Dynamic Programming

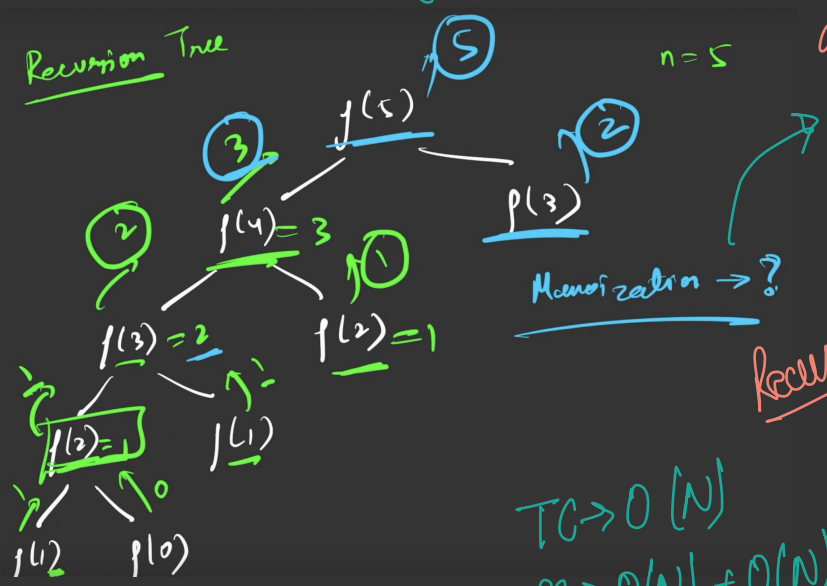
Pre-requisites → Recursion

① Tabulation → bottom up

② memoization → top down → tend to store the value of sub-problems in some map / table

Q.1 Fibonacci numbers

overlapping sub problems



Memoization → ?

dp = [-1, -1, -1, -1, -1, -1]
0 1 2 3 4 5

declare → dp[n+1]

Recursion

```
function fibonacci(n) {
    let dp = Array(n+1).fill(-1);
    return helper(n, dp);
}

function helper(n, dp) {
    if(n <= 1) return n;
    if(dp[n] != -1) return dp[n];
    return dp[n] = helper(n-1, dp) + helper(n-2, dp);
}
```

TC → O(N)

SC → O(N) + O(N)
↑ recursion ↑ array

tabulation (bottom-up)

base case to the required

① declare dp[n+1] (n+1 = size)

② base case
dp[0] = 0 → prev2
dp[1] = 1 → prev

③ recursion relationship
for(i=2; i<=n; i++)
{ dp[i] = dp[i-1] + dp[i-2];
}

TC → O(N)
SC → O(N)

④ space improvement
use two variables instead of array

prev = i-1
prev2 = i-2

curi = prev + prev2
prev2 = prev
prev = curi

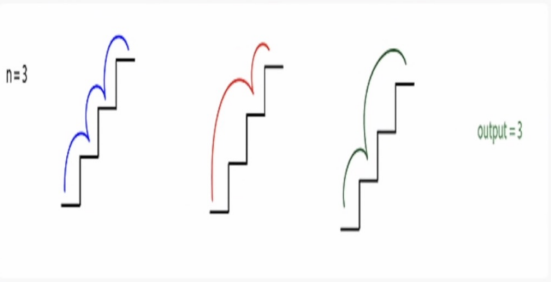
TC → O(N)
SC → O(1)

2.

You have been given a number of stairs. Initially, you are at the 0th stair, and you need to reach the Nth stair. Each time you can either climb one step or two steps. You are supposed to return the number of distinct ways in which you can climb from the 0th step to Nth step.

