightarrow 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.

[LinkedIn/kapilyadav22](#)

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.
```

$$\text{nums} = \underline{2} \ \underline{7} \ \underline{9} \ \underline{3} \ \underline{1}$$

Here Robber can rob $2 + 9 + 1 = 12$

$$\rightarrow 2 + 3 = 5$$

$$\rightarrow 2 + 1 = 3$$

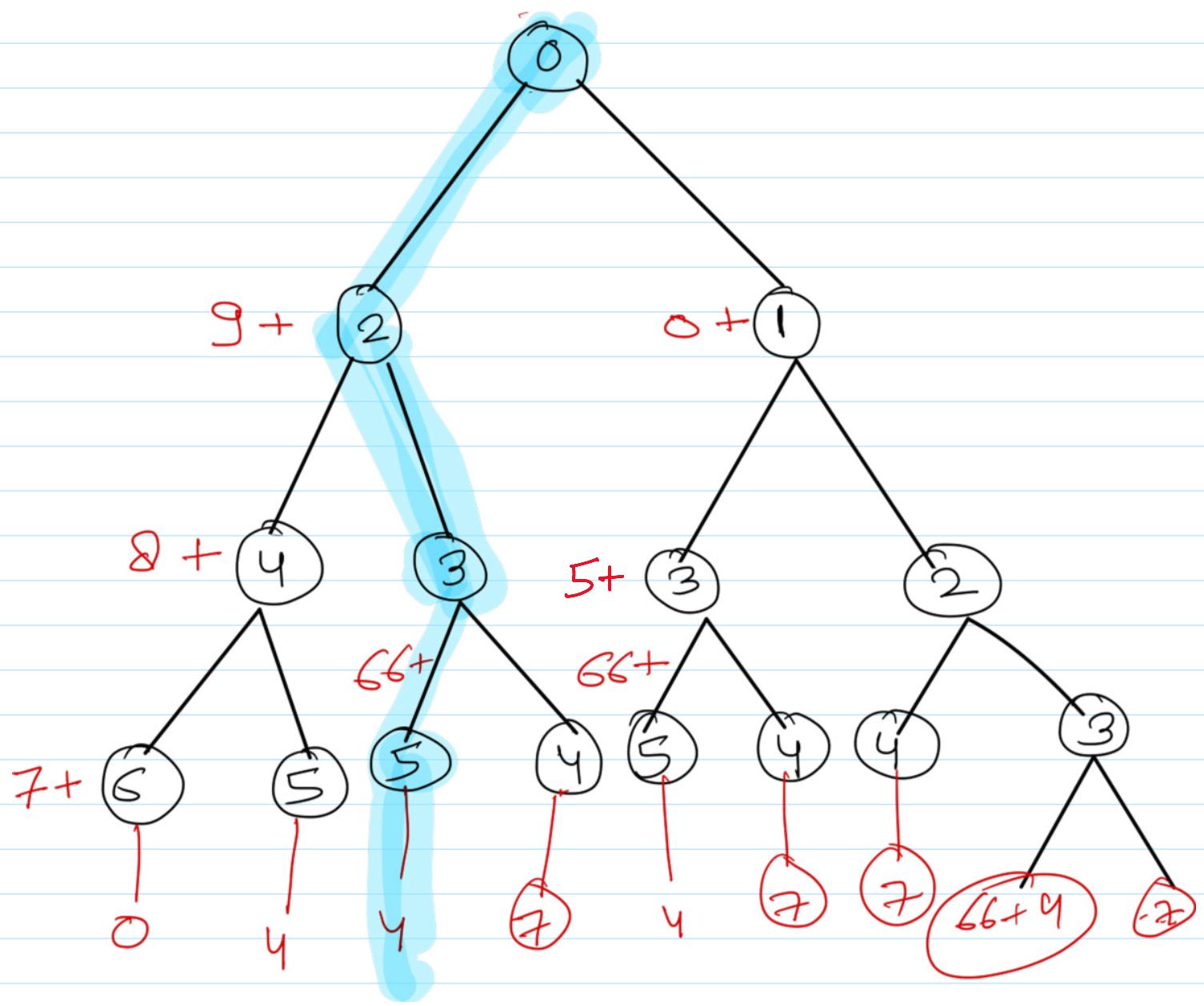
$$\rightarrow 7 + 3 = 10$$

$$\rightarrow 7 + 1 = 8$$

$$\rightarrow 9 + 1 = 10$$

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.) $\text{nums} = \underline{\underline{9}} \ \underline{5} \ \underline{8} \ \underline{\underline{66}} \ \underline{7} \ \underline{\underline{4}}$



GFG:- Recursion + Memoization.

```
● ● ●  
1 int rob(vector<int>& nums) {  
2     int n=nums.size();  
3     vector<int> v(n,-1);  
4     return nonadjacent(0,n-1,v,nums);  
5 }  
6  
7     int nonadjacent(int currentindex, int target, vector<int> &v,  
8     vector<int>& nums)  
9     {  
10         if(currentindex==target)  
11             return nums[currentindex];  
12         if(currentindex>target)  
13             return 0;  
14         int currentkey = currentindex;  
15         if(v[currentkey]!=-1)  
16             return v[currentkey];  
17         int rob = nums[currentindex] +  
18             nonadjacent(currentindex+2,target, v,nums);  
19         int dontrob = nonadjacent(currentindex+1, target, v,nums);  
20         v[currentkey]= max(rob,dontrob);  
21         return v[currentkey];  
22     }  
23 }
```

$$TC = O(N)$$

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

TABULATION:

```

1 int rob(vector<int>& nums) {
2     int n=nums.size();
3     vector<int> dp(n+1,0);
4     dp[n-1]=nums[n-1];
5     dp[n] =0;
6     for(int currIndex=n-2;currIndex>=0;currIndex--)
7     {   int pick = nums[currIndex] + dp[currIndex+2];
8         int notpick = dp[currIndex+1];
9         dp[currIndex] = max(pick,notpick);
10    }
11    return dp[0];
12}

```

$$TC = O(N)$$

$$SC = O(N)$$

SPACE OPTIMIZATION IN TABULATION:

```

1 //TABULATION
2 int rob(vector<int>& nums) {
3     //space optimization in tabulation
4     int n=nums.size();
5     int next2 =0;
6     int next = nums[n-1];
7     for(int currentindex =n-2;currentindex>=0;currentindex--){
8         int rob = nums[currentindex] + next2;
9         int dontrob = next;
10        int curramount= max(rob,dontrob);
11        next2 = next;
12        next = curramount;
13    }
14    return next;
15}

```

$$TC = O(N)$$

$$SC = O(1)$$

BY DOING SAME AS DELETE AND EARN:

same as Delete & earn.

$$\left. \begin{array}{l} ni = arr[i] + excl; \\ ne = \max(incl, excl); \end{array} \right\}$$

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/kapilyadav22)



```
1 int findMaxSum(int *arr, int n) {  
2  
3     int incl=0;  
4     int excl=0;  
5  
6     for(int i=0;i<n;i++)  
7     { int ni = arr[i] + excl;  
8         int ne = max(incl,excl);  
9         incl = ni;  
10        excl = ne;  
11    }  
12    return max(incl,excl);  
13}
```

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

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:

```
● ● ●  
1 int rob(vector<int>& nums) {  
2     int n = nums.size();  
3     vector<int> dp(n,-1);  
4     if(n==1)  
5         return nums[0];  
6     int excludinglast = maxamount(0,n-2,nums,dp);  
7     fill(dp.begin(), dp.end(), -1);  
8     int excludingfirst = maxamount(1,n-1,nums,dp);  
9     return max(excludinglast,excludingfirst);  
10 }  
11 int maxamount(int CurrInd,int target,vector<int>& nums,vector<int>& dp)  
12 {  if(CurrInd==target)  
13     return nums[CurrInd];  
14     if(CurrInd>target)  
15         return 0;  
16     if(dp[CurrInd]!=-1)  
17         return dp[CurrInd];  
18  
19     int rob = nums[CurrInd] + maxamount(CurrInd+2,target,nums,dp);  
20     int notrob = maxamount(CurrInd+1,target,nums,dp);  
21     return dp[CurrInd] = max(rob,notrob);  
22 }
```

$$\begin{cases} TC = O(2N) \approx O(N) \\ SC = \underline{O(N) + O(N)} \end{cases}$$

TABULATION:



$TC = O(N)$

$SC = O(N)$

```
1 int rob(vector<int>& nums) {
2     int n = nums.size();
3     vector<int> dp1(n+1, 0);
4     vector<int> dp2(n+1, 0);
5     if(n==1)
6         return nums[0];
7     dp1[n-1]=nums[n-1];
8     dp2[0]=nums[0];
9
10    for(int CurrInd=n-2;CurrInd>=1;CurrInd--)
11    {   int rob = nums[CurrInd] + dp1[CurrInd+2];
12        int notrob = dp1[CurrInd+1];
13        dp1[CurrInd] = max(rob,notrob);
14    }
15    for(int CurrInd=n-2;CurrInd>=0;CurrInd--)
16    {   int rob = nums[CurrInd] + dp2[CurrInd+2];
17        int notrob = dp2[CurrInd+1];
18        dp2[CurrInd] = max(rob,notrob);
19    }
20    int excludinglast = dp1[1];
21    int excludingfirst =dp2[0];
22    return max(excludinglast,excludingfirst);
23 }
```

SPACE OPTIMIZATION:

```
● ● ●
```

```
1 int houserobber1(int currindex,int target,vector<int>& nums) {
2     //space optimization in tabulation
3     int next2 =0;
4     int next = nums[currindex];
5     for(int currentindex = currindex-1; currentindex>=target;currentindex--){
6         int rob = nums[currentindex] + next2;
7         int dontrob = next;
8         int curramount= max(rob,dontrob);
9         next2 = next;
10        next = curramount;
11    }
12    return next;
13 }
14
15 int rob(vector<int>& nums) {
16     //it is most optimized
17     int n=nums.size();
18     if (n == 1) return nums[0];
19     return max(houserobber1(n-1,1,nums),houserobber1(n-2,0,nums));
20 }
```

$TC = O(2N) \leq O(N)$
 $SC = O(1)$

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/kapilyadav22)

*LECTURE-7 NINJA'S Training

Friday, 17 June 2022 8:28 PM

Ninja's Training

Difficulty: MEDIUM

Contributed By Srejan Kumar Bera | Level 1

Avg. time to solve 30 min

Success Rate 50%

Problem Statement

Suggest Edit

Ninja is planning 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 $N \times 3$ '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 $\begin{bmatrix} [1, 2, 5], [3, 1, 1], [3, 3, 3] \end{bmatrix}$, 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.

$$\text{so } \text{maxprofit} = 50 + 11 = 61.$$

→ But, if i choose $10 + 100$

$$\text{maxprofit} = 110$$

DP → Memoization.

~~f(day, lasttask)~~

{ 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[day][i]);
        return maxi;
    }

```

```

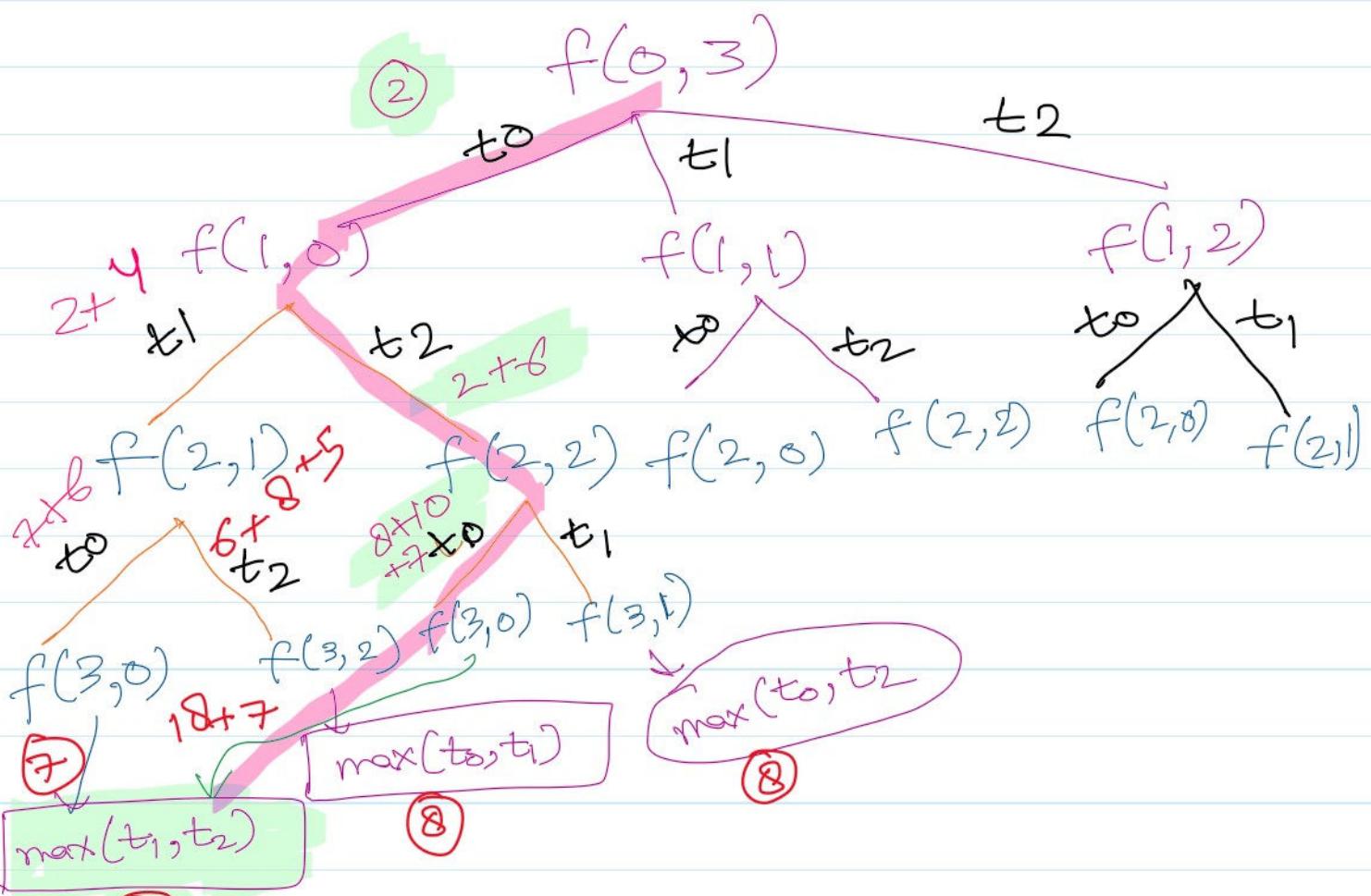
    int maxi = 0;
    for(int i=0; i<3; i++)
    {
        if(i != lasttask)
            points = task[day][i] + f(day+1, i);
        maxi = max(maxi, points);
    }
    return maxi;
}

```

$\exists \underline{n=4}$

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

$\begin{cases} 0 - \text{task1} \\ 1 - \text{task2} \\ 2 - \text{task3} \\ 3 - \text{No task selected} \end{cases}$



$\max(t_1, t_2)$

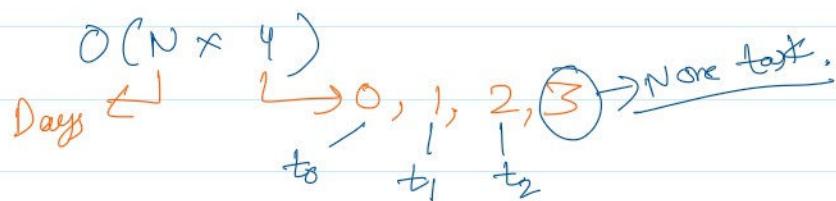
(8)

(7)

$$\rightarrow t_0 \rightarrow t_1 \rightarrow t_3 \rightarrow t_2 = 2 + 4 + 10 + 7 \\ = 17 + 6 = 23$$

$$\rightarrow t_0 \rightarrow t_2 \rightarrow t_0 \rightarrow t_2 = 2 + 6 + 10 + 7 \\ = 17 + 8 = 25$$