Example :

```
N=3
```




```
We can climb one step at a time i.e. {(0, 1) ,(1, 2),(2,3)} or we can climb the first two-step and then one step i.e. {(0,2),(1, 3)} or we can climb first one step and then two step i.e. {(0,1), (1,3)}.
```

1D problems

→ it's dp problems if

① count the total num of ways

② min/max output

try all possible ways comes

- count all ways
- best ways

shortcut

① try to represent the problem in terms of index

② do all possible stuffs on that index a/c to problem statement

③ sum of all stuffs → count all ways

min (of all stuffs) → find min

```
function totalSteps(n) {
    let dp = Array(n+1).fill(-1);
    return helper(n, dp);
}

function helper(n, dp) {
    if(n == 0 || n == 1) return 1;
    if(dp[n] != -1) return dp[n];
    return dp[n] = helper(n-1, dp) + helper(n-2, dp);
}
```

TC → O(N)

Q3. Frog jump

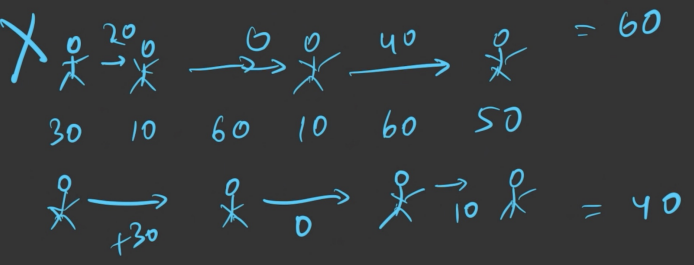
There is a frog on the 1st step of an N stairs long staircase. The frog wants to reach the Nth stair. HEIGHT[i] is the height of the (i+1)th stair.If Frog jumps from ith to jth stair, the energy lost in the jump is given by |HEIGHT[i-1] - HEIGHT[j-1] |.In the Frog is on ith staircase, he can jump either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

① why the greedy doesn't work here

using greedy you can miss out some significant jump in future



10 20 30 10

0 1 2 3

0 → 3

Recursion

① index based

② do all stuff on that index

③ take the min (all stuffs)

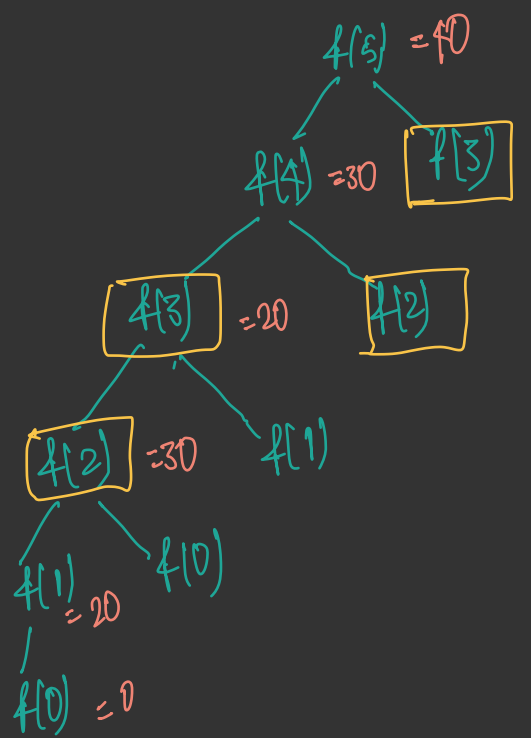
f(n-1) -> energy required to reach (n-1) from 0

```
f(index) {
    if(index == 0) return 0;
    left = f(index-1) + abs(a[index] - a[index-1])
    if(index > 1)
    right = f(index-2) + abs(a[index] - a[index-2])
    return min(left, right)
}
```

overlapping sub problems

0	1	2	3	4	5
30	10	60	10	60	50

n=6



convert this recursion to DP memoization

TC->O(N) , SC->O(N)

```
function frogJump(n, heights) {
    let dp = Array(n+1).fill(-1);
    return helper(n-1, heights, dp);
}

function helper(i, heights, dp) {
    if(i == 0) return 0;
    if(dp[i] !== -1) return dp[i];
    let left = helper(i-1, heights, dp) + Math.abs(heights[i] - heights[i-1]);
    let right = Number.MAX_SAFE_INTEGER;
    if(i > 1) right = helper(i-2, heights, dp) + Math.abs(heights[i] - heights[i-2]);
    return dp[i]=Math.min(left, right);
}
```

Tabulation

① declare dp = Array(n+1)

② fill the base case,
dp[0] = 0

③ for loop, i=1 to n-1

left = dp[i-1] + abs(a[i] - a[i-1])

if i>1
right = dp[i-2] + abs[a[i] - a[i-2])

dp[i] = min(left, right)

```
function frogJump(n, heights) {
    return helper(n-1, heights);
}

function helper(i, heights) {
    if(i == 0) return 0;
    let left = helper(i-1, heights) + Math.abs(heights[i] - heights[i-1]);
    let right = Number.MAX_SAFE_INTEGER;
    if(i > 1) right = helper(i-2, heights) + Math.abs(heights[i] - heights[i-2]);
    return Math.min(left, right);
}
```

Tabulation space optimized

TC->O(N)
SC->O(1)

```
function frogJump(n, heights) {
    let prev = 0;
    let prev2 = 0;
    for(let i=1; i<n; i++){
        let left = prev + Math.abs(heights[i] - heights[i-1]);
        let right = Number.MAX_SAFE_INTEGER;
        if(i>1) right = prev2 + Math.abs(heights[i] - heights[i-2]);
        let curr = Math.min(left, right);
        prev2 = prev;
        prev = curr;
    }
    return prev;
}
```

TC->O(N)
SC->O(N)

```
function frogJump(n, heights) {
    let dp = Array(n+1).fill(-1);
    dp[0] = 0;
    for(let i=1; i<n; i++){
        let left = dp[i-1] + Math.abs(heights[i] - heights[i-1]);
        let right = Number.MAX_SAFE_INTEGER;
        if(i>1) right = dp[i-2] + Math.abs(heights[i] - heights[i-2]);
        dp[i] = Math.min(left, right);
    }
    return dp[n-1];
}
```

follow-up

i → i+1
i → i+2

K jumps

i+1, i+2, i+3, ..., i+K

```javascript
function frogJump(n, heights, k) {
    let dp = Array(n+1).fill(-1);
    dp[0] = 0;
    for (let i = 1; i < n; i++) {
        let minSteps = Number.MAX_SAFE_INTEGER;
        for (let j = 1; j <= k; j++) {
            if (i - j >= 0) {
                let jump = dp[i-j] + Math.abs(heights[i] - heights[i - j]);
                minSteps = Math.min(minSteps, jump);
            }
        }
        dp[i] = minSteps
    }
    return dp[n-1];
}
```

TC → O(N*K)
SC → O(N)

we can space optimize from O(N) to O(K)

if K=4, 4 variables are needed to store last 4 values

if K=N, SC → O(N)
so optimization is not needed

Q4.

You are given an array/list of 'N' integers. You are supposed to return the maximum sum of the subsequence with the constraint that no two elements are adjacent in the given array/list.

Note:

A subsequence of an array/list is obtained by deleting some number of elements (can be zero) from the array/list, leaving the remaining elements in their original order.

Sample Input 1:

```
2
3
1 2 4
4
2 1 4 9
```

Sample Output 1:

```
5
11
```

Lect 687 of recursion

concept of pick & notpick

→ it means ind==1 is not picked

f(ind)

```
{ if (ind==0) return a[ind];
  if (ind<0) return 0;
  pick = a[ind] + f(ind-2);
  notpick = 0 + f(ind-1);
  return max(pick, notpick);
}
```

TC → O(2^n)

How to optimize recussion
if it has overlapping subproblems → memoize it

f(1) → sum kw 0 to index 1
f(2) → sum kw 0 to index 2

memoization

```
function maxSum(n, values) {
    let dp = Array(n).fill(-1);
    return helper(n-1, values, dp)
}

function helper(i, nums, dp) {
    if (i == 0) return nums[i];
    if (i < 0) return 0;
    if(dp[i] !== -1) return dp[i]

    let pick = nums[i] + helper(i - 2, nums, dp);
    let notPick = 0 + helper(i - 1, nums, dp);
    return dp[i] = Math.max(pick, notPick);
}
```

recursion

```
function maxSum(n, values) {
    return helper(n-1, values)
}

function helper(i, nums) {
    if (i == 0) return nums[i];
    if (i < 0) return 0;
    let pick = nums[i] + helper(i - 2, nums);
    let notPick = 0 + helper(i - 1, nums);
    return Math.max(pick, notPick);
}
```

TC->O(N)
SC->O(N)

```
function maxSum(n, values) {
    let dp = Array(n).fill(-1);
    dp[0] = values[0];
    for(let i=1; i<n; i++) {
        let pick = values[i] + (i>1 ? dp[i-2] : 0);
        let notPick = 0 + dp[i-1];
        dp[i] = Math.max(pick, notPick);
    }
    return dp[n-1];
}
```

TC->O(N)
SC->O(1)

```
function maxSum(n, values) {
    let prev = values[0];
    let prev2 = 0;
    for(let i=1; i<n; i++) {
        let pick = values[i] + prev2;
        let notPick = 0 + prev;
        curr = Math.max(pick, notPick);
        prev2 = prev;
        prev = curr;
    }
    return prev;
}
```

Q5.