$$[TC = O(N \times 4) \times 3] \\ [SC = O(N) + O(N \times 4)]$$



Memoization:



```

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

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

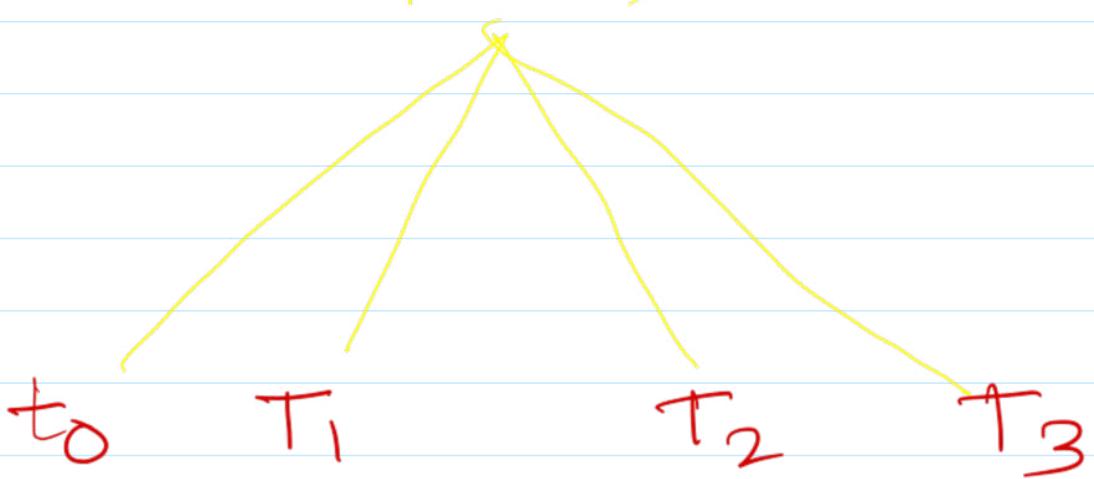
$$SC = O(N \times 4)$$

TC explained?

	t_0	T_1	T_2	T_3
d_0	d_{0t_0} d_{1t_1} d_{2t_2}	d_{0t_0} d_{1t_1} d_{2t_2}	d_{0t_0} d_{1t_1} d_{2t_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 $TC = O(N \times 4) \times 3$.

$$f(0, 3)$$



d_0	$\frac{t_0 +}{d_1 t_1}$ $\frac{}{d_1 t_2}$	$\frac{d_1 t_0}{t_1 +}$ $\frac{}{d_1 t_2}$	$\frac{d_1 t_0}{d_1 t_1}$ $\frac{t_2 +}{d_2 t_2}$	Final answer
d_1	$\frac{t_0 +}{d_2 t_1}$ $\frac{}{d_2 t_2}$	$\frac{d_2 t_0}{t_1 +}$ $\frac{}{d_2 t_2}$	$\frac{d_2 t_0}{d_2 t_1}$ $\frac{}{t_2 +}$	≈ 1
d_2	$\frac{t_0 +}{d_3 t_1}$ $\frac{}{d_3 t_2}$	$\frac{d_3 t_0}{t_1 +}$ $\frac{}{d_3 t_2}$	$\frac{d_3 t_0}{d_3 t_1}$ $\frac{}{t_2 +}$	-1
d_3	Some value	Some value	Some value	-1



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

Space Optimization:



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

	T_0	T_1	T_2	T_3
d_0	$\max(\text{arr}[d_0][T_0] + \text{dp}[d_1][T_1], \text{arr}[d_0][T_2] + \text{dp}[d_1][T_2])$	$\max(\text{arr}[d_0][T_0] + \text{dp}[d_1][T_0], \text{arr}[d_0][T_2] + \text{dp}[d_1][T_2])$	$\max(\text{arr}[d_0][T_0] + \text{dp}[d_1][T_0], \text{arr}[d_0][T_1] + \text{dp}[d_1][T_1])$	$\max(d_0 t_0, d_0 t_1, d_0 t_2)$
d_1	$\max(\text{arr}[d_1][T_0] + \text{dp}[d_2][T_1], \text{arr}[d_1][T_2] + \text{dp}[d_2][T_2])$	$\max(\text{arr}[d_1][T_0] + \text{dp}[d_2][T_0], \text{arr}[d_1][T_2] + \text{dp}[d_2][T_2])$	$\max(\text{arr}[d_1][T_0] + \text{dp}[d_2][T_0], \text{arr}[d_1][T_1] + \text{dp}[d_2][T_1])$	$\max(d_1 t_0, d_1 t_1, d_1 t_2)$
d_2	$\max(\text{arr}[d_2][t_0] + \text{dp}[d_3][t_1], \text{arr}[d_2][t_2] + \text{dp}[d_3][t_2])$	$\max(\text{arr}[d_2][t_0] + \text{dp}[d_3][t_0], \text{arr}[d_2][t_2] + \text{dp}[d_3][t_2])$	$\max(\text{arr}[d_2][t_0] + \text{dp}[d_3][t_0], \text{arr}[d_2][t_1] + \text{dp}[d_3][t_1])$	$\max(d_2 t_0, d_2 t_1, d_2 t_2)$
d_3	$\max(t_1, t_2)$	$\max(t_0, t_2)$	$\max(t_0, t_1)$	$\max(t_0, t_1, t_2)$

$n = 4$

T_0	T_1	T_2	<u>t_{avg}</u>
2	1	3	d_0
3	4	6	d_1
10	1	6	d_2
8	3	7	d_3

	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

*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 $m \times n$ grid. The robot is initially located at the **top-left corner** (i.e., $\text{grid}[0][0]$). The robot tries to move to the **bottom-right corner** (i.e., $\text{grid}[m - 1][n - 1]$). 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

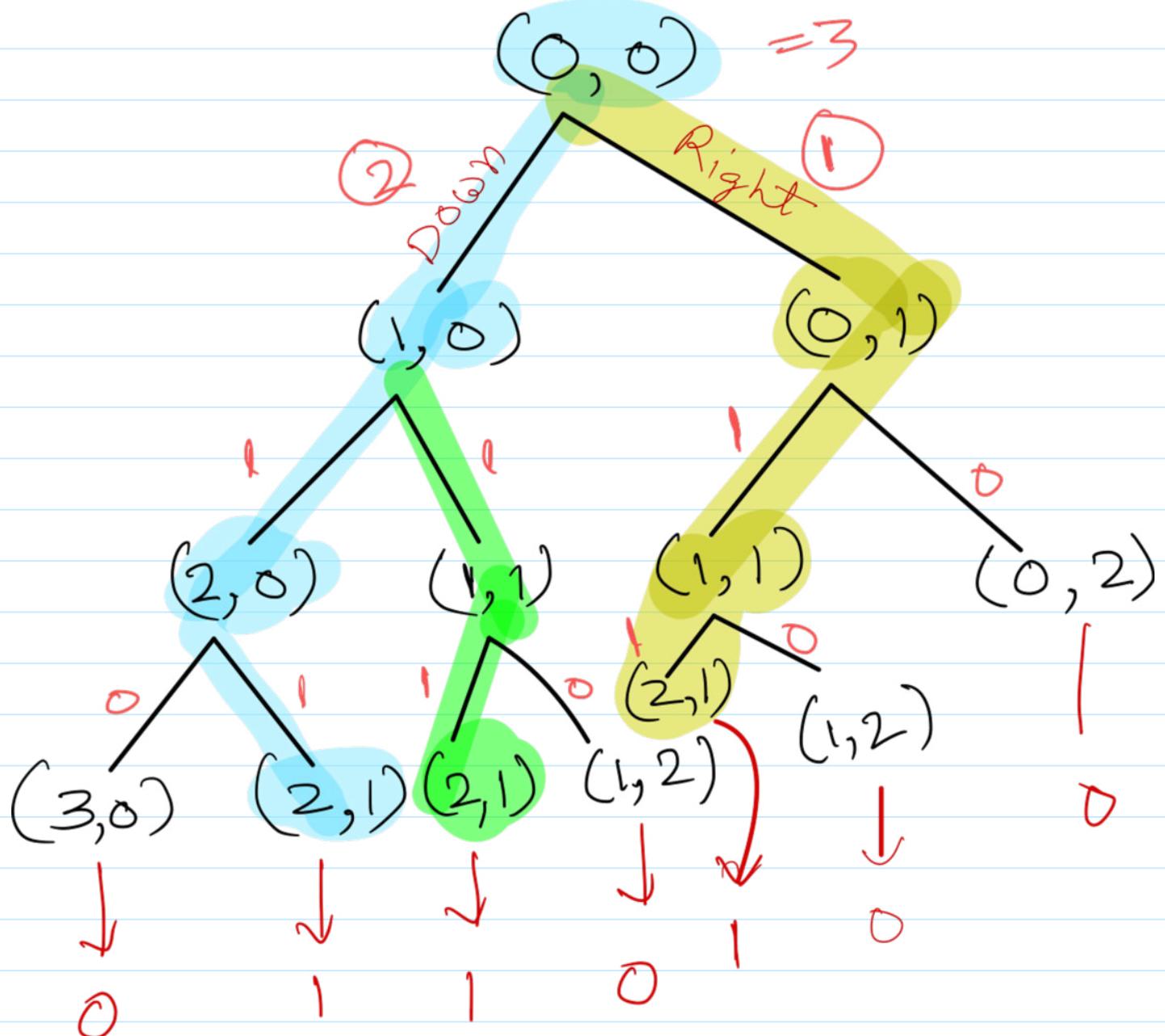
Example 2:

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:

```

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

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.



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

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

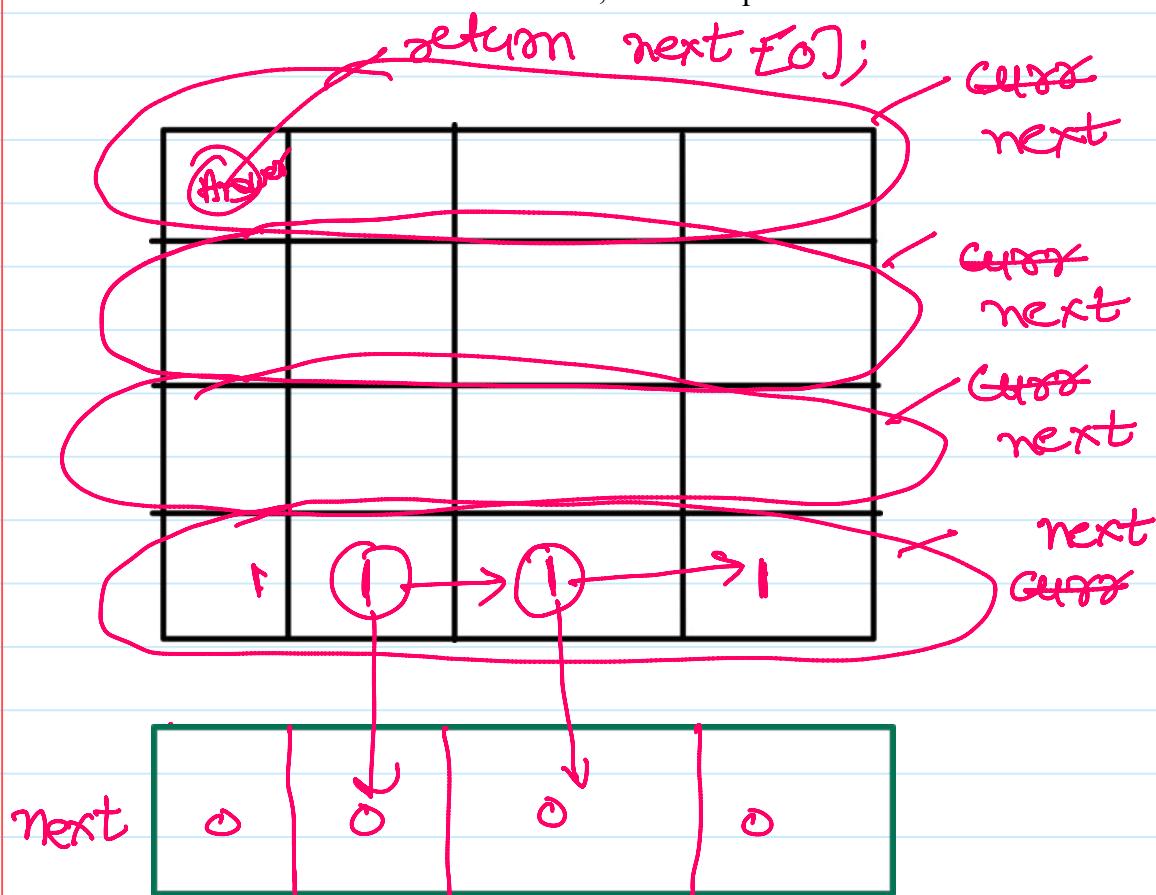
OR

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

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, sum right for $\text{currRow} < m-1$, & Down for $\text{currCol} < n-1$
and Down paths is in next vector.

Right path is in curr vector

so, $\text{curr}[\text{currCol}] = \text{Down} + \text{right}$.
 $\text{Down} = \text{next}[\text{currCol}]$
 $\text{Right} = \text{curr}[\text{currCol}+1]$

→ after a row ends, store curr in next.

$\text{next} = \text{curr}$.

→ 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



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

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)



```
1 int uniquePaths(int m, int n) {  
2     //it will done using combinatorics  
3     // ncr, where n is the number of steps require, r is the  
4     // arrangement of these steps.  
5     // TC = O(m-1) or (n-1)  
6     int Number_of_steps = n+m-2 ; // (m-1) + (n-1)  
7     int r = m-1; // or n-1;  
8     double totalpaths = 1;  
9     for(int i = 1;i<=r;i++)  
10    {  
11        totalpaths= totalpaths*(Number_of_steps-r+i)/i;  
12    }  
13    return (int) totalpaths;  
14}
```

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

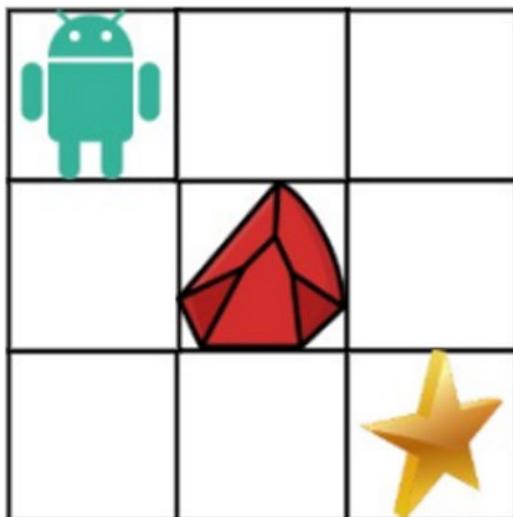
You are given an $m \times n$ integer array `grid`. There is a robot initially located at the **top-left corner** (i.e., `grid[0][0]`). The robot tries to move to the **bottom-right corner** (i.e., `grid[m-1][n-1]`). The robot can only move either down or right at any point in time.

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]]`

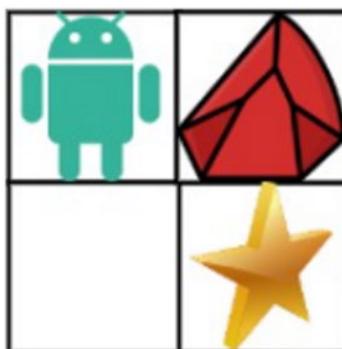
`Output: 2`

`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

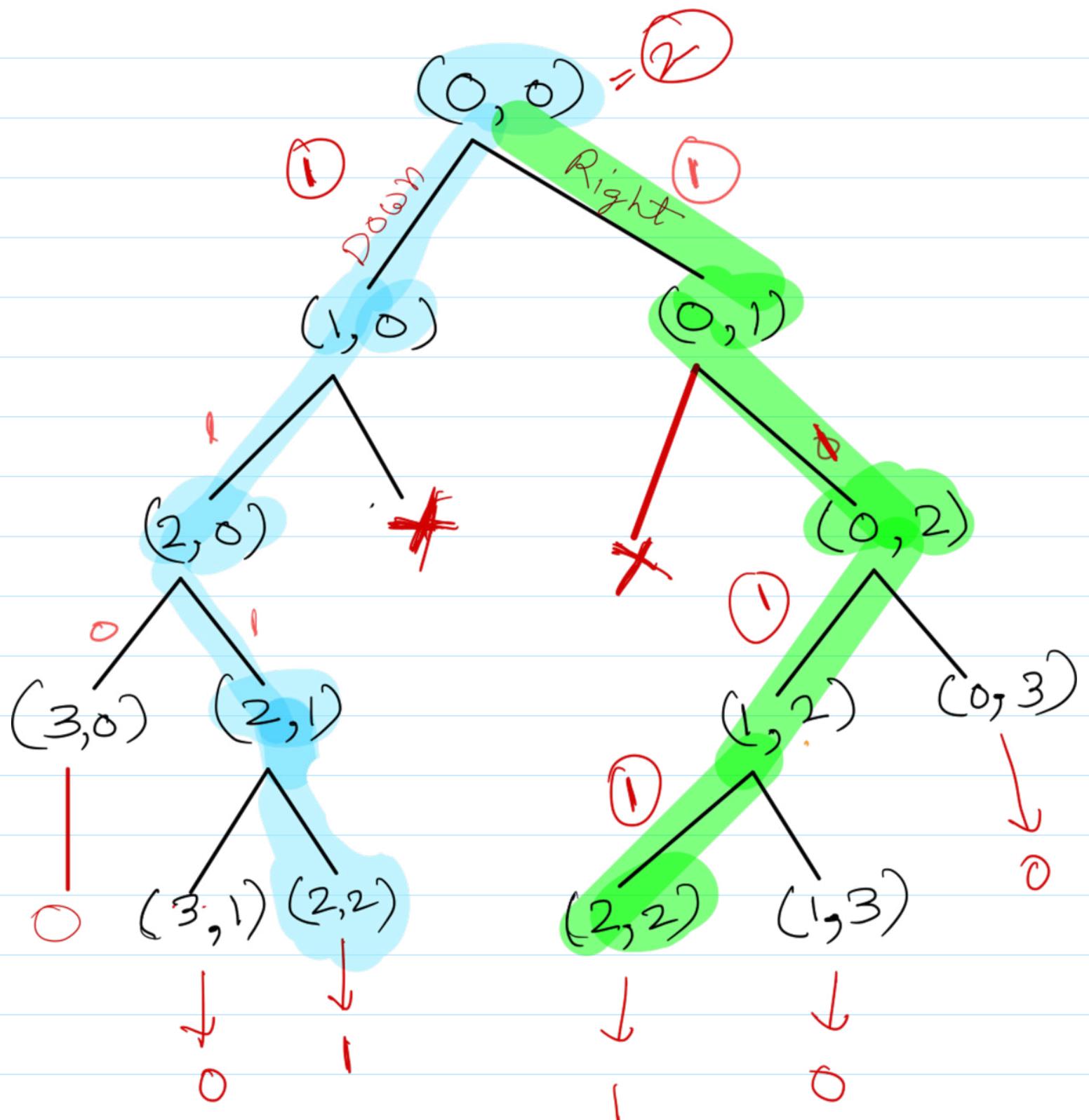
Example 2:



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

`Output: 1`

$\Leftrightarrow [[0,0,0], [0,1,0], [0,0,0]]$



MEMOIZATION:

● ● ●

```
1 int totalpaths(int CurrRow, int CurrCol, vector<vector<int>>& obstacleGrid, vector<vector<int>>& dp)
2     { int targetRow = obstacleGrid.size()-1;
3         int targetCol = obstacleGrid[0].size()-1;
4
5         if(CurrRow == targetRow && CurrCol == targetCol && obstacleGrid[CurrRow][CurrCol]!=1)
6             return 1;
7         if(CurrRow> targetRow || CurrCol>targetCol ||obstacleGrid[CurrRow][CurrCol]==1)
8             return 0;
9
10        if(dp[CurrRow][CurrCol]!=-1)
11            return dp[CurrRow][CurrCol];
12        int moveright = totalpaths(CurrRow,CurrCol+1,obstacleGrid,dp);
13        int movedown = totalpaths(CurrRow+1,CurrCol,obstacleGrid,dp);
14        return dp[CurrRow][CurrCol] = moveright + movedown;
15    }
16
17
18    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
19        int m = obstacleGrid.size();
20        int n = obstacleGrid[0].size();
21        vector<vector<int>> dp(m, vector<int> (n,-1));
22        return totalpaths(0,0,obstacleGrid,dp);
23    }
```

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:

● ● ●

```
1 int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
2     int m = obstacleGrid.size();
3     int n = obstacleGrid[0].size();
4     vector<vector<double>> dp(m, vector<double> (n,0));
5
6     for(int CurrRow=m-1;CurrRow>=0;CurrRow--)
7         { for(int CurrCol=n-1;CurrCol>=0;CurrCol--)
8             { if(CurrRow ==m-1 && CurrCol==n-1 && obstacleGrid[CurrRow][CurrCol]==0)
9                 dp[CurrRow][CurrCol]=1;
10             else if(obstacleGrid[CurrRow][CurrCol]==1)
11                 dp[CurrRow][CurrCol]==0;
12             else { double movedown =0;
13                 double moveright =0;
14                 if(CurrRow<m-1)
15                     movedown = dp[CurrRow+1][CurrCol];
16                 if(CurrCol<n-1)
17                     moveright = dp[CurrRow][CurrCol+1];
18                 dp[CurrRow][CurrCol] = movedown+moveright;
19             }
20         }
21     }
22     return (int)dp[0][0];
23 }
```

TC: O(M*N)

SC : O(M*N)

SPACE OPTIMIZATION:

```
● ● ●
```

```
1 int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
2     int m = obstacleGrid.size();
3     int n = obstacleGrid[0].size();
4     vector<double> next(n, 0);
5
6     for(int CurrRow=m-1;CurrRow>=0;CurrRow--)
7         { vector<double> curr(n, 0);
8             for(int CurrCol=n-1;CurrCol>=0;CurrCol--)
9                 { if(CurrRow ==m-1 && CurrCol==n-1 && obstacleGrid[CurrRow]
10                   [CurrCol]==0)
11                     curr[CurrCol]=1;
12                 else if(obstacleGrid[CurrRow][CurrCol]==1)
13                     curr[CurrCol]=0;
14                 else { double movedown =0;
15                     double moveright =0;
16                     if(CurrRow<m-1)
17                         movedown = next[CurrCol];
18                     if(CurrCol<n-1)
19                         moveright = curr[CurrCol+1];
20                     curr[CurrCol] = movedown+moveright;
21                 }
22             next = curr;
23         }
24     return (int)next[0];
25 }
```

TC: O(M*N)

SC : O(N)

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/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 $m \times n$ 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.

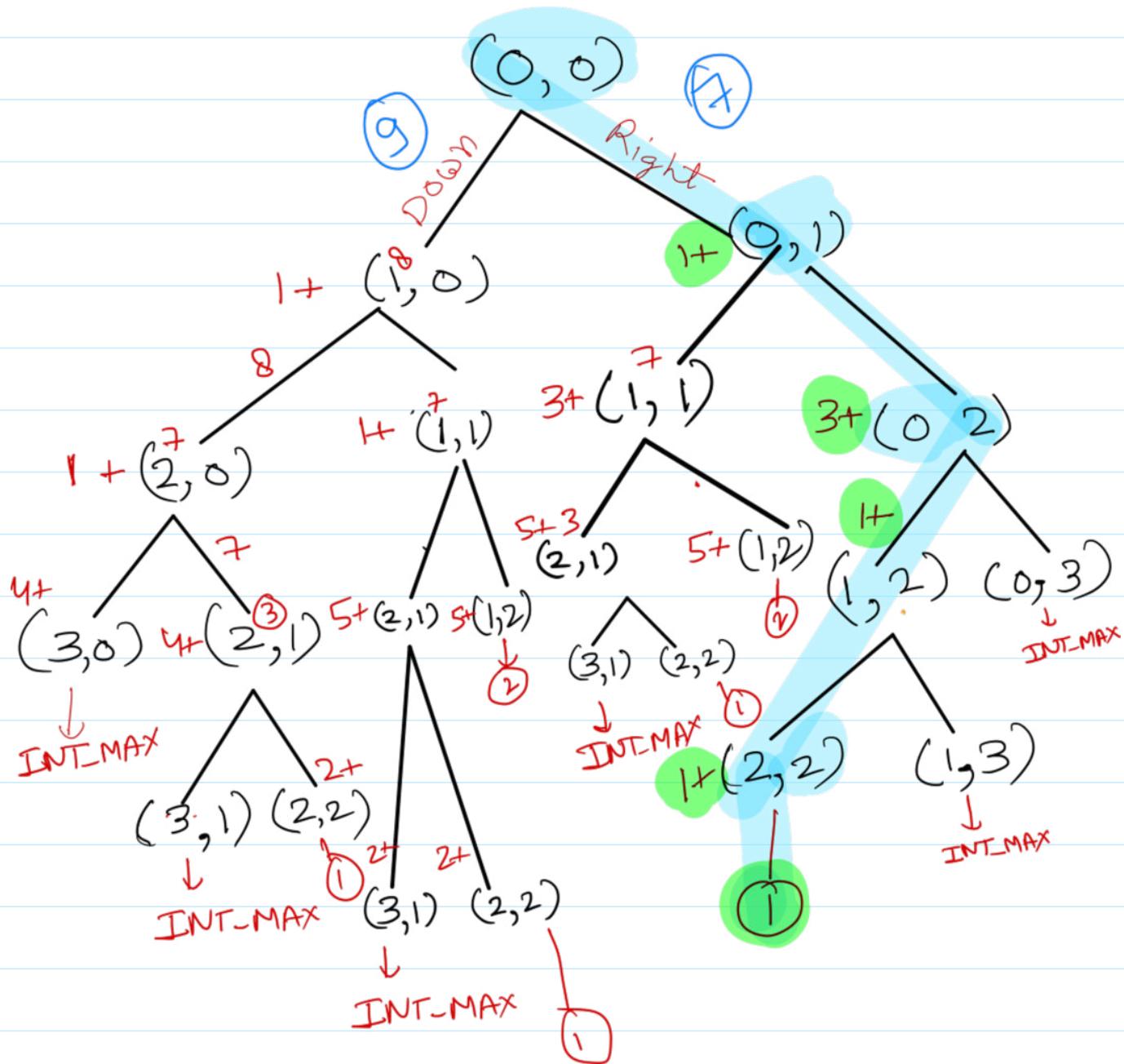
Note: You can only move either down or right at any point in time.

Example 1:

1	3	1
1	5	1
4	2	1

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

Output: 7



MEMOIZATION:

```

1 int findpath(int CurrRow, int CurrCol, int TargetRow, int TargetCol,
2     vector<vector<int>>& grid, vector<vector<int>> &dp)
3     { if(CurrRow==TargetRow && CurrCol == TargetCol)
4         return grid[CurrRow][CurrCol];
5     if(CurrRow>TargetRow || CurrCol>TargetCol)
6         return INT_MAX;
7     if(dp[CurrRow][CurrCol]!=-1)
8         return dp[CurrRow][CurrCol];
9     int movedown = findpath(CurrRow+1,CurrCol,TargetRow, TargetCol,grid,dp);
10    int moveright = findpath(CurrRow,CurrCol+1,TargetRow,TargetCol,grid,dp);
11    return dp[CurrRow][CurrCol] = grid[CurrRow][CurrCol] + min(movedown,moveright);
12 }
13
14 int minPathSum(vector<vector<int>>& grid) {
15     int rowsize = grid.size();
16     int colszie = grid[0].size();
17     vector<vector<int>> dp(rowsize,vector<int> (colszie,-1));
18     return findpath(0,0,rowsize-1,colszie-1,grid,dp);
19 }
20 }
```

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:

```
1 int minPathSum(vector<vector<int>>& grid) {  
2     int rowsize = grid.size();  
3     int colszie = grid[0].size();  
4     vector<vector<int>> dp(rowsize, vector<int> (colszie, INT_MAX));  
5     for(int CurrRow=rowsize-1; CurrRow>=0; CurrRow--)  
6     { for(int CurrCol = colszie-1; CurrCol>=0; CurrCol--)  
7         {  
8             if(CurrRow==rowsize-1 && CurrCol==colszie-1)  
9                 dp[CurrRow][CurrCol]=grid[CurrRow][CurrCol];  
10            else{  
11                int movedown=INT_MAX; int moveright=INT_MAX;  
12                if(CurrRow<rowsize-1)  
13                    movedown =  dp[CurrRow+1][CurrCol];  
14                if(CurrCol<colszie-1)  
15                    moveright =  dp[CurrRow][CurrCol+1];  
16                dp[CurrRow][CurrCol] =  grid[CurrRow][CurrCol] + min(movedown,moveright);  
17            }  
18        }  
19    }  
20    return dp[0][0];  
21 }
```

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

SPACE OPTIMIZATION:

```
1 int minPathSum(vector<vector<int>>& grid) {  
2     int rowsize = grid.size();  
3     int colszie = grid[0].size();  
4     vector<int> next(colszie, INT_MAX);  
5     for(int CurrRow=rowsize-1; CurrRow>=0; CurrRow--)  
6     { vector<int> curr(colszie, INT_MAX);  
7         for(int CurrCol = colszie-1; CurrCol>=0; CurrCol--)  
8             {  
9                 if(CurrRow==rowsize-1 && CurrCol==colszie-1)  
10                     curr[CurrCol]=grid[CurrRow][CurrCol];  
11                 else{  
12                     int movedown=INT_MAX; int moveright=INT_MAX;  
13                     if(CurrRow<rowsize-1)  
14                         movedown = next[CurrCol];  
15                     if(CurrCol<colszie-1)  
16                         moveright = curr[CurrCol+1];  
17                     curr[CurrCol] = grid[CurrRow][CurrCol] + min(movedown,moveright);  
18                 }  
19             }  
20             next = curr;  
21         }  
22     return next[0];  
23 }
```

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

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/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:

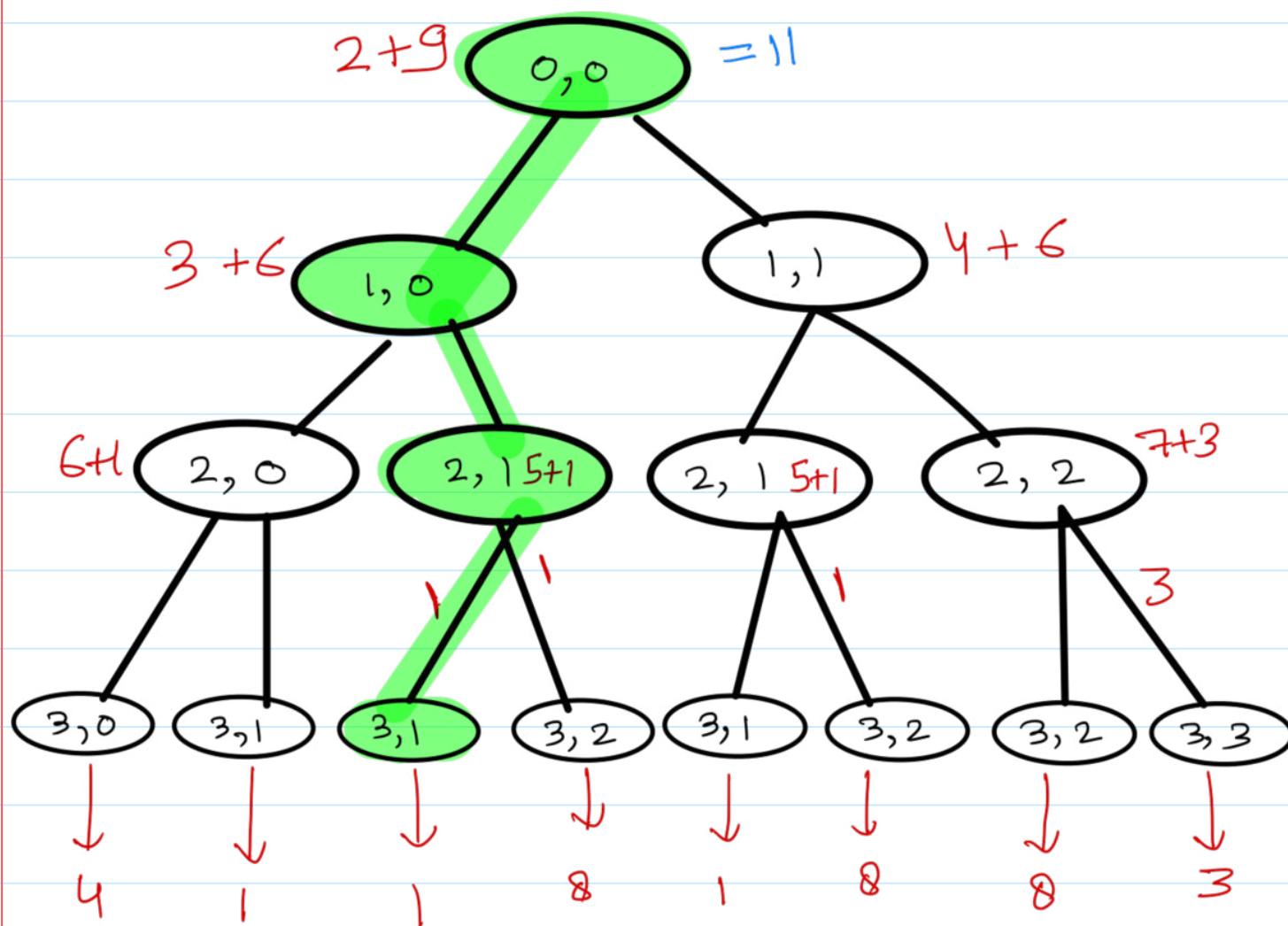
2
3 4
6 5 7
4 1 8 3

The minimum path sum from top to bottom is $2 + 3 + 5 + 1 = 11$ (underlined above).

RECURSIVE SOLUTION:

TC : $O(2^{1+2+3+4+\dots+N})$ where N is the number of rows

SC : $O(N)$



MEMOIZATION :

TC : N^N where N is the number of rows

SC : $(N^N) + (N)$

```
● ○ ●

1 int findminpath(int CurrRow, int CurrCol,vector<vector<int>>& triangle,
      vector<vector<int>> &dp)
2     {   int rowsize = triangle.size();
3         int colsize = triangle[CurrRow].size();
4         if(CurrRow == rowsize-1)
5             return triangle[CurrRow][CurrCol];
6
7         if(dp[CurrRow][CurrCol]!=-1)
8             return dp[CurrRow][CurrCol];
9
10        int onestep = findminpath(CurrRow+1,CurrCol,triangle,dp);
11        int twostep = findminpath(CurrRow+1, CurrCol+1,triangle,dp);
12        dp[CurrRow][CurrCol] = triangle[CurrRow][CurrCol] + min(onestep,twostep);
13
14        return dp[CurrRow][CurrCol];
15    }
16
17    int minimumTotal(vector<vector<int>>& triangle) {
18        int rowsize = triangle.size();
19        int colsize = triangle[rowsize-1].size();
20        vector<vector<int>> dp(rowsize,vector<int> (colsize,-1));
21        return findminpath(0,0,triangle,dp);
22    }

```

Tabulation:

```
● ○ ●

1 int minimumTotal(vector<vector<int>>& triangle) {
2     int rowsize = triangle.size();
3     int colsize = triangle[rowsize-1].size();
4
5     vector<vector<int>> dp(rowsize,vector<int> (colsize,0));
6     for(int i =0;i<colsize;i++)
7         dp[rowsize-1][i] = triangle[rowsize-1][i];
8
9     for(int CurrRow = rowsize-2;CurrRow>=0;CurrRow--)
10    { for(int CurrCol = CurrRow;CurrCol>=0;CurrCol--)
11        {
12            int onestep = dp[CurrRow+1][CurrCol];
13            int twostep = dp[CurrRow+1][CurrCol+1];
14            dp[CurrRow][CurrCol] = triangle[CurrRow][CurrCol] + min(onestep,twostep);
15        }
16    }
17    return dp[0][0];
18 }
```

TC : N^N where N is the number of rows

SC : (N^N)

SPACE OPTIMIZATION:

```
● ○ ●

1 int minimumTotal(vector<vector<int>>& triangle) {
2     int rowsize = triangle.size();
3     int colszie = triangle[rowsize-1].size();
4
5     vector<int> next(rowsize, 0);
6     vector<vector<int>> dp(rowsize, vector<int> (colszie, 0));
7     for(int i = 0; i < colszie; i++)
8         next[i] = triangle[rowsize-1][i];
9
10    for(int CurrRow = rowsize - 2; CurrRow >= 0; CurrRow--)
11    { vector<int> curr(rowsize, 0);
12        for(int CurrCol = CurrRow; CurrCol >= 0; CurrCol--)
13        {
14            int onestep = next[CurrCol];
15            int twostep = next[CurrCol + 1];
16            curr[CurrCol] = triangle[CurrRow][CurrCol] + min(onestep, twostep);
17        }
18        next = curr;
19    }
20    return next[0];
21 }
```

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 $n \times n$ 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:

2	1	3
6	5	4
7	8	9

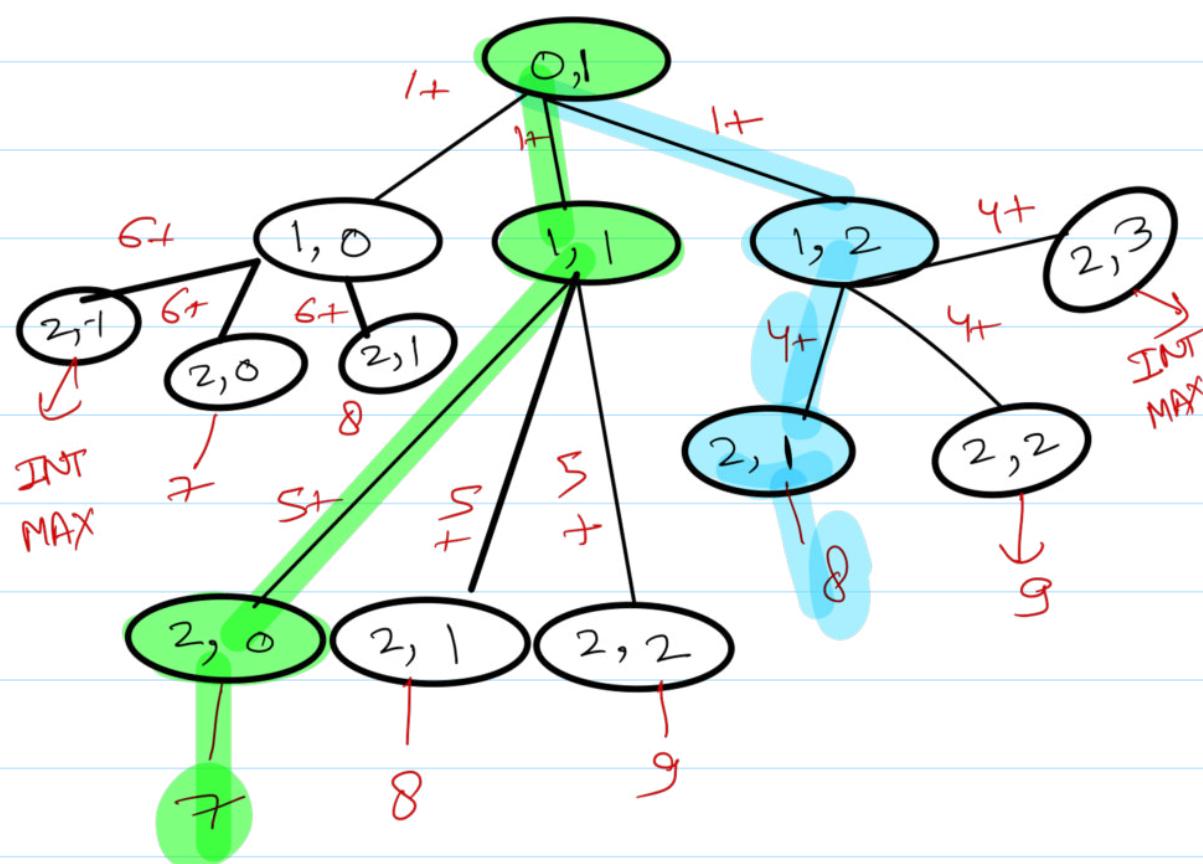
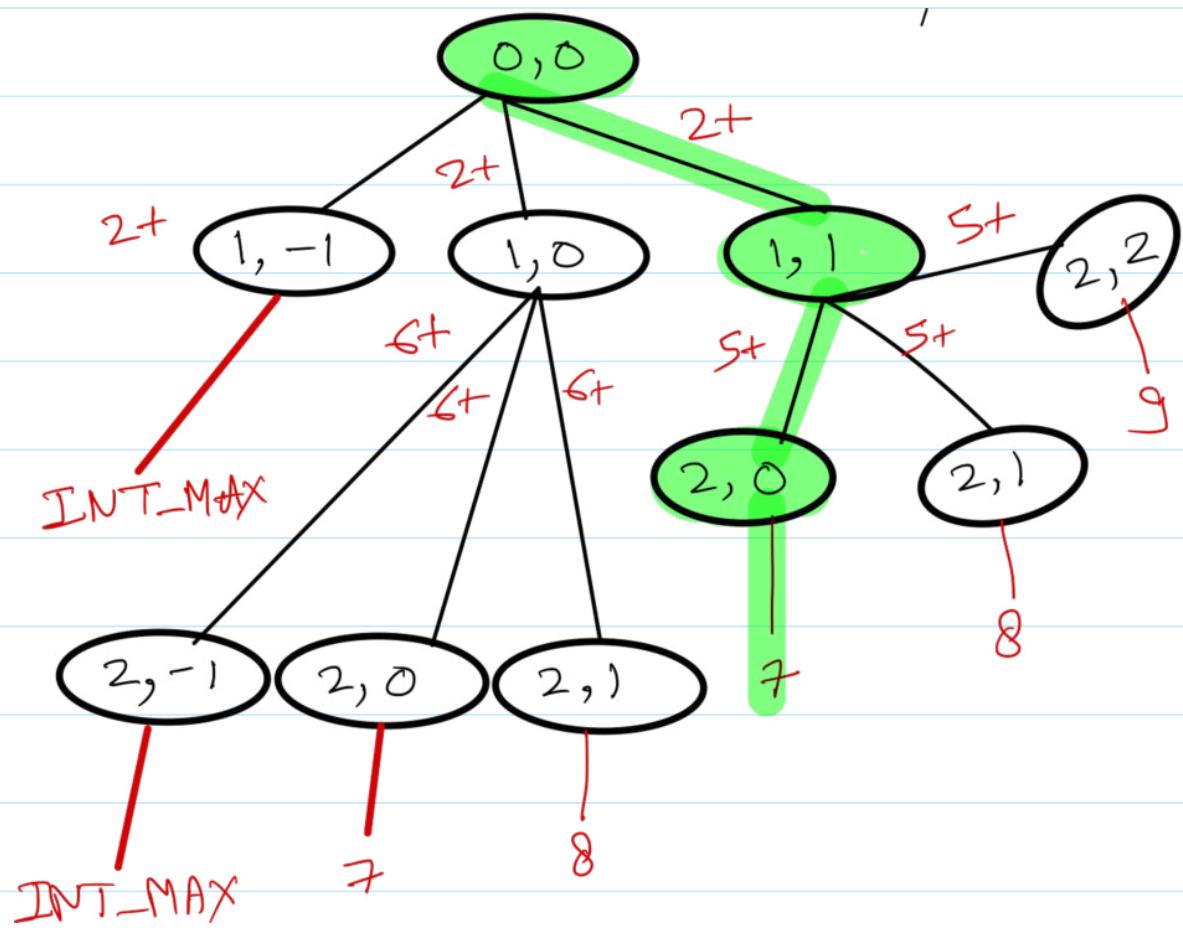
2	1	3
6	5	4
7	8	9