Mr. X is a professional robber planning to rob houses along a street. Each house has a certain amount of money hidden. All houses along this street 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.

You are given an array/list of non-negative integers 'ARR' representing the amount of money of each house. Your task is to return the maximum amount of money Mr. X can rob tonight without alerting the police.

Note:

It is possible for Mr. X to rob the same amount of money by looting two different sets of houses. Just print the maximum possible robbed amount, irrespective of sets of houses robbed.

For Example:

(i) Given the input array arr[] = {2, 3, 2} the output will be 3 because Mr X cannot rob house 1 (money = 2) and then rob house 3 (money = 2), because they are adjacent houses. So, he'll rob only house 2 (money = 3)

Sample Input 1:

```
3
1
0
3
2 3 2
4
1 3 2 1
```

Sample Output 1:

```
0
3
4
```

here, first & last cannot be together since it's adjacent

```
function maxNonAdjacentSum(values) {
    let n = values.length;
    let prev = values[0];
    let prev2 = 0;
    for(let i=1; i<n; i++) {
        let pick = values[i] + prev2;
        let notPick = 0 + prev;
        curr = Math.max(pick, notPick);
        prev2 = prev;
        prev = curr;
    }
    return prev;
}

function houseRobber(valueInHouse) {
    const temp1 = [], temp2 = [];
    let n = valueInHouse.length;
    if(n==1) return valueInHouse[0];
    for(let i=0; i<n; i++) {
        if(i != 0) temp1.push(valueInHouse[i]);
        if(i != n-1) temp2.push(valueInHouse[i]);
    }
    return Math.max(maxNonAdjacentSum(temp1), maxNonAdjacentSum(temp2));
}
```

TC->O(N)
SC->O(N)

Q6.

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

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

For Example

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

10 50 1 → day 0

5 100 11 → day 1

greedy → 50 + 11 ⇒ 61

greedy fails here

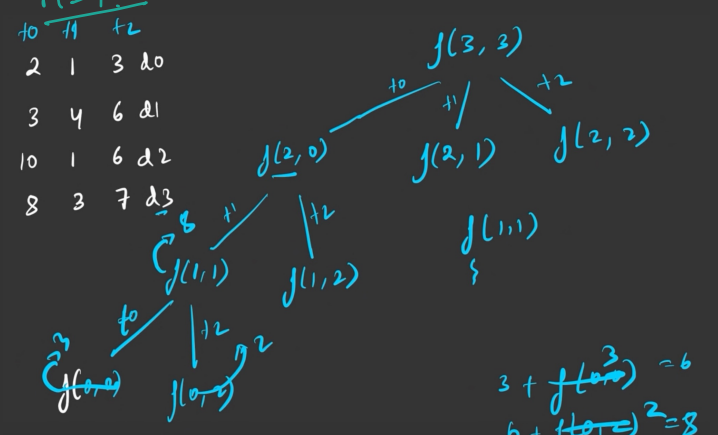
when greedy fails → try all possible ways → do recursion

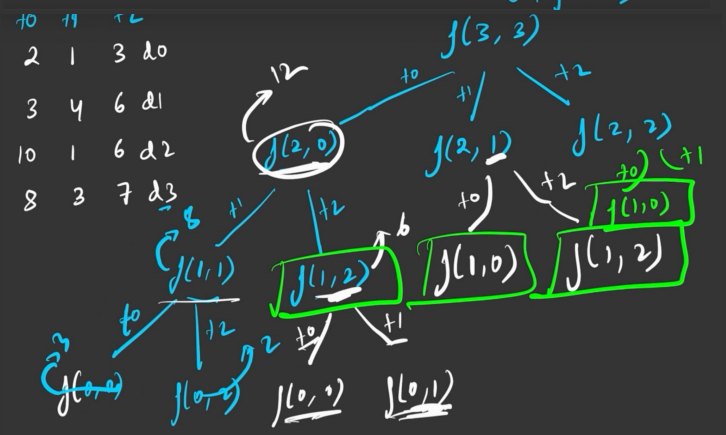
① index
② do stuff on the index
③ find max / all ways