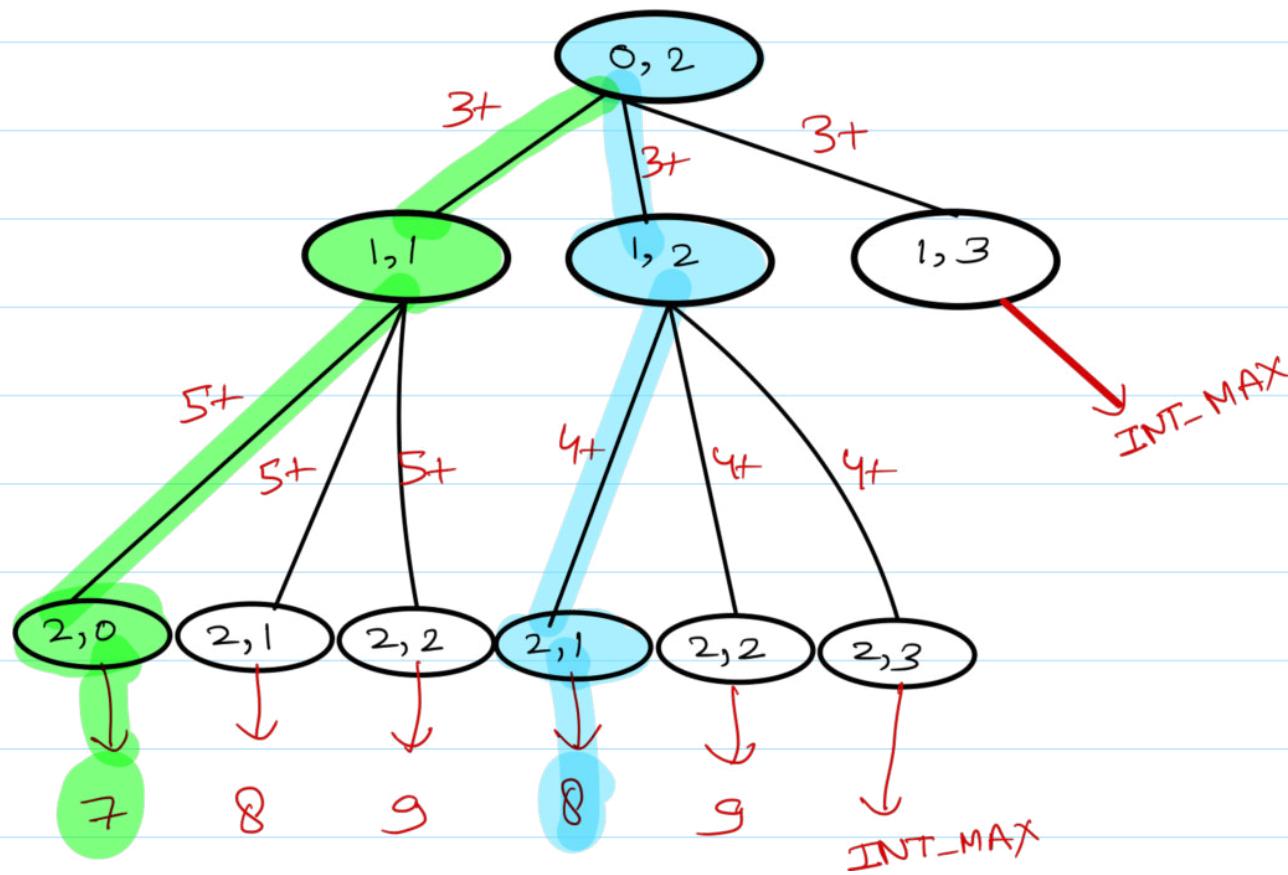
2	1	3
6	5	4
7	8	9

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.





RECURSION:

TC: 3^N Exponential

SC: $O(N)$

MEMOIZATION:

```

● ● ●
1 int findmin(int CurrRow, int CurrCol,vector<vector<int>>& matrix, vector<vector<int>> &dp)
2 {
3     int n = matrix[0].size();
4
5     if(CurrRow>n-1 || CurrRow<0 || CurrCol<0 || CurrCol>n-1)
6         return INT_MAX;
7
8     if(CurrRow==n-1)
9         return matrix[CurrRow][CurrCol];
10
11    if(dp[CurrRow][CurrCol]!=-1)
12        return dp[CurrRow][CurrCol];
13
14    int down = findmin(CurrRow+1,CurrCol,matrix,dp);
15    int downleft = findmin(CurrRow+1,CurrCol-1,matrix,dp);
16    int downright = findmin(CurrRow+1,CurrCol+1,matrix,dp);
17    return dp[CurrRow][CurrCol] = matrix[CurrRow][CurrCol] + min(down,min(downleft,downright));
18 }
19
20
21 int minFallingPathSum(vector<vector<int>>& matrix) {
22     int n = matrix.size();
23     vector<vector<int>> dp(n,vector<int> (n,-1));
24     int mini = INT_MAX;
25     for(int i =0;i<n;i++)
26     { mini = min(mini, findmin(0,i,matrix,dp));
27     }
28     return mini;
29 }
```

TC: $O(N \cdot M)$

SC : $O(N \cdot M) + O(N)$

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/kapilyadav22)

TABULATION:

```
1 int minFallingPathSum(vector<vector<int>>& matrix) {
2     int n = matrix.size();
3     vector<vector<int>> dp(n, vector<int> (n, 0));
4
5
6     for(int i = 0; i < n; i++)
7     { dp[n-1][i] = matrix[n-1][i];
8     }
9
10    for(int CurrRow = n-2; CurrRow >= 0; CurrRow--)
11    { for(int CurrCol = n-1; CurrCol >= 0; CurrCol--)
12        { int down = INT_MAX;
13            int downleft = INT_MAX;
14            int downright = INT_MAX;
15            down = dp[CurrRow+1][CurrCol];
16            if(CurrCol > 0)
17                downleft = dp[CurrRow+1][CurrCol-1];
18            if(CurrCol < n-1) downright = dp[CurrRow+1][CurrCol+1];
19            dp[CurrRow][CurrCol] = matrix[CurrRow][CurrCol] + min(down, min(downleft, downright));
20        }
21    }
22
23    int mini = INT_MAX;
24    for(int i = 0; i < n; i++)
25    { mini = min(mini, dp[0][i]);
26    }
27    return mini;
28 }
```

TC: $O(N \cdot M)$ + $O(M)$, where M is the number of columns

SC : $O(N \cdot M)$

SPACE OPTIMIZATION:

```
● ● ●

1 int minFallingPathSum(vector<vector<int>>& matrix) {
2     int n = matrix.size();
3     vector<int> next(n, 0);
4
5     for(int i = 0; i < n; i++) {
6         next[i] = matrix[n - 1][i];
7     }
8
9     for(int CurrRow = n - 2; CurrRow >= 0; CurrRow--) {
10    vector<int> curr(n, 0);
11    for(int CurrCol = n - 1; CurrCol >= 0; CurrCol--) {
12        int down = INT_MAX; int downleft = INT_MAX; int downright = INT_MAX;
13
14        down = next[CurrCol];
15        if(CurrCol > 0)
16            downleft = next[CurrCol - 1];
17        if(CurrCol < n - 1) downright = next[CurrCol + 1];
18        curr[CurrCol] = matrix[CurrRow][CurrCol] + min(down, min(downleft, downright));
19    }
20    next = curr;
21 }
22
23 int mini = INT_MAX;
24 for(int i = 0; i < n; i++) {
25     mini = min(mini, next[i]);
26 }
27 return mini;
28 }
```

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

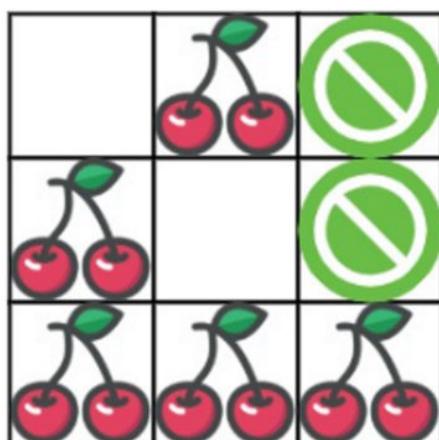
You are given an $n \times n$ grid representing a field of cherries, each cell is one of three possible integers.

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

0	1	-1
1	0	-1
1	1	1

→ 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:

```
● ● ●

1 //RECURSIVE SOLUTION
2 int cherryPickup(vector<vector<int>>& grid) {
3     return max(0, maxcherry(0, 0, 0, 0, grid));
4 }
5
6 int maxcherry(int r1, int c1, int r2, int c2, vector<vector<int>>& grid)
7 {
8     int cherries = 0;
9     int n = grid.size();
10    if( r1>=n || r2>=n || c1 >=n || c2 >=n || grid[r1][c1]==-1 || grid[r2][c2] == -1)
11        return -100;
12
13    if((r1==n-1) && (c1==grid[0].size()-1))
14        return grid[r1][c1];
15
16
17    if(r1==r2 || c1==c2)
18        cherries+=grid[r2][c2];
19    else
20        cherries+= grid[r1][c1]+grid[r2][c2];
21
22
23    int rr = maxcherry(r1,c1+1,r2,c2+1,grid);
24    int rd = maxcherry(r1,c1+1,r2+1,c2,grid);
25    int dd = maxcherry(r1+1,c1,r2+1,c2,grid);
26    int dr = maxcherry(r1+1,c1,r2,c2+1,grid);
27
28    cherries+= max({rr,rd,dd,dr});
29    return cherries;
30 }
```

MEMOIZATION:

```
● ● ●
```

```
1 int cherryPickup(vector<vector<int>>& grid) {
2     int dp[50][50][50][50];
3     memset(dp,-1,sizeof(dp));
4
5     return max(0, maxcherry(0,0,0,0, grid,dp));
6 }
7
8 int maxcherry(int r1, int c1, int r2,int c2, vector<vector<int>>& grid, int dp[50]
[50][50][50])
9 {
10    int cherries = 0;
11    //int c2=r1+c1-r2;
12    int n = grid.size();
13    if( r1>=n || r2>=n || c1 >=n || c2 >=n || grid[r1][c1]==-1 || grid[r2][c2] == -1)
14        return -10000;
15
16    if(dp[r1][c1][r2][c2]!=-1){
17        return dp[r1][c1][r2][c2];
18    }
19
20    if((r1==n-1) && (c1==grid[0].size()-1))
21        return grid[r1][c1];
22
23
24    if(r1==r2 || c1==c2)
25        cherries+=grid[r1][c1];
26    else
27        cherries+= grid[r1][c1]+grid[r2][c2];
28
29
30    int rr = maxcherry(r1,c1+1,r2,c2+1,grid,dp);
31    int rd = maxcherry(r1,c1+1,r2+1,c2,grid,dp);
32    int dd = maxcherry(r1+1,c1,r2+1,c2,grid,dp);
33    int dr = maxcherry(r1+1,c1,r2,c2+1,grid,dp);
34
35    cherries+= max({rr,rd,dd,dr});
36    dp[r1][c1][r2][c2]=cherries;
37    return cherries;
38 }
```

Further Optimization:

$$\begin{matrix} x_1, c_1 \\ 0, 0 \end{matrix}$$

$$\begin{matrix} x_2, c_2 \\ 0, 0 \end{matrix}$$

Since both are taking equal steps, so

$$x_1 + c_1 = x_2 + c_2$$

OK, Let's understand more.

- Let's say one person move right, $c_1 = c_1 + 1$ and another person move down, $x_2 = x_2 + 1$ so

$$x_1 + \underline{c_1 + 1} = \underline{x_2 + 1} + c_2$$

$$x_1 + c_1 + 1 = x_2 + c_2 + 1$$

$$x_1 + c_1 = x_2 + c_2$$

$$c_2 = x_1 + c_1 - x_2$$



```
1 int cherryPickup(vector<vector<int>>& grid) {
2     int dp[50][50][50];
3     memset(dp,-1,sizeof(dp));
4
5     return max(0, maxcherry(0, 0, 0, grid,dp));
6 }
7
8 int maxcherry(int r1, int c1, int r2, vector<vector<int>>& grid, int dp[50][50][50])
9 {
10    int cherries = 0;
11    int c2=r1+c1-r2;
12    int n = grid.size();
13    if( r1>=n || r2>=n || c1 >=n || c2 >=n || grid[r1][c1]==-1 || grid[r2][c2] == -1)
14        return -100000;
15
16    if(dp[r1][c1][r2]!=-1){
17        return dp[r1][c1][r2];
18    }
19
20    if((r1==n-1) && (c1==grid[0].size()-1))
21        return grid[r1][c1];
22
23
24    if(r1==r2 || c1==c2)
25        cherries+=grid[r1][c1];
26    else
27        cherries+= grid[r1][c1]+grid[r2][c2];
28
29
30    int rr = maxcherry(r1,c1+1,r2,grid,dp);
31    int rd = maxcherry(r1,c1+1,r2+1,grid,dp);
32    int dd = maxcherry(r1+1,c1,r2+1,grid,dp);
33    int dr = maxcherry(r1+1,c1,r2,grid,dp);
34
35    cherries+= max({rr,rd,dd,dr});
36    dp[r1][c1][r2]=cherries;
37    return cherries;
38 }
```

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/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

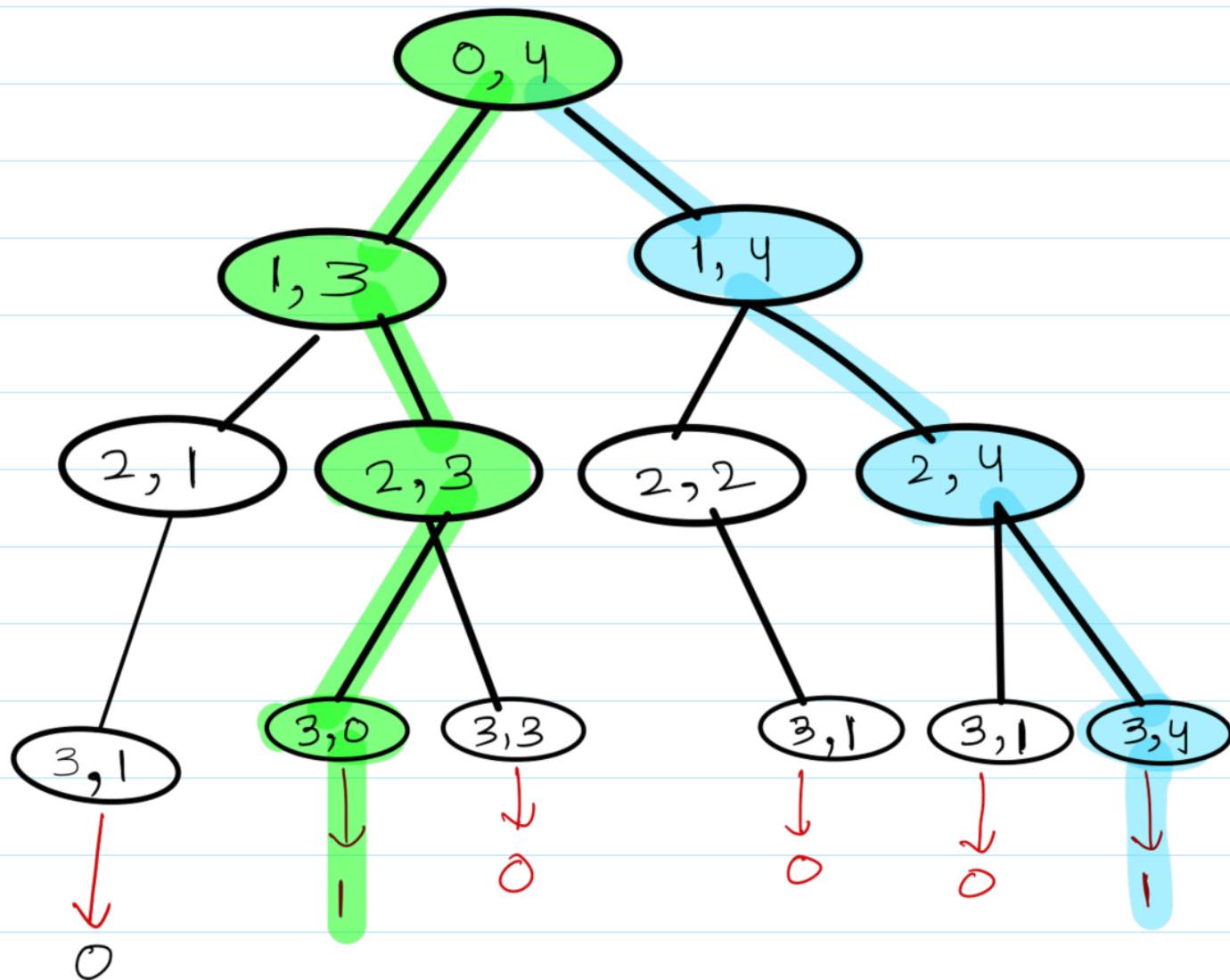
Suggest Edit

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:

```
● ● ●
```

```
1 bool findpartition(int currind, vector<int>& nums, vector<vector<int>> & dp, int sum)
2 {
3     if(sum==0)
4         return 1;
5     if(currind>=nums.size())
6         return 0;
7     if(dp[currind][sum]!=-1)
8         return dp[currind][sum];
9     bool consider = false;
10    if(sum>=nums[currind])
11        consider = findpartition(currind+1,nums,dp,sum-nums[currind]);
12    bool notconsider = findpartition(currind+1,nums,dp,sum);
13    return dp[currind][sum] = consider + notconsider;
14 }
15
16 bool subsetSumToK(int n, int k, vector<int> &nums) {
17     vector<vector<int>> dp(n, vector<int> (k+1,-1));
18     return findpartition(0,nums,dp,k);
19 }
20
```

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:

```
● ● ●
```

```
1 bool subsetSumToK(int n, int k, vector<int> &nums) {
2     vector<vector<bool>> dp(n, vector<bool> (k+1,false));
3     for(int i=0;i<n;i++)
4     {   dp[i][0] = true; }
5     if(nums[n-1]<=k)
6         dp[n-1][nums[n-1]] = true;
7     for(int currind=n-2;currind>=0;currind--)
8     {   for(int target=k;target>=1;target--)
9         {   bool consider = false;
10             if(target>=nums[currind])
11                 consider = dp[currind+1][target-nums[currind]];
12             bool notconsider = dp[currind+1][target];
13             dp[currind][target] = consider + notconsider;
14         }
15     }
16     return dp[0][k];
17 }
```

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 $dp[n-1][nums[n-1]] = \text{true}$, ($dp[n-1][nums[n-1]]$ 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 $arr[n-1] > \text{target}$, so we first check it: if($arr[n-1] \leq \text{target}$) then set $dp[n-1][nums[n-1]] = \text{true}$.

SPACE OPTIMIZATION:

```
1 bool subsetSumToK(int n, int k, vector<int> &nums) {
2     vector<bool> next(k+1, 0);
3     next[0]=1;
4     if(nums[n-1]<=k)
5         next[nums[n-1]] = true;
6     for(int currind=n-2;currind>=0;currind--)
7     {   vector<bool> curr(k+1, 0);
8         for(int target=k;target>=0;target--)
9         {   bool consider = false;
10             if(target>=nums[currind])
11                 consider = next[target-nums[currind]];
12             bool notconsider = next[target];
13             curr[target] = consider + notconsider;
14         }
15         next = curr;
16     }
17     return next[k];
18 }
```

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 can run our target loop till $\text{target} \geq 1$. and put $\text{curr}[0]=1$.

```
1 bool subsetSumToK(int n, int k, vector<int> &nums) {  
2     vector<bool> next(k+1,0);  
3     next[0]=1;  
4     if(nums[n-1]<=k)  
5         next[nums[n-1]] = true;  
6     for(int currind=n-2;currind>=0;currind--)  
7     { vector<bool> curr(k+1,0);  
8         curr[0]=1;  
9         for(int target=k;target>=1;target--)  
10            { bool consider = false;  
11                if(target>=nums[currind])  
12                    consider = next[target-nums[currind]];  
13                bool notconsider = next[target];  
14                curr[target] = consider + notconsider;  
15            }  
16            next = curr;  
17        }  
18        return next[k];  
19 }
```

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/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]`

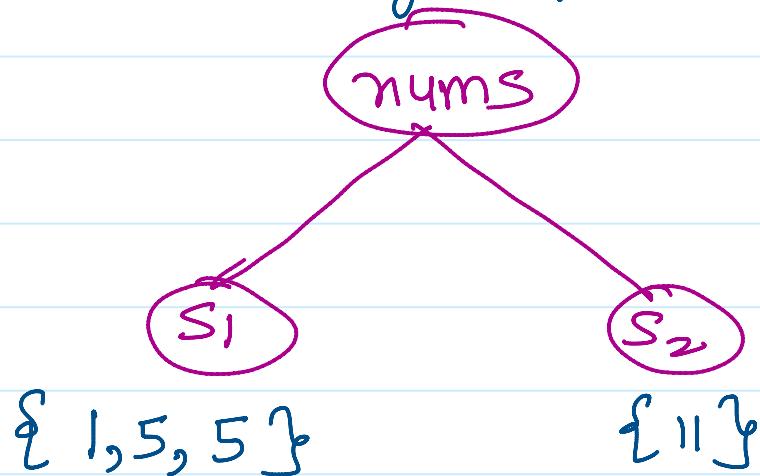
Output: false

Explanation: The array cannot be partitioned into equal sum subsets.

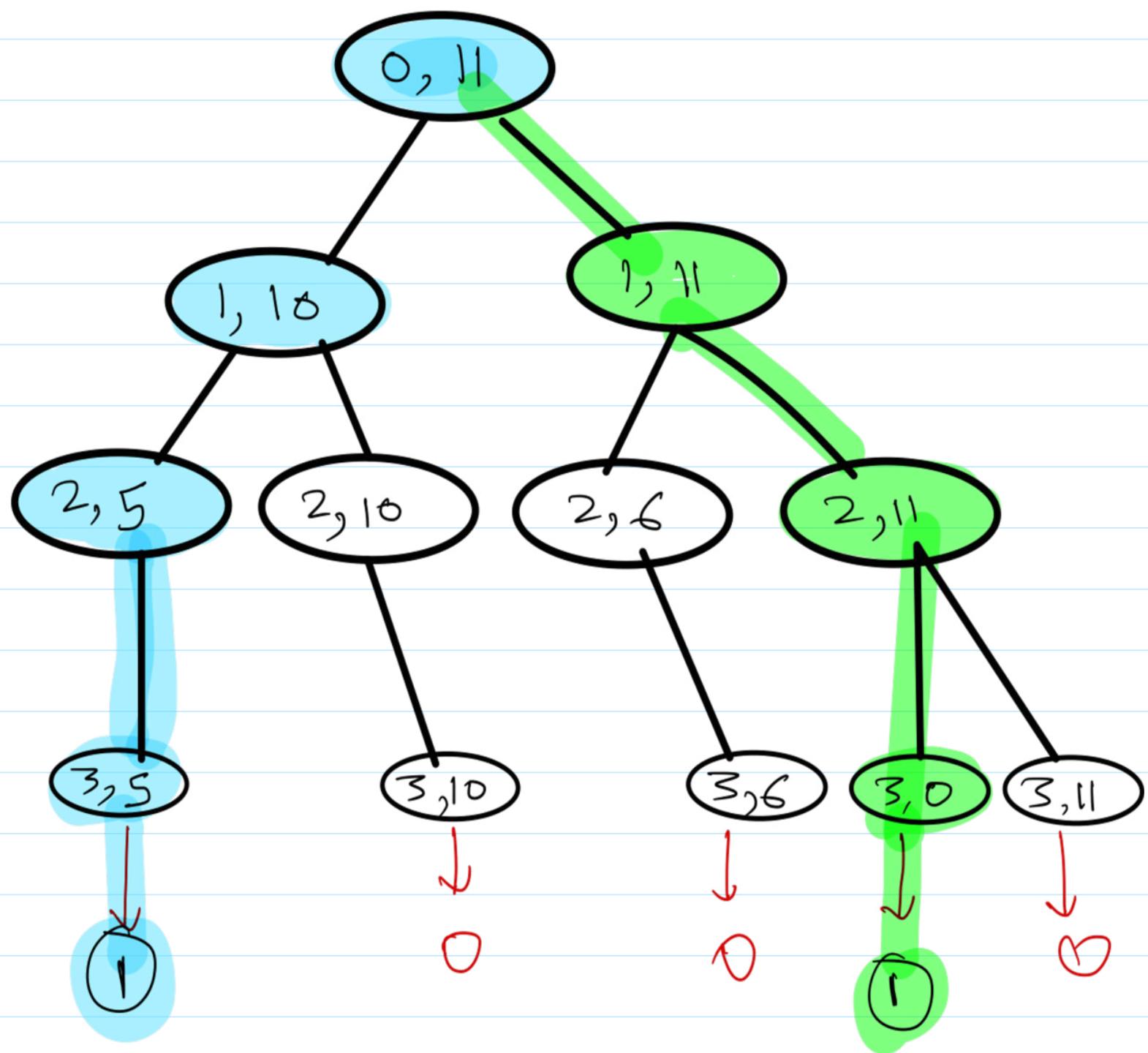
Same as Lecture 14. We just need to check if total sum is odd, we cannot partition it, return false. Else check for the sum of `totalsum/2`.

ex $\Rightarrow \{1, 5, 11, 5\}$

$\text{totalsum} = 22$, so, we can divide it into two subsets having equal sum.



\rightarrow So we need to find, whether there is a subset of $\text{sum} = \text{totalsum}/2$ or not.



MEMOIZATION:

```
● ● ●
```

```
1 bool findpartition(int currind, vector<int>& nums, vector<vector<int>>
& dp, int sum)
2 {
3     if(sum==0)
4         return 1;
5     if(currind>=nums.size())
6         return 0;
7     if(dp[currind][sum]!=-1)
8         return dp[currind][sum];
9     bool consider = false;
10    if(sum>=nums[currind])
11        consider = findpartition(currind+1,nums,dp,sum-nums[currind]);
12    bool notconsider = findpartition(currind+1,nums,dp,sum);
13    return dp[currind][sum] = consider + notconsider;
14 }
15
16 bool canPartition(vector<int> &nums, int n)
17 { int sum=0;
18     for(int i =0;i<n;i++)
19         sum+=nums[i];
20     if(sum%2!=0)
21         return false;
22     vector<vector<int>> dp(n,vector<int> (sum/2+1,-1));
23     return findpartition(0,nums,dp,sum/2);
24 }
```

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 $K = \text{sum}/2$

TABULATION:

```
1 bool canPartition(vector<int>& nums) {
2     int n = nums.size();
3     int sum=0;
4     for(int i =0;i<n;i++)
5         sum+=nums[i];
6     if(sum%2!=0)
7         return false;
8     vector<vector<bool>> dp(n,vector<bool> (sum/2+1,0));
9     for(int i=0;i<n;i++)
10    {      dp[i][0] = true;
11    }
12    if(nums[n-1]<=sum/2)
13        dp[n-1][nums[n-1]] = true;
14    for(int currind=n-2;currind>=0;currind--)
15    {  for(int target=sum/2;target>=0;target--)
16        { bool consider = false;
17            if(target>=nums[currind])
18                consider = dp[currind+1][target-nums[currind]];
19            bool notconsider = dp[currind+1][target];
20            dp[currind][target] = consider + notconsider;
21        }
22    }
23    return dp[0][sum/2];
24 }
```

TC: $O(N \cdot K)$

SC : $O(N \cdot k)$

Where $k = \text{sum}/2$

SPACE OPTIMIZATION:

```
● ● ●
```

```
1 bool canPartition(vector<int>& nums) {
2     int n = nums.size();
3     int sum=0;
4     for(int i =0;i<n;i++)
5         sum+=nums[i];
6     if(sum%2!=0)
7         return false;
8
9     vector<bool> next(sum/2+1,0);
10    next[0]=1;
11    if(nums[n-1]<=sum/2)
12        next[nums[n-1]] = true;
13    for(int currind=n-2;currind>=0;currind--)
14    {   vector<bool> curr(sum/2+1,0);
15        for(int target=sum/2;target>=0;target--)
16        {   bool consider = false;
17            if(target>=nums[currind])
18                consider = next[target-nums[currind]];
19            bool notconsider = next[target];
20            curr[target] = consider + notconsider;
21        }
22        next = curr;
23    }
24    return next[sum/2];
25 }
```

Time Complexity: $O(N*K)$

Space Complexity: $O(K)$

Where $k = \text{sum}/2$

[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.

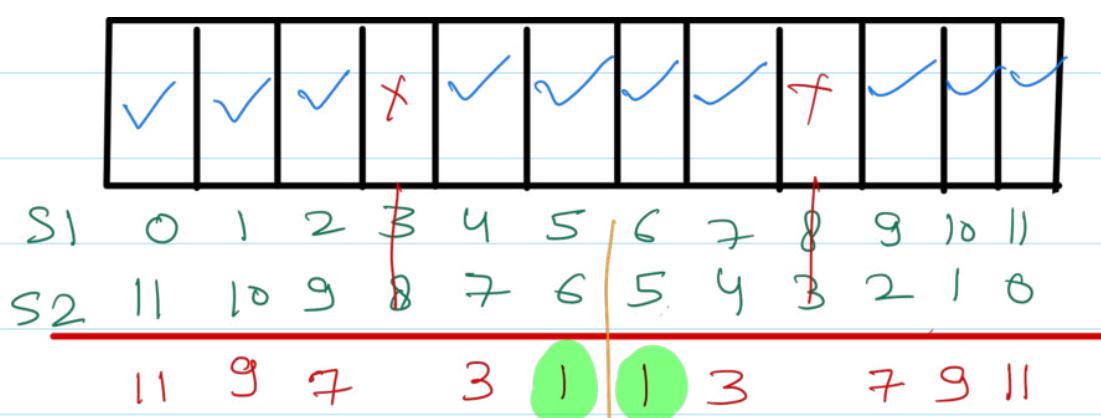
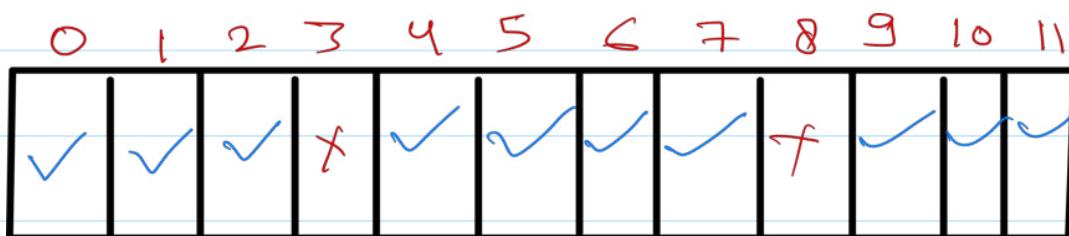
You just need to find the minimum absolute difference considering any valid division of the array elements.

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.

Ex → {1, 5, 4, 13}



*we can run check for
sum/2.*



```
1 int minSubsetSumDifference(vector<int>& nums, int n)
2 {   int k=0;
3     for(int i =0;i<n;i++)
4     {   k+=nums[i];
5     }
6     vector<vector<bool>> dp(n,vector<bool> (k+1,false));
7     for(int i=0;i<n;i++)
8     {   dp[i][0] = true; }
9     if(nums[n-1]<=k)
10       dp[n-1][nums[n-1]] = true;
11     for(int currind=n-2;currind>=0;currind--)
12     {   for(int target=k;target>=1;target--)
13         {   bool consider = false;
14             if(target>=nums[currind])
15               consider = dp[currind+1][target-nums[currind]];
16             bool notconsider = dp[currind+1][target];
17             dp[currind][target] = consider + notconsider;
18         }
19     }
20
21 int mini = 1e9;
22 for(int s1=0;s1<=k;s1++)
23 {   if(dp[0][s1]==true)
24     {   int s2 = k-s1;
25       mini = min(mini,abs(s2-s1));
26     }
27 }
28 return mini;
29 }
```

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

SPACE OPTIMIZATION:



```
1 int minSubsetSumDifference(vector<int>& nums, int n)
2 {    int k=0;
3     for(int i =0;i<n;i++)
4     {    k+=nums[i];
5     }
6     vector<bool> next(k+1,0);
7
8     next[0]=1;
9     if(nums[n-1]<=k)
10    next[nums[n-1]] = true;
11    for(int currind=n-2;currind>=0;currind--)
12    {   vector<bool> curr(k+1,0);
13        for(int target=k;target>=0;target--)
14        {   bool consider = false;
15            if(target>=nums[currind])
16                consider = next[target-nums[currind]];
17            bool notconsider = next[target];
18            curr[target] = consider + notconsider;
19        }
20        next = curr;
21    }
22    int mini = 1e9;
23    for(int s1=0;s1<=k/2;s1++)
24    {   if(next[s1]==true)
25        {   int s2 = k-s1;
26            mini = min(mini,abs(s2-s1));
27        }
28    }
29 return mini;
30 }
```

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:

```
● ● ●
```

```
1 int countways(int currind, int tar, vector<int>& nums,
   vector<vector<int>>& dp)
2 {   if(tar==0)
3     return 1;
4
5   if(currind==nums.size()-1)
6     return (nums[currind]==tar);
7   if(currind>nums.size()-1 )
8     return 0;
9   if(dp[currind][tar]!=-1)
10    return dp[currind][tar];
11   int pick = 0;
12   if(nums[currind]<=tar)
13     pick = countways(currind+1,tar-
14       nums[currind],nums,dp);
14     int notpick = countways(currind+1,tar,nums,dp);
15   return dp[currind][tar] = pick+notpick;
16 }
17
18 int findWays(vector<int> &nums, int tar)
19 {  int n= nums.size();
20  vector<vector<int>> dp(n,vector<int> (tar+1,-1));
21  return countways(0,tar,nums,dp);
22 }
```

TC: $O(N \cdot K)$

SC : $O(N \cdot k) + O(N)$

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

Tabulation:



```
1 int findWays(vector<int> &nums, int tar)
2 { int n= nums.size();
3 vector<vector<int>> dp(n, vector<int> (tar+1,0));
4 for(int i =0;i<n;i++)
5     dp[i][0]=1;
6 if(nums[n-1]<=tar)
7     dp[n-1][nums[n-1]] = 1;
8 for(int currind=n-2;currind>=0;currind--)
9 { for(int target=tar;target>=1;target--)
10     { int pick = 0;
11         if(nums[currind]<=target)
12             pick = dp[currind+1][target-nums[currind]];
13         int notpick = dp[currind+1][target];
14         dp[currind][target] = pick+notpick;
15     }
16 }return dp[0][tar];
17 }
```

TC: $O(N \cdot K)$

SC : $O(N \cdot k)$

Space Optimization:



```
1 int findWays(vector<int> &nums, int k)
2 { int n= nums.size();
3  vector<int> next(k+1,0);
4   next[0]=1;
5   if(nums[n-1]<=k)
6     next[nums[n-1]] = 1;
7   for(int currind=n-2;currind>=0;currind--)
8   {   vector<int> curr(k+1,0);
9    curr[0]=1;
10   for(int target=k;target>=1;target--)
11   { int consider = 0;
12     if(target>=nums[currind])
13       consider = next[target-nums[currind]];
14     int notconsider = next[target];
15     curr[target] = consider + notconsider;
16   }
17   next = curr;
18 }
19 return next[k];
20 }
```

Time Complexity: $O(N*K)$

Space Complexity: $O(K)$



```
1 int findWays(vector<int> &nums, int k)
2 { int n= nums.size();
3  vector<int> next(k+1,0);
4   next[0]=1;
5   if(nums[n-1]<=k)
6     next[nums[n-1]] = 1;
7   for(int currind=n-2;currind>=0;currind--)
8   {
9     vector<int> curr(k+1,0);
10    //curr[0]=1;
11    for(int target=k;target>=0;target--)
12    {
13      int consider = 0;
14      if(target>=nums[currind])
15        consider = next[target-nums[currind]];
16      int notconsider = next[target];
17      curr[target] = consider + notconsider;
18    }
19    next = curr;
20  }
21  return next[k];
22 }
```

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 $1 \leq \text{nums}[i] \leq 1000$, so modify our base cases.

- If our $\text{nums}[n-1] == 0$ and $\text{target} == 0$, 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 $\text{nums}[n-1]$ doesn't equal to 0, then return 1, because, the upper path is covering the target.
- If our $\text{target} == \text{nums}[n-1]$ then also, return 1, because adding $\text{nums}[n-1]$ will equal to our target.
- If $\text{target} != \text{nums}[n-1]$, then we don't get a path, return 0.

Memoization:

```
1 int countways(int currind, int tar, vector<int>& nums,
   vector<vector<int>>& dp)
2 {  int n=nums.size();
3   if(currind==n-1)
4     {  if(tar==0 && nums[n-1]==0)
5        return 2;
6        if(tar==0 || tar==nums[n-1]) return 1;
7        return 0;
8     }
9
10  if(dp[currind][tar]!=-1)
11    return dp[currind][tar];
12  int pick = 0;
13  if(nums[currind]<=tar)
14    pick = countways(currind+1,tar-nums[currind],nums,dp);
15    int notpick = countways(currind+1,tar,nums,dp);
16  return dp[currind][tar] = pick+notpick;
17 }
18
19 int findWays(vector<int> &nums, int tar)
20 {  int n= nums.size();
21  vector<vector<int>> dp(n,vector<int> (tar+1,-1));
22  return countways(0,tar,nums,dp);
23 }
```

If in constraint, $0 \leq N$, 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 $\text{target} == 0$, if $\text{target} == 0$, there can be case, $\text{nums}[i]$ 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 , 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:

```
● ○ ● ●  
1 int findWays(vector<int> &nums, int tar)  
2 {  int n= nums.size();  
3  vector<vector<int>> dp(n, vector<int> (tar+1,0));  
4  if(nums[n-1]==0)  
5      dp[n-1][0]=2;  
6  else dp[n-1][0]=1;  
7  if(nums[n-1]!=0 && nums[n-1]<=tar)  
8      dp[n-1][nums[n-1]]=1;  
9  
10 for(int currind=n-2;currind>=0;currind--)  
11 {   for(int target=tar;target>=0;target--)  
12     {   int pick = 0;  
13       if(nums[currind]<=target)  
14           pick = dp[currind+1][target-nums[currind]];  
15       int notpick = dp[currind+1][target];  
16       dp[currind][target] = pick+notpick;  
17     }  
18 }return dp[0][tar];  
19 }
```

Space Optimization:



```
1 int findWays(vector<int> &nums, int tar)
2 {   int n= nums.size();
3   vector<int> next(tar+1,0);
4   if(nums[n-1]==0)
5     next[0]=2;
6   else next[0]=1;
7   if(nums[n-1]!=0 && nums[n-1]<=tar)
8     next[nums[n-1]]=1;
9   for(int currind=n-2;currind>=0;currind--)
10 {   for(int target=tar;target>=0;target--)
11     {   int pick = 0;
12       if(nums[currind]<=target)
13         pick = next[target-nums[currind]];
14       int notpick = next[target];
15       next[target] = pick+notpick;
16     }
17   }
18   return next[tar];
19 }
```