Recursion

n = 4

to	t1	t2	
2	1	3	d0
3	4	6	d1
10	1	6	d2
8	3	7	d3





Overlapping subproblems

day → N

last → (0, 1, 2, 3)

dp[N][4] → -1

(N x 4) size dparray

TC → O(N x 4) x 3

SC → O(N) + O(N x 4)

```
function helper(day, last, points) {
    if(day === 0) {
        let max = 0;
        for (let task = 0; task < 3; task++) {
            if(task != last) {
                max = Math.max(max, points[0][task]);
            }
        }
        return max;
    }

    let max = 0;
    for(let task = 0; task < 3; task++) {
        if(task != last) {
            let point = points[day][task] + helper(day - 1, task, points);
            max = Math.max(max, point);
        }
    }
    return max;
}

function ninjaTraining(n, points) {
    return helper(n-1, 3, points)
}

let points = [[1,2,5], [3,1,1], [3,3,3]];
ninjaTraining(3, points); //?
```

to initialize the recursion number of task

```
function helper(day, last, points, dp) {
    if(day === 0) {
        let max = 0;
        for (let task = 0; task < 3; task++) {
            if(task != last) {
                max = Math.max(max, points[0][task]);
            }
        }
        return max;
    }

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

    let max = 0;
    for(let task = 0; task < 3; task++) {
        if(task != last) {
            let point = points[day][task] + helper(day - 1, task, points, dp);
            max = Math.max(max, point);
        }
    }
    return dp[day][last] = max;
}

function ninjaTraining(n, points) {
    let dp = Array(n).fill().map(() => Array(4).fill(-1));
    return helper(n-1, 3, points, dp)
}

let points = [[1,2,5], [3,1,1], [3,3,3]];
ninjaTraining(3, points); //?
```

Tabulation (bottom-Up)

① declare the similar size dp array

dp[n][4]

② base cases

dp[0][0] = max(a[0][1], a[0][2])

dp[0][1] = max(a[0][0], a[0][2])

dp[0][2] = max(a[0][0], a[0][1])

dp[0][3] = max(a[0][0], a[0][1], a[0][2])

③

④ return what you returned in the recussion (or, the initial call in recussion)

```
function ninjaTraining(n, points) {
    let dp = Array(n).fill().map(() => Array(4).fill(-1));

    dp[0][0] = Math.max(points[0][1], points[0][2])
    dp[0][1] = Math.max(points[0][0], points[0][2])
    dp[0][2] = Math.max(points[0][0], points[0][1])
    dp[0][3] = Math.max(points[0][1], points[0][1], points[0][2]);

    for(let day=1; day<n; day++) {
        for (let last = 0; last < 4; last++) {
            dp[day][last] = 0;
            for(let task = 0; task < 3; task++) {
                let point = points[day][task] + dp[day-1][task];
                dp[day][last] = Math.max(dp[day][last], point);
            }
        }
    }

    return dp[n-1][3];
}

let points = [[1,2,5], [3,1,1], [3,3,3]];
ninjaTraining(3, points); //?
```

TC -> O(N x 4 x 3)

SC -> O(N x 4)

space optimization

```
function ninjaTraining(n, points) {
    let prev = Array(4).fill(0);

    prev[0] = Math.max(points[0][1], points[0][2])
    prev[1] = Math.max(points[0][0], points[0][2]);
    prev[2] = Math.max(points[0][0], points[0][1]);
    prev[3] = Math.max(points[0][1], points[0][1], points[0][2]);

    for(let day=1; day<n; day++) {
        let temp = Array(4).fill(0);
        for (let last = 0; last < 4; last++) {
            temp[last] = 0;
            for(let task = 0; task < 3; task++) {