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

- 1) $P_1 S_1 \neq P_2 S_1$ i.e., at least one of the elements of $P_1 S_1$ is different from $P_2 S_2$.
- 2) $P_1 S_1 == P_2 S_2$, but the indices set represented by $P_1 S_1$ is not equal to the indices set of $P_2 S_2$. Here, the indices set of $P_1 S_1$ 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 $0 \leq \text{nums}[i] \leq 50$, so use Lecture-17 updated code(last code)

$$S_1 \geq S_2$$
$$S_1 - S_2 = D$$

$$S_1 > S_2,$$

$$S_1 = \text{totalsum} - S_2$$

$$S_1 - S_2 = D$$

$$\text{totalsum} - S_2 - S_2 = D$$

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

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

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

ex = $N = 4, D = 3, \text{arr} = \{5, 2, 5, 1\}$

$$P_1 = \{5, 2, 1\}, P_2 = \{5\}$$

$$P_1 = \{5, 2, 1\}, P_2 = \{5\}$$

$$\text{totalsum} = 5 + 2 + 5 + 1 = 13$$

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

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

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

→ $\text{totalsum} \geq d$, then only we can find count of subsets

Memoization:

```
1 int mod =(int)(1e9 + 7);
2 int countways(int currind, int tar, vector<int>& nums,
   vector<vector<int>>& dp)
3 {   int n=nums.size();
4     if(currind==n-1)
5       {   if(tar==0 && nums[n-1]==0)
6           return 2;
7           if(tar==0 || tar==nums[n-1]) return 1;
8           return 0;
9       }
10    if(dp[currind][tar]!=-1)
11      return dp[currind][tar];
12    int pick = 0;
13    if(nums[currind]<=tar)
14      pick = countways(currind+1,tar-nums[currind],nums,dp);
15      int notpick = countways(currind+1,tar,nums,dp);
16    return dp[currind][tar] = (pick+notpick)%mod;
17  }
18
19 int countPartitions(int n, int d, vector<int> &arr) {
20   int totalsum = 0;
21   for(int i=0;i<n;i++)
22     totalsum+=arr[i];
23   if((totalsum-d)<0 || (totalsum-d)%2!=0)
24     return 0;
25   int tar = (totalsum-d)/2;
26   vector<vector<int>> dp(n,vector<int> (tar+1,-1));
27   return countways(0,tar,arr,dp);
28 }
```

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

Space Complexity: $O(N*K) + O(N)$, where $K = \text{totalsum}-d/2$

Tabulation:

```
● ● ●  
1 int mod =(int)(1e9 + 7);  
2 int countways(int tar, vector<int>& nums)  
3 {    int n= nums.size();  
4     vector<vector<int>> dp(n,vector<int> (tar+1,0));  
5     if(nums[n-1]==0)  
6         dp[n-1][0]=2;  
7     else dp[n-1][0]=1;  
8     if(nums[n-1]!=0 && nums[n-1]<=tar)  
9         dp[n-1][nums[n-1]]=1;  
10  
11     for(int currind=n-2;currind>=0;currind--)  
12     {        for(int target=tar;target>=0;target--)  
13             {            int pick = 0;  
14             if(nums[currind]<=target)  
15                 pick = dp[currind+1][target-nums[currind]];  
16             int notpick = dp[currind+1][target];  
17             dp[currind][target] = (pick+notpick)%mod;  
18         }  
19     }return dp[0][tar];  
20 }  
21  
22 int countPartitions(int n, int d, vector<int> &arr) {  
23     int totalsum = 0;  
24     for(int i=0;i<n;i++)  
25         totalsum+=arr[i];  
26     if((totalsum-d)<0 || (totalsum-d)%2!=0)  
27         return 0;  
28     int tar = (totalsum-d)/2;  
29     return countways(tar,arr);  
30 }
```

[LinkedIn](#)/kapilyadav22

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

Space Complexity: $O(N*K)$, where $K = \text{totalsum}-d/2$

Space Optimization:

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

Space Complexity: $O(K)$, where $K = \text{totalsum}-d/2$



```
1 int mod =(int)(1e9 + 7);
2 int countways(int tar, vector<int>& nums)
3 {   int n= nums.size();
4     vector<int> next(tar+1,0);
5     if(nums[n-1]==0)
6         next[0]=2;
7     else next[0]=1;
8     if(nums[n-1]!=0 && nums[n-1]<=tar)
9         next[nums[n-1]]=1;
10    for(int currind=n-2;currind>=0;currind--)
11    {   vector<int> curr(tar+1,0);
12        for(int target=tar;target>=0;target--)
13        {   int pick = 0;
14            if(nums[currind]<=target)
15                pick = next[target-nums[currind]];
16            int notpick = next[target];
17            curr[target] = (pick+notpick)%mod;
18        }
19        next = curr;
20    }
21    return next[tar];
22 }
23
24 int countPartitions(int n, int d, vector<int> &arr)
{
25     int totalsum = 0;
26     for(int i=0;i<n;i++)
27         totalsum+=arr[i];
28     if((totalsum-d)<0 || (totalsum-d)%2!=0)
29         return 0;
30     int tar = (totalsum-d)/2;
31     return countways(tar,arr);
32 }
```

*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 $\text{val}[0..N-1]$ and $\text{wt}[0..N-1]$ 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 $\text{val}[]$ 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:

$N = 3$

$W = 4$

$\text{values}[] = \{1, 2, 3\}$

$\text{weight}[] = \{4, 5, 1\}$

Output: 3

Can we use greedy?

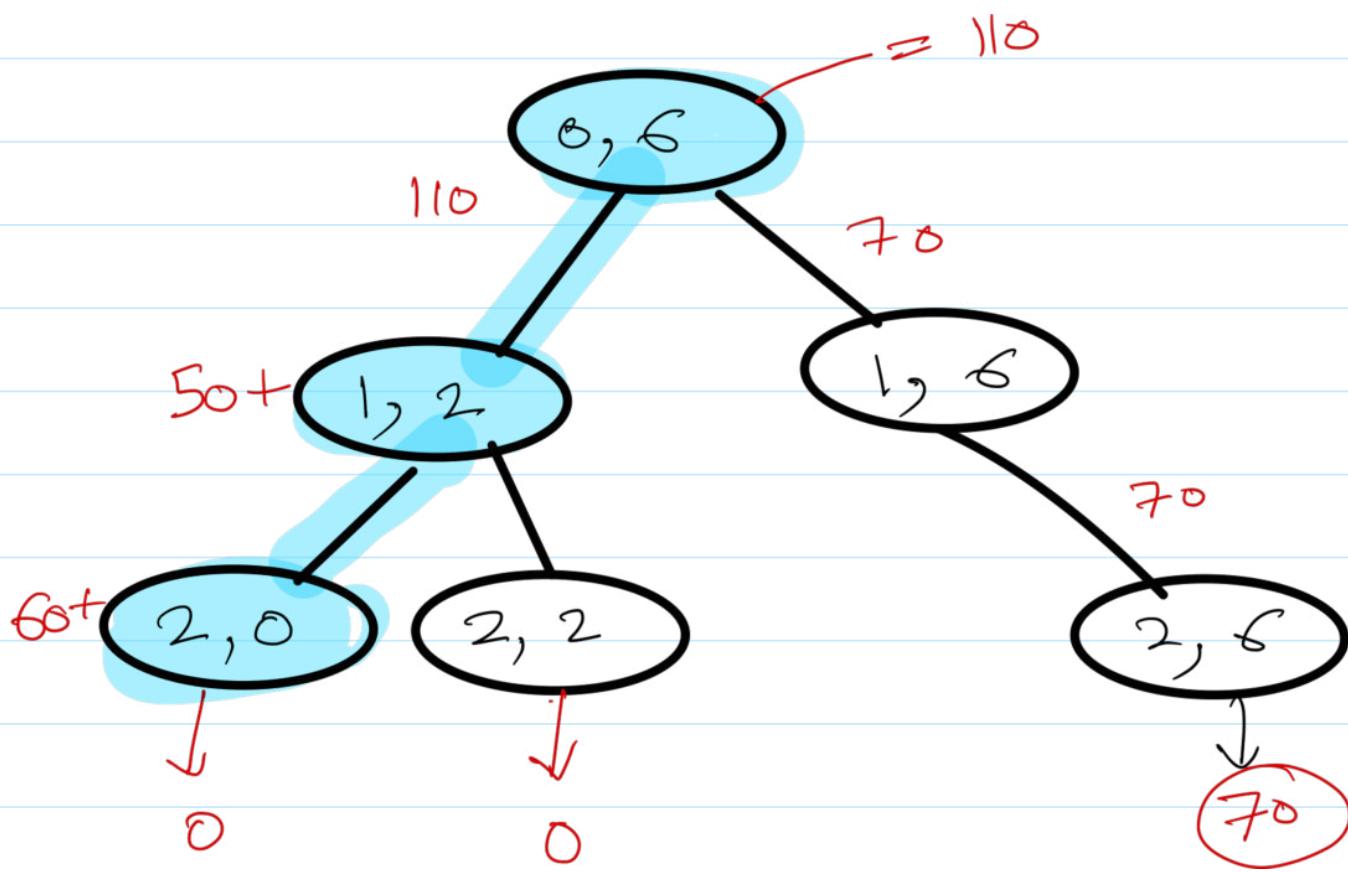
Ex → weight → 4 2 6
value → 50 60 70
 $W = 6$.

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 $\rightarrow 50 + 60 = \underline{\underline{110}}$.

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

```

TC : O(N*W)

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

Tabulation:



```
1 int knapsack(vector<int>& wt, vector<int>& profit, int n, int W)
2 {  vector<vector<int>> dp(n, vector<int> (W+1,0));
3
4   for(int i=wt[n-1];i<=W;i++)
5   {  dp[n-1][i] = profit[n-1];
6   }
7   for(int currind=n-2;currind>=0;currind--)
8   {  for(int capacity=W;capacity>=0;capacity--)
9     {    int notconsider = 0 + dp[currind+1][capacity];
10      int consider = INT_MIN;
11      if(wt[currind]<=capacity)
12        consider = profit[currind] + dp[currind+1][capacity-wt[currind]];
13      dp[currind][capacity] = max(consider,notconsider);
14    }
15  }
16  return dp[0][W];
17 }
```

TC : O(N*W)

SC : O(N*W)

Space Optimization:

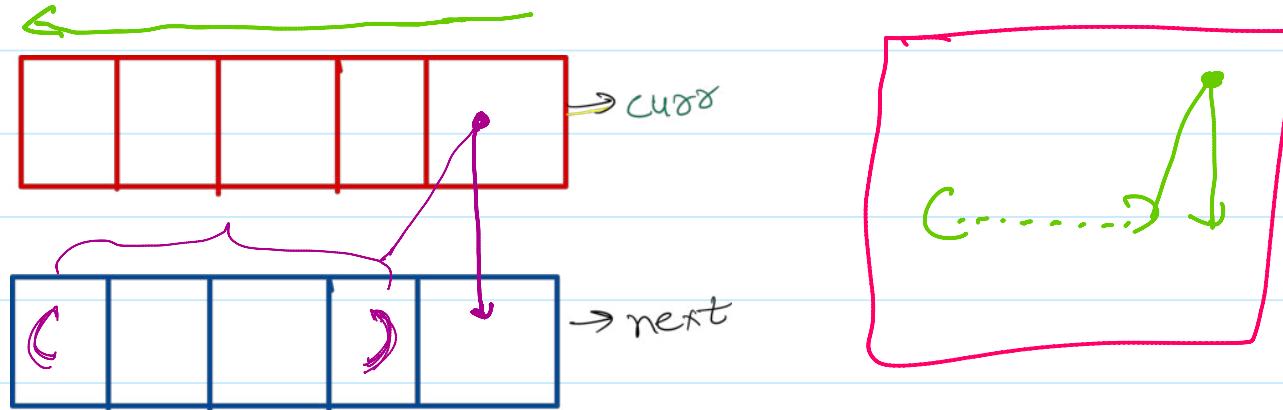


```
1 int knapsack(vector<int>& wt, vector<int>& profit, int n, int W)
2 {  vector<int> next(W+1,0);
3   for(int i=wt[n-1];i<=W;i++)
4   {  next[i] = profit[n-1];
5   }
6   for(int currind=n-2;currind>=0;currind--)
7   {  vector<int> curr(W+1,0);
8     for(int capacity=W;capacity>=0;capacity--)
9     {    int notconsider = next[capacity];
10       int consider = INT_MIN;
11       if(wt[currind]<=capacity)
12         consider = profit[currind] + next[capacity-wt[currind]];
13       curr[capacity] = max(consider,notconsider);
14     }
15   next = curr;
16 }
17 return next[W];
18 }
```

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.



```

● ● ●

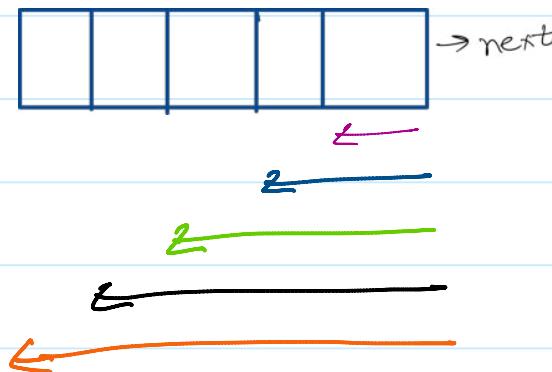
1 int knapsack(vector<int>& wt, vector<int>& profit, int n, int W)
2 {   vector<int> next(W+1,0);
3     for(int i=wt[n-1];i<=W;i++)
4     {   next[i]= profit[n-1];
5     }
6     for(int currind=n-2;currind>=0;currind--)
7     {   for(int capacity=W;capacity>=0;capacity--)
8         {   int notconsider =next[capacity];
9             int consider = INT_MIN;
10            if(wt[currind]<=capacity)
11                consider = profit[currind] + next[capacity-wt[currind]];
12            next[capacity] = max(consider,notconsider);
13        }
14    }
15    return next[W];
16 }

```

TC : O(N*W)

SC : O(W)

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/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`.

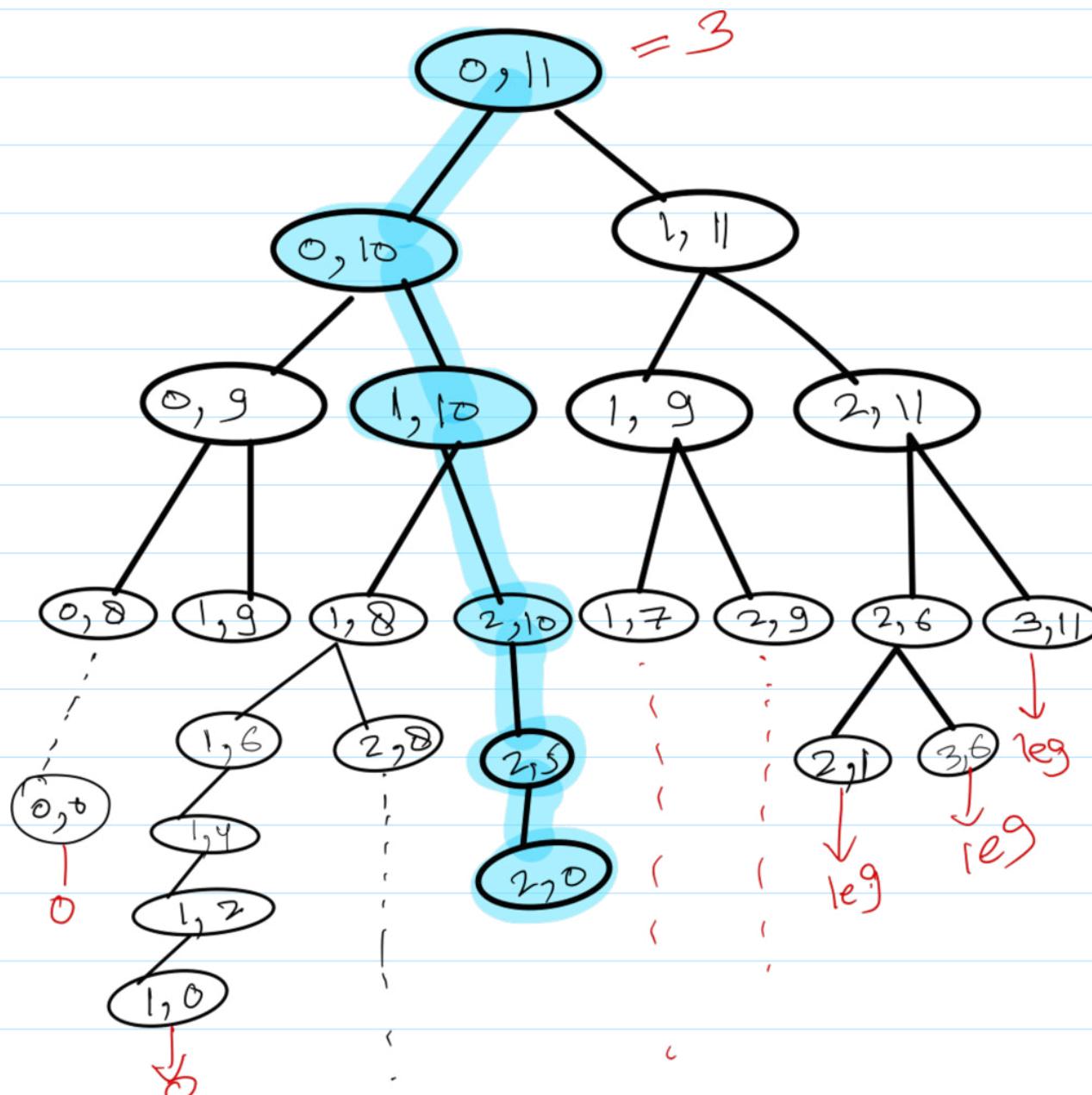
You may assume that you have an infinite number of each kind of coin.

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.

$$\rightarrow 1+1+1+\dots = 11$$

$$\rightarrow 1+2+2+2+2+2 = 11$$

$$\rightarrow 1+1+2+2+5 = 11$$

$$\rightarrow 2+2+2+5 = 11$$

$$\rightarrow \underline{1+5+5} = \underline{11}$$

MEMOIZATION:

```
● ● ●

1 int calcoins(int currind, int lastindex, int price, vector<int> & coins,
   vector<vector<int>> & dp)
2 {   if(price==0)
3     return 0;
4   if(currind>lastindex)
5     return 1e9;
6   if(dp[currind][price]!=-1)
7     return dp[currind][price];
8   int notconsider = calcoins(currind+1, lastindex, price, coins, dp);
9   int consider = 1e9;
10  if(coins[currind]<=price)
11    consider = 1 + calcoins(currind, lastindex, price-coins[currind], coins, dp);
12  return dp[currind][price] = min(consider, notconsider);
13 }
14
15 int minimumElements(vector<int> &num, int x)
16 {   int n = num.size();
17   vector<vector<int>> dp(n, vector<int> (x+1, -1));
18   int ans = calcoins(0, n-1, x, num, dp);
19   if(ans>=1e9) return -1;
20   return ans;
21 }
```

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

SC : O(N*x) + O(x),

Tabulation:

```
● ● ●
```

```
1 int minimumElements(vector<int> &coins, int x)
2 {    int n = coins.size();
3     vector<vector<int>> dp(n, vector<int> (x+1, 1e9));
4     for(int i = 0; i <= x; i++)
5     {    if(i % coins[n-1] == 0)
6         dp[n-1][i] = (i / coins[n-1]);
7     else
8         dp[n-1][i] = 1e9;
9    }
10    for(int currind = n-2; currind >= 0; currind--)
11    {    for(int price = 0; price <= x; price++)
12        {int notconsider = dp[currind+1][price];
13         int consider = 1e9;
14         if(coins[currind] <= price)
15             consider = 1 + dp[currind][price - coins[currind]];
16         dp[currind][price] = min(consider, notconsider);
17        }
18    }
19    int ans = dp[0][x];
20    if(ans >= 1e9)
21        return -1;
22    else return ans;
23 }
```

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

SC : O(N*x)

Space Optimization:

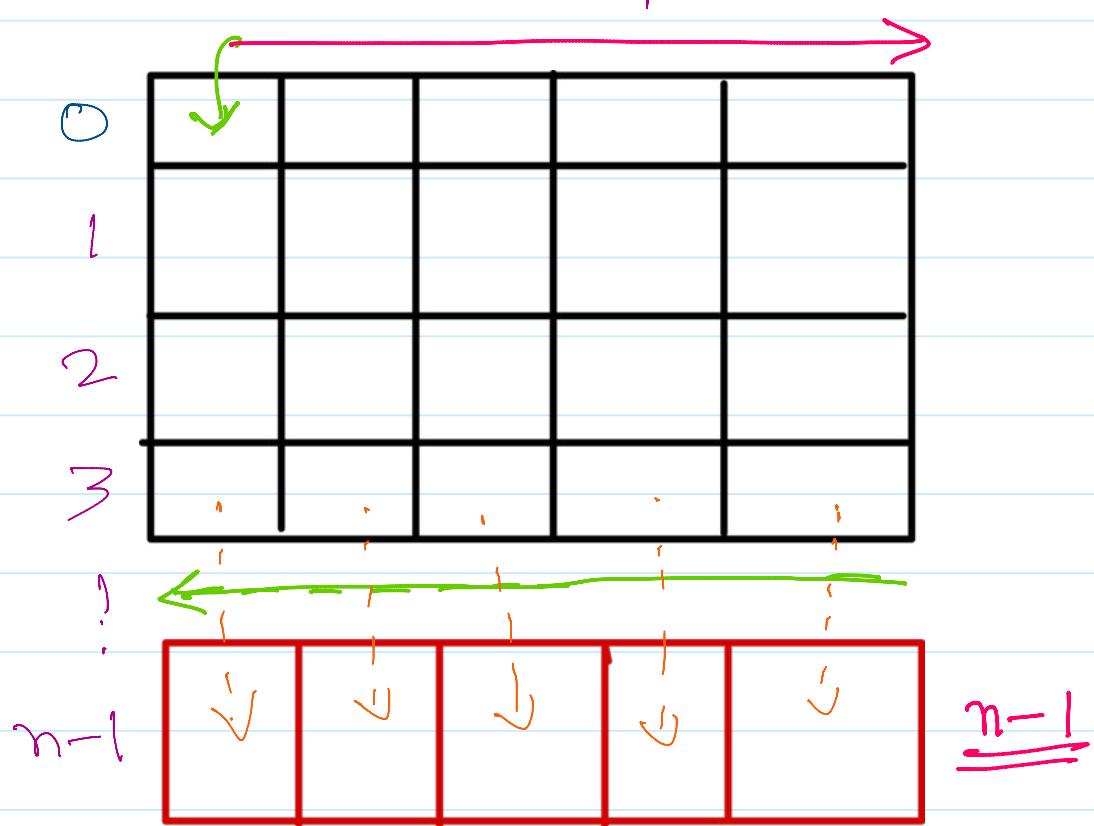
```
1 int minimumElements(vector<int> &coins, int x)
2 {    int n = coins.size();
3     vector<int> next(x+1,0);
4     for(int i =0;i<=x;i++)
5     {    if(i%coins[n-1]==0)
6         next[i] = (i/coins[n-1]);
7     else
8         next[i] = 1e9;
9    }
10    for(int currind=n-2;currind>=0;currind--)
11    {   vector<int> curr(x+1,0);
12        for(int price =0;price<=x;price++)
13        {int notconsider = next[price];
14            int consider = 1e9;
15            if(coins[currind]<=price)
16                consider = 1 + curr[price-coins[currind]];
17            curr[price] = min(consider,notconsider);
18        }
19        next = curr;
20    }
21    int ans = next[x];
22    if(ans>=1e9)
23        return -1;
24    else return ans;
25 }
```

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. $dp[\text{current index} + 1]$



⇒ 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.



```
1 int minimumElements(vector<int> &coins, int x)
2 {    int n = coins.size();
3     vector<int> next(x+1,0);
4     for(int i =0;i<=x;i++)
5     {    if(i%coins[n-1]==0)
6         next[i] = (i/coins[n-1]);
7     else
8         next[i] = 1e9;
9    }
10    for(int currind=n-2;currind>=0;currind--)
11    {
12        for(int price =0;price<=x;price++)
13        {int notconsider = next[price];
14            int consider = 1e9;
15            if(coins[currind]<=price)
16                consider = 1 + next[price-coins[currind]];
17            next[price] = min(consider,notconsider);
18        }
19    }
20    int ans = next[x];
21    if(ans>=1e9)
22        return -1;
23    else return ans;
24 }
```

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:

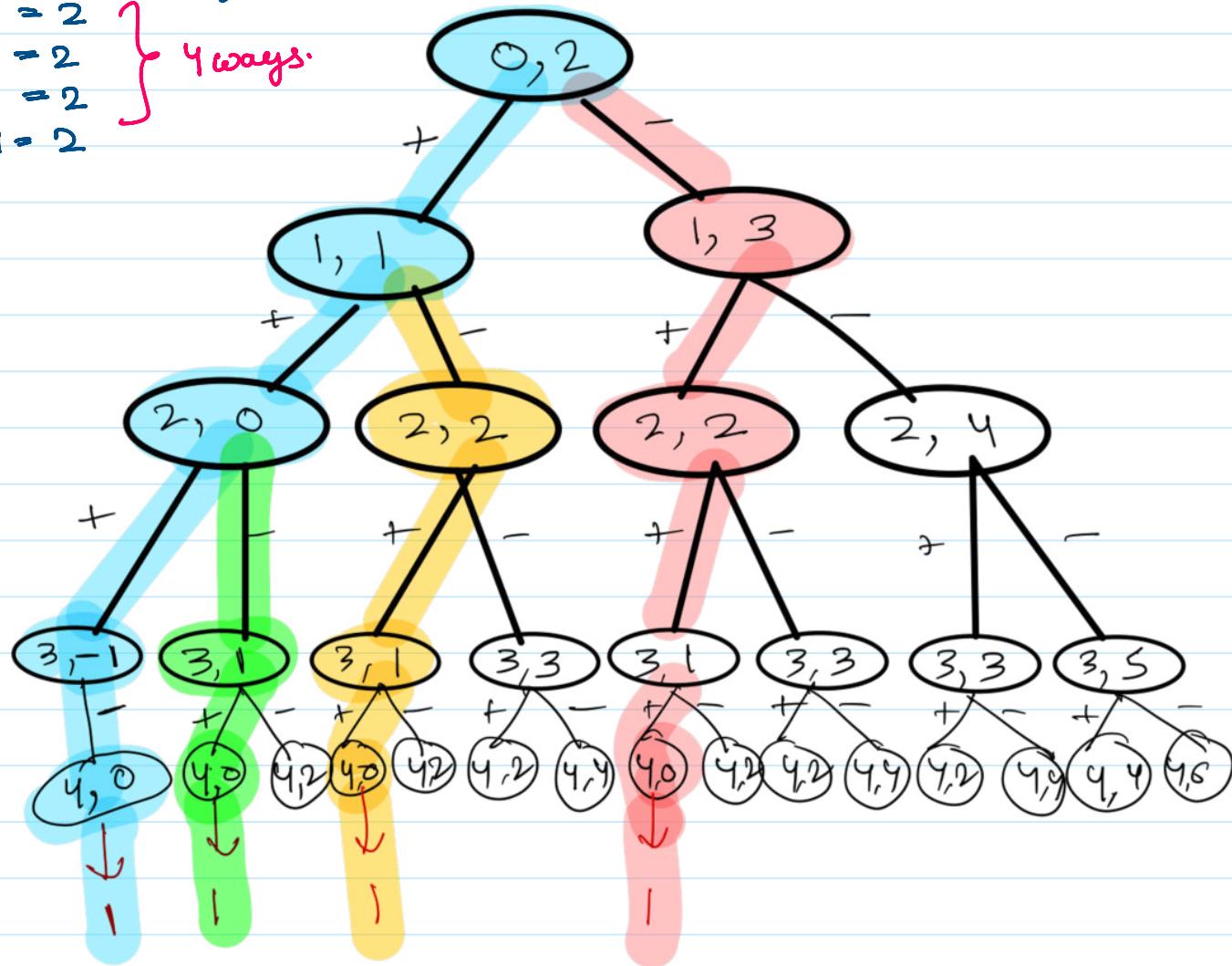
1. You can do combinations of additions and subtraction in the array elements.

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;

Ex- {1, 1, 1, 1}, Target = 2

$$\begin{aligned} &+1+1+1-1 = 2 \\ &+1+1-1+1 = 2 \\ &+1-1+1+1 = 2 \\ &-1+1+1+1 = 2 \end{aligned}$$

} 4 ways.





```
1 int findTargetSumWays(vector<int>& nums, int target) {
2     unordered_map<string,int>mp;
3     return totalways(0,target,nums,mp);
4 }
5
6 int totalways(int currentindex, int target,vector<int>& nums,unordered_map<string,int>&mp)
7 {   int n=nums.size();
8     if(currentindex>=n & target!=0)
9         return 0;
10
11    if(currentindex>=n & target==0)
12        return 1;
13    string currentkey = to_string(currentindex)+"_"+to_string(target);
14    if(mp.find(currentkey)!=mp.end())
15        return mp[currentkey];
16
17    int plus = totalways(currentindex+1,target - nums[currentindex], nums,mp);
18    int minus = totalways(currentindex+1, target+nums[currentindex], nums,mp);
19
20    return mp[currentkey]= plus+minus;
21
22 }
```

TC: $O(N \cdot K)$, where K is the target

SC: $O(N \cdot K) + O(N)$

2nd Way:

$$\{1, 1, 1, 1\} \text{ Target} = 2$$

$$\underbrace{\{1+1+1\}}_{S_1} - \underbrace{\{1\}}_{S_2} = 2$$

→ It can be solved using count partition with given difference

Tabulation:

```
● ● ●
```

```
1 int countways(int tar, vector<int>& nums)
2 {   int n= nums.size();
3   vector<int> next(tar+1,0);
4   if(nums[n-1]==0)
5     next[0]=2;
6   else next[0]=1;
7   if(nums[n-1]!=0 && nums[n-1]<=tar)
8     next[nums[n-1]]=1;
9   for(int currind=n-2;currind>=0;currind--)
10  {    vector<int> curr(tar+1,0);
11    for(int target=tar;target>=0;target--)
12      {  int pick = 0;
13        if(nums[currind]<=target)
14          pick = next[target-nums[currind]];
15        int notpick = next[target];
16        curr[target] = (pick+notpick);
17      }
18    next = curr;
19  }
20 return next[tar];
21 }
22
23
24 int targetSum(int n, int target, vector<int>& arr) {
25   int totalsum = 0;
26   for(int i=0;i<n;i++)
27     totalsum+=arr[i];
28   if((totalsum-target)<0 || (totalsum-target)%2!=0)
29     return 0;
30   int tar = (totalsum-target)/2;
31   return countways(tar,arr);
32 }
33 TC: O(N*K), SC: O(N*K), where K = totalsum-target/2
```

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/kapilyadav22)

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.

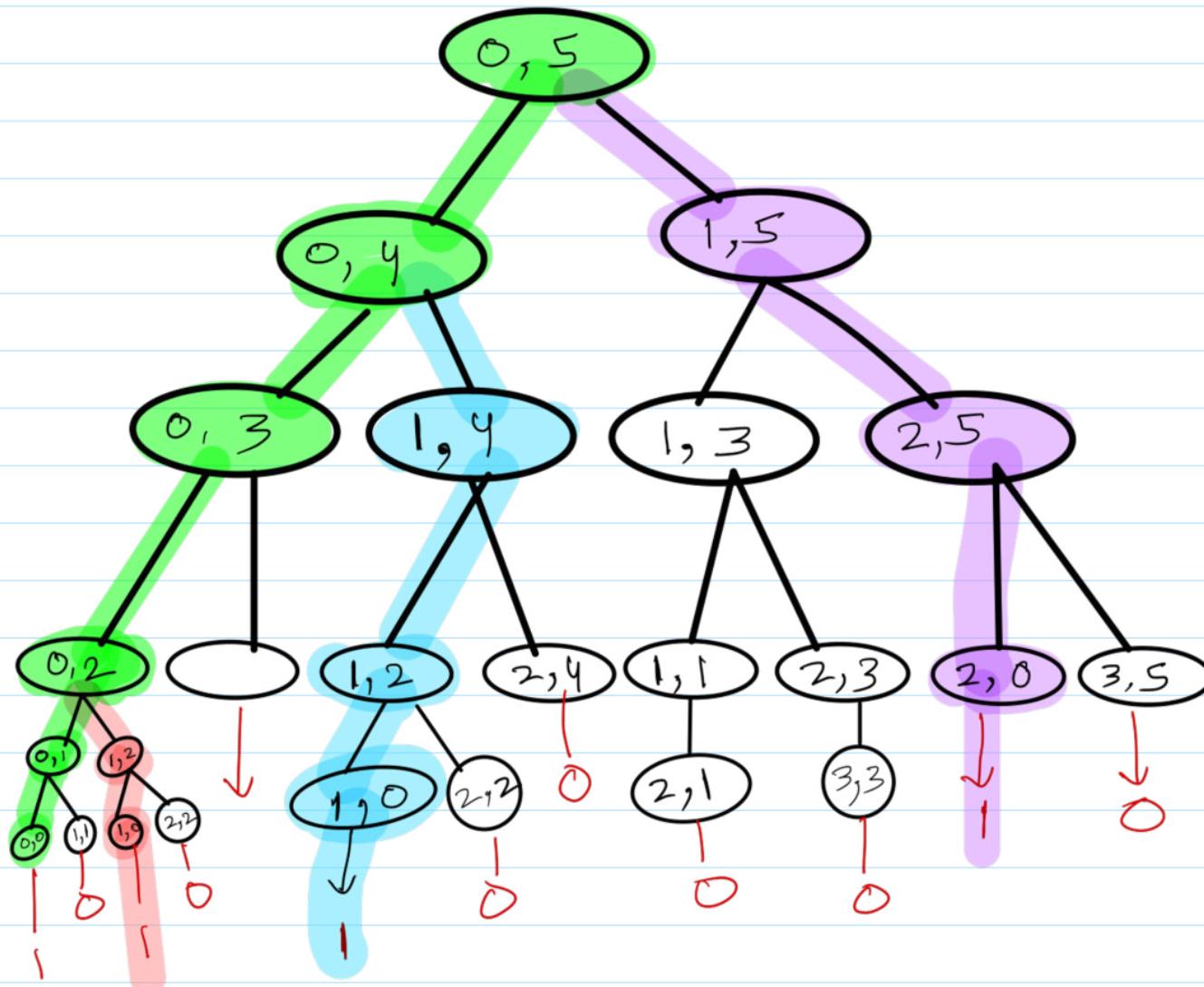
You may assume that you have an infinite number of each kind of coin.

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:

```
1 #include<bits/stdc++.h>
2 long countways(int currind,int lastindex,int value,int *denominations,
    vector<vector<long>> & dp)
3 {   if(value==0)
4     return 1;
5   if(currind>lastindex)
6     return 0;
7   if(dp[currind][value]!=-1)
8     return dp[currind][value];
9   long notpick = countways(currind+1,lastindex,value,denominations,dp);
10  long pick = 0;
11  if(denominations[currind]<=value)
12    pick = countways(currind,lastindex,value-
    denominations[currind],denominations,dp);
13  return dp[currind][value]= pick + notpick;
14 }
15
16
17 long countWaysToMakeChange(int *denominations, int n, int value)
18 {  vector<vector<long>> dp(n,vector<long> (value+1,-1));
19  return countways(0,n-1,value,denominations,dp);
20 }
```

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

Tabulation:

```
1 long countWaysToMakeChange(int *denominations, int n, int value)
2 {  vector<vector<long>> dp(n,vector<long> (value+1,0));
3
4  for(int i=0;i<=value;i++)
5  {    dp[n-1][i] = (i%denominations[n-1]==0);
6  }
7  for(int currind=n-2;currind>=0;currind--)
8  {    for(int price =0;price<=value;price++)
9      { long notpick = dp[currind+1][price];
10        long pick = 0;
11        if(denominations[currind]<=price)
12          pick = dp[currind][price-denominations[currind]];
13        dp[currind][price]= pick + notpick;
14      }
15  }
16  return dp[0][value];
17 }
```

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

Space Optimization:

```
● ● ●

1 long countWaysToMakeChange(int *denominations, int n, int value)
2 {  vector<int> next(value+1,0);
3   for(int i=0;i<=value;i++)
4   {    next[i]= (i%denominations[n-1]==0);
5   }
6   for(int currind=n-2;currind>=0;currind--)
7   {    vector<int> curr(value+1,0);
8     for(int price =0;price<=value;price++)
9       { long notpick = next[price];
10        long pick = 0;
11        if(denominations[currind]<=price)
12          pick = curr[price-denominations[currind]];
13        curr[price]= pick + notpick;
14      }
15      next = curr;
16    }
17  return next[value];
18 }
```

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

SC: O(K)

Further Space Optimization: Using one 1D array only

```
● ● ●

1 long countWaysToMakeChange(int *denominations, int n, int value)
2 {  vector<long> next(value+1,0);
3   for(int i=0;i<=value;i++)
4   {    next[i]= (i%denominations[n-1]==0);
5   }
6   for(long currind=n-2;currind>=0;currind--)
7   {
8     for(long price =0;price<=value;price++)
9       { long notpick = next[price];
10        long pick = 0;
11        if(denominations[currind]<=price)
12          pick = next[price-denominations[currind]];
13        next[price]= pick + notpick;
14      }
15    }
16  return next[value];
17 }
```

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

SC: O(K)

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/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.

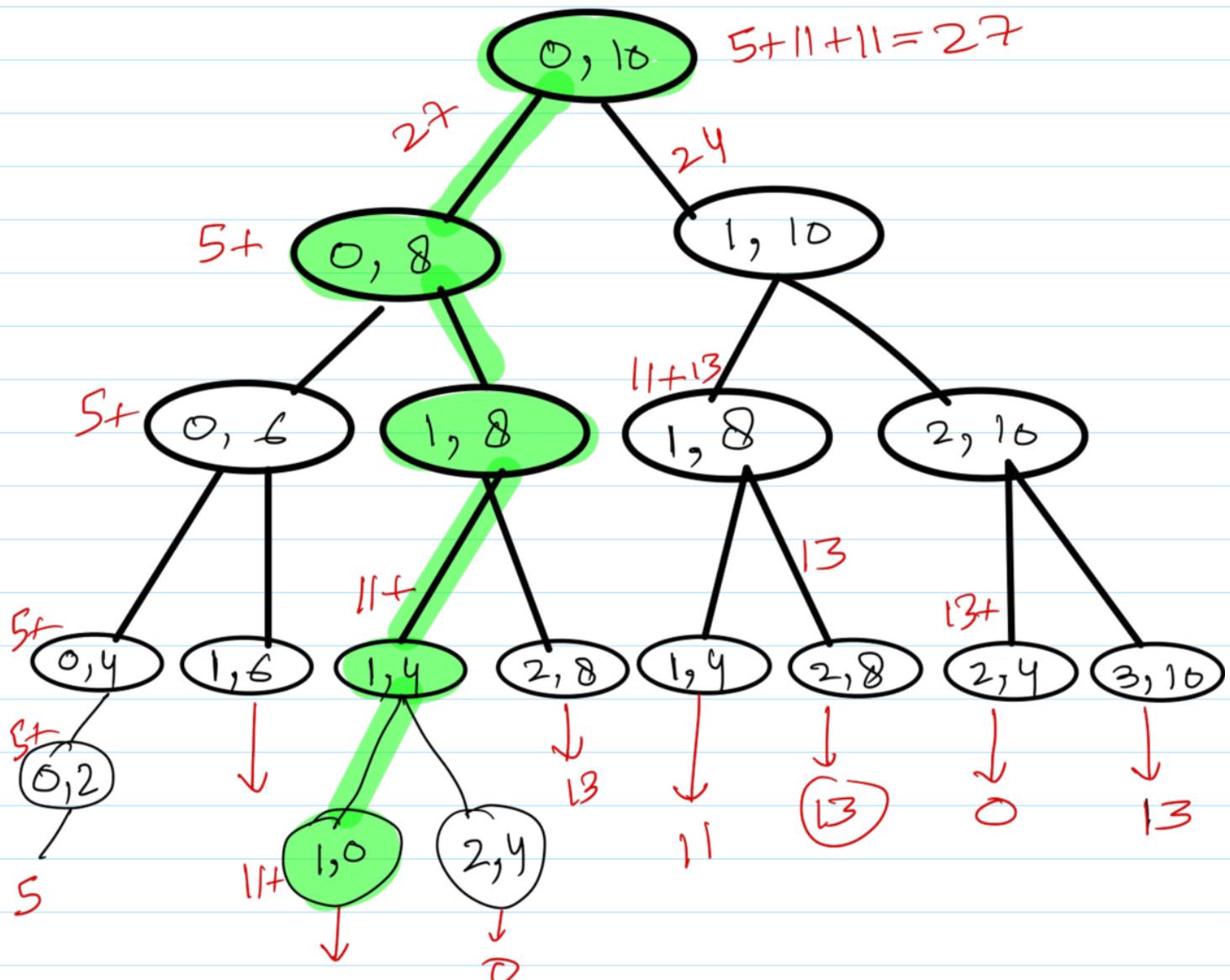
The maximum profit will be from case 3 i.e '27'. Therefore maximum profit = 27.

wt	2	4	6	13
value	5	11	13	

unlimited supply

$$W = 10$$

$$\begin{aligned} \rightarrow & 5 \times 5 = 25, \\ \rightarrow & 11 + 11 + 5 = 27 \\ \rightarrow & 13 + 11 = 24 \\ \rightarrow & 13 + 10 = 23 \end{aligned}$$



MEMOIZATION:

```
1 int findknap(int currind, int lastind, int capacity, vector<int>
2 &profit, vector<int> &weight,
3 &vector<vector<int>> & dp)
4 { if(currind==lastind)
5 { return ((int)(capacity/weight[lastind])) * profit[lastind];
6 }
7 if(dp[currind][capacity]!=-1)
8 return dp[currind][capacity];
9 int notpick =
10 findknap(currind+1,lastind,capacity,profit,weight,dp);
11 int pick=-1e9;
12 if(weight[currind]<=capacity)
13 pick = profit[currind] + findknap(currind,lastind,capacity-
14 weight[currind],profit,weight,dp);
15 return dp[currind][capacity]= max(pick,notpick);
16 }
17 int unboundedKnapsack(int n, int w, vector<int> &profit, vector<int>
&weight)
18 { vector<vector<int>> dp(n,vector<int> (w+1,-1));
19 return findknap(0,n-1,w,profit,weight,dp);
20 }
```


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

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

Tabulation:

```
● ○ ●

1 int unboundedKnapsack(int n, int w, vector<int> &profit, vector<int>
2   &weight)
3 {  vector<vector<int>> dp(n, vector<int> (w+1, 0));
4   for(int i=weight[n-1]; i<=w; i++)
5     dp[n-1][i] = ((int)(i/weight[n-1])) * profit[n-1];
6
7   for(int currind=n-2; currind>=0; currind--)
8   {   for(int capacity=0; capacity<=w; capacity++)
9     { int notpick = dp[currind+1][capacity];
10    int pick=0;
11    if(weight[currind]<=capacity)
12      pick = profit[currind] + dp[currind][capacity-weight[currind]];
13    dp[currind][capacity]= max(pick,notpick);
14    }
15  }
16 }

return dp[0][w];
17 }
```

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

SC: O(N*W)

Space Optimization:

```
● ○ ●

1 int unboundedKnapsack(int n, int w, vector<int> &profit, vector<int>
2   &weight)
3 {  vector<int> next(w+1, 0);
4   for(int i=weight[n-1]; i<=w; i++)
5     next[i] = ((int)(i/weight[n-1])) * profit[n-1];
6
7   for(int currind=n-2; currind>=0; currind--)
8   {   vector<int> curr(w+1, 0);
9     for(int capacity=0; capacity<=w; capacity++)
10    { int notpick = next[capacity];
11      int pick=0;
12      if(weight[currind]<=capacity)
13        pick = profit[currind] + curr[capacity-weight[currind]];
14      curr[capacity]= max(pick,notpick);
15    }
16    next=curr;
17  }
18 return next[w];
19 }
```


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

Further Space Optimization: Using one 1D array

```
● ● ●  
1 int unboundedKnapsack(int n, int w, vector<int> &profit, vector<int>
2   &weight)
3 {  vector<int> next(w+1,0);
4   for(int i=weight[n-1];i<=w;i++)
5     next[i] = ((int)(i/weight[n-1])) * profit[n-1];
6
7   for(int currind=n-2;currind>=0;currind--)
8   {  for(int capacity=0;capacity<=w;capacity++)
9     { int notpick = next[capacity];
10    int pick=0;
11    if(weight[currind]<=capacity)
12      pick = profit[currind] + next[capacity-weight[currind]];
13    next[capacity]= max(pick,notpick);
14    }
15  }
16  return next[w];
17 }
```

[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
Mutuir Rehman khan | Level 1

Avg. time to solve

40 min

Success Rate

75%



Problem Statement

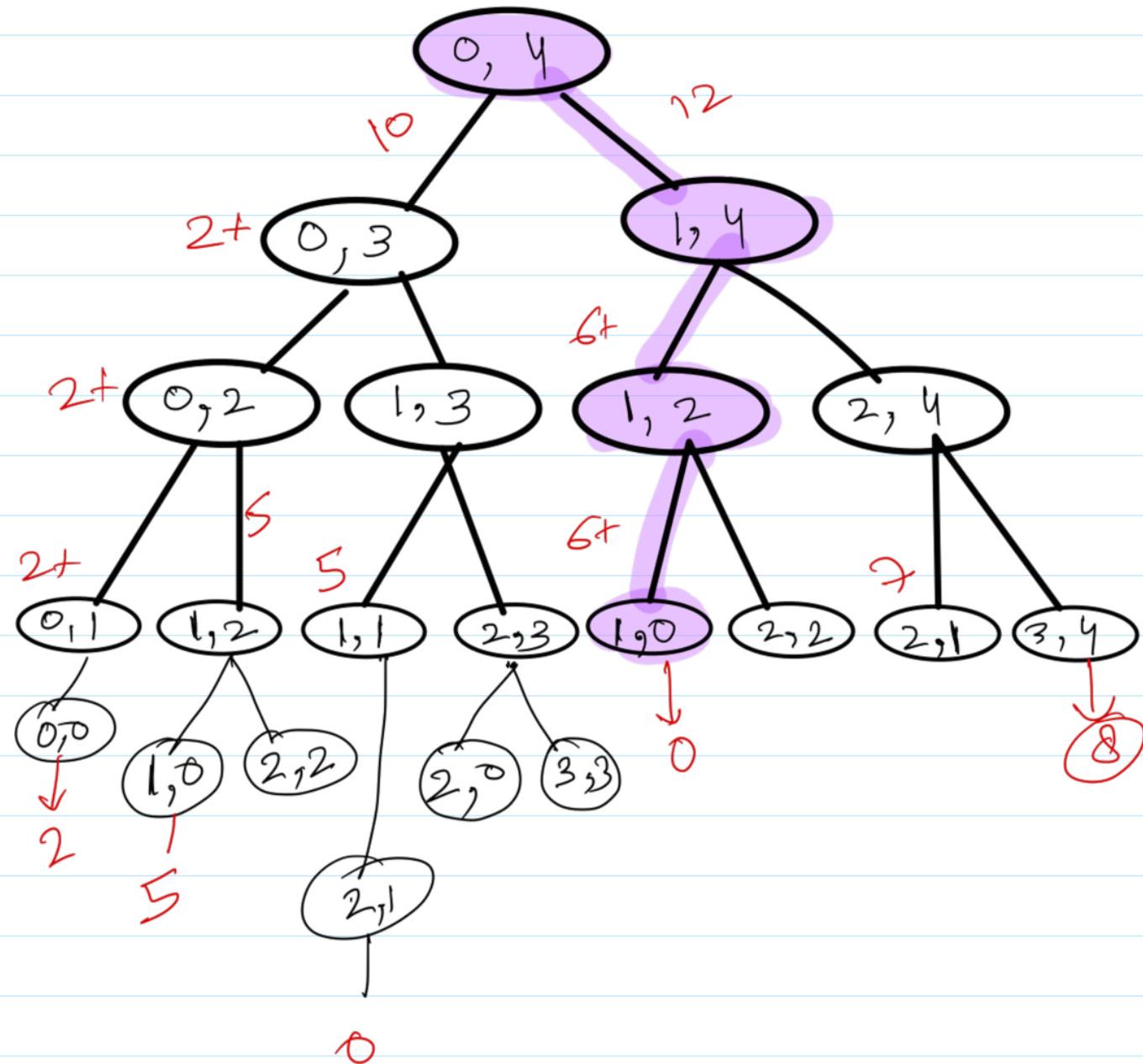
Suggest Edit

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:

```
● ● ●
```

```
1 int cutRod(int price[], int n) {  
2     //code here  
3     vector<vector<int>> v(n+1, vector<int> (n+1,-1));  
4     return maxprofit(price,0,n,n,v);  
5 }  
6  
7 int maxprofit(int price[], int currentindex, int n,int length,vector<vector<int>>& v )  
8 {  
9     if(currentindex>=n || length==0)  
10         return 0;  
11  
12     int currentlength= currentindex+1;  
13  
14     if(v[currentlength][length]!=-1)  
15         return v[currentlength][length];  
16  
17  
18     int consider=0;  
19     if(currentlength<=length)  
20         consider= price[currentindex] + maxprofit(price, currentindex,n, length-currentlength,v);  
21     int notconsider= maxprofit(price,currentindex+1,n,length,v);  
22  
23     return v[currentlength][length]=max(consider,notconsider);  
24 }  
25  
26 }
```

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

SC: O(N*N) + O(N) recursive stack space where the rod is divided into N parts

Tabulation:

```
● ● ●
```

```
1 int cutRod(vector<int> &price, int n)
2 { vector<vector<int>> dp(n,(vector<int> (n+1,0)));
3     for(int i = 0;i<=n;i++)
4         { dp[0][i] = i* price[0];
5         }
6
7     for(int currind = 1;currind<n;currind++)
8     { for(int N =0;N<=n;N++)
9         { int nottake = dp[currind-1][N];
10            int take = -100000000;
11            int rodlenth = currind+1;
12            if(rodlenth<=N){
13                take = price[currind] + dp[currind][N - rodlenth];
14            } dp[currind][N] = max(take,nottake);
15        }
16    }
17    return dp[n-1][n];
18 }
```

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

SC: O(N*N)

Space Optimization:

```
1 int cutRod(vector<int> &price, int n)
2 { vector<int> next(n+1,0);
3     for(int i = 0;i<=n;i++)
4         { next[i] = i* price[0];
5     }
6
7     for(int currind = 1;currind<n;currind++)
8     { vector<int> curr(n+1,0);
9         for(int N =0;N<=n;N++)
10            { int nottake = next[N];
11                int take = -100000000;
12                int rodlenth = currind+1;
13                if(rodlenth<=N){
14                    take = price[currind] + curr[N - rodlenth];
15                } curr[N] = max(take,nottake);
16            }
17            next = curr;
18        }
19    return next[n];
20 }
```

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

SC: O(N)

Further Space Optimization: Using one 1D array



```
1 int cutRod(vector<int> &price, int n)
2 { vector<int> next(n+1,0);
3   for(int i = 0;i<=n;i++)
4     { next[i] = i* price[0];
5     }
6
7   for(int currind = 1;currind<n;currind++)
8   {
9     for(int N =0;N<=n;N++)
10    { int nottake = next[N];
11      int take = -100000000;
12      int rodlength = currind+1;
13      if(rodlength<=N){
14        take = price[currind] + next[N - rodlength];
15      }   next[N] = max(take,nottake);
16    }
17  }
18 return next[n];
19 }
```

[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.

$\text{Abc} = \text{a, b, c, ab, bc, ac, abc, "}$

→ $\text{st1} \Rightarrow \text{a } \text{t} \text{ d}$ i
 $\text{st2} \Rightarrow \text{c } \text{b} \text{ d}$ j

} if ($\text{st1} == \text{st2}$)
action $f(i-1, j-1)$,

→ $\text{st1} \Rightarrow \text{c } \text{c}$ i
 $\text{st2} \Rightarrow \text{c } \text{e}$ j

or $\text{e } \text{c}$
 $c \text{ e } \text{j}$

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

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

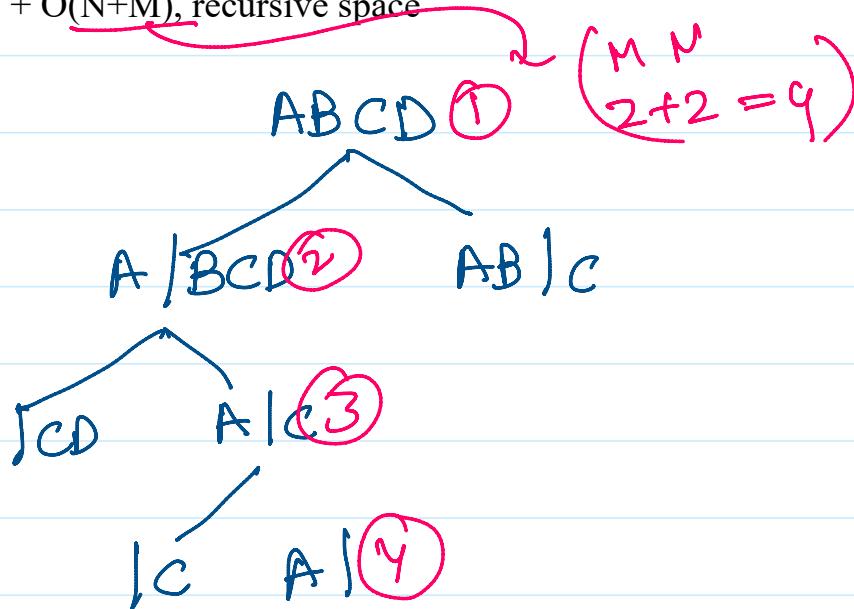
MEMOIZATION:

```

1 int findlcs(int ind1, int ind2, string s1, string s2, vector<vector<int>> &dp)
2     { if(ind1>s1.length()-1 || ind2>s2.length()-1)
3         return 0;
4         if(dp[ind1][ind2]!=-1)
5             return dp[ind1][ind2];
6         if(s1[ind1]==s2[ind2])
7             return dp[ind1][ind2]=1 + findlcs(ind1+1,ind2+1,s1,s2,dp);
8         if(s1[ind1]!=s2[ind2])
9             return dp[ind1][ind2]=max(findlcs(ind1+1, ind2,s1,s2,dp),
10                               findlcs(ind1,ind2+1,s1,s2,dp));
11     return 0;
12 }
13
14
15 int longestCommonSubsequence(string s1, string s2) {
16     int m = s1.length();
17     int n = s2.length();
18     vector<vector<int>> dp(m+1, vector<int>(n+1,-1));
19     return findlcs(0,0,s1,s2,dp);
20 }
```

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:

```
1 int longestCommonSubsequence(string s1, string s2) {  
2     int m = s1.length();  
3     int n = s2.length();  
4  
5     vector<vector<int>> dp(m+1, vector<int>(n+1, 0));  
6  
7     for(int ind1 = m-1; ind1 >= 0; ind1--)  
8     { for(int ind2 = n-1; ind2 >= 0; ind2--)  
9         { if(s1[ind1] == s2[ind2])  
10             dp[ind1][ind2] = 1 + dp[ind1+1][ind2+1];  
11         else  
12             dp[ind1][ind2] = max(dp[ind1+1][ind2], dp[ind1][ind2+1]);  
13         }  
14     }  
15     return dp[0][0];  
16 }
```

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:

```
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 }
```

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:

```
1 //print the longest common susbsequence
2 for(int i =0;i<m+1;i++)
3     { for(int j=0;j<n+1;j++)
4         { cout<<dp[i][j]<<" ";
5             if(dp[i][j]>dp[i+1][j] && dp[i][j]>dp[i][j+1])
6                 cout<<s2[j]<<" ";
7                 //or print s1[i];
8             }
9         cout<<endl;
10    }
```

Second Way:

	c	b	d	
q	2	2	1	0
i t t b	2	2	1	0
d)	1	1	0
	0	0	0	0

The diagram shows a 5x5 grid representing a dynamic programming table for the Longest Common Subsequence problem. The columns are labeled 'c', 'b', 'd' at the top, and the rows are labeled 'q', 'i', 't', 't', 'b' on the left. The values in the grid are: (q, c) = 2, (q, b) = 2, (q, d) = 1, (i, c) = 2, (i, b) = 2, (i, d) = 1, (t, c) = 1, (t, b) = 1, (t, d) = 1, (t, t) = 1, (b, c) = 0, (b, b) = 0, (b, d) = 0, (b, t) = 0, (b, b) = 0. Handwritten annotations include orange circles around the values 2, 2, 1, 1, and 0 in the bottom-right quadrant. A green circle highlights the value 1 in the (t, t) cell. Orange arrows point from the value 2 in the (i, b) cell to the (t, t) cell, and from the (t, t) cell to the value 0 in the (b, b) cell.



```
1  string ans = "";
2  int i=0;int j=0;
3  while(i<m && j<n)
4  {  if(s1[i]==s2[j])
5      {  ans+=s1[i];
6          i++;
7          j++;
8      }
9      else if(dp[i+1][j]>dp[i][j+1])
10         i++;
11     else j++;
12   }
13 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 Common Subsequence, 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, increment 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: abbbba.

If we only increment 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.

Str1 = b b b a
Str2 = a b b b b a
 $i=0, j=0,$
 $Str1[i] \neq Str2[j],$

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:

```

1 int findlcs(int ind1, int ind2,int countlcs, string s1, string s2,vector<vector<vector<int>>& dp)
2 {   if(ind1>s1.length()-1 || ind2>s2.length()-1)
3     return countlcs;
4
5     if (dp[ind1][ind2][countlcs]!= -1)
6       return dp[ind1][ind2][countlcs];
7     int match=countlcs;
8     if(s1[ind1]==s2[ind2])
9       match = findlcs(ind1+1,ind2+1,countlcs+1,s1,s2,dp);
10
11    int notmatch= max(findlcs(ind1,ind2+1,0,s1,s2,dp),findlcs(ind1+1,ind2,0,s1,s2,dp));
12    return dp[ind1][ind2][countlcs] =max(match,notmatch);
13 }
14
15 int lcs(string &s1, string &s2){
16     int m = s1.length();
17     int n = s2.length();
18     int countlcs=min(m,n);
19     vector<vector<vector<int>> dp(m+1,vector<vector<int>> (n+1,vector<int>(countlcs,-1)));
20     return findlcs(0,0,0,s1,s2,dp);
21 }
```

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:

```
1 int lcs(string &s1, string &s2){  
2     int m = s1.length();  
3     int n = s2.length();  
4  
5     vector<vector<int>> dp(m+1, vector<int>(n+1, 0));  
6     int ans = -10000;  
7     for(int ind1 = m-1; ind1 >= 0; ind1--)  
8     { for(int ind2 = n-1; ind2 >= 0; ind2--)  
9         { if(s1[ind1] == s2[ind2])  
10             dp[ind1][ind2] = 1 + dp[ind1+1][ind2+1];  
11             ans = max(ans, dp[ind1][ind2]);  
12         }  
13     }  
14     return ans;  
15 }
```

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

Space Optimization:

```
1 int lcs(string &s1, string &s2){  
2     int m = s1.length();  
3     int n = s2.length();  
4  
5     vector<int> next(n+1, 0);  
6     int ans = -10000;  
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            ans = max(ans, curr[ind2]);  
13        }  
14        next = curr;  
15    }  
16    return ans;  
17 }
```

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

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/kapilyadav22)

LECTURE - 28 LONGEST PALINDROMIC SUBSEQUENCE

23 June 2022 12:41

$Gx \rightarrow "bbqabbqabb" \rightarrow S1$

Take reverse of string
 $"bbqabbqabb" \rightarrow S2$

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

$\rightarrow bbqabbqabb. \approx bbaabb$

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

[LinkedIn/kapilyadav22](https://www.linkedin.com/in/kapilyadav22)

Congrats,
You Have
studied DP



 Kapilyadav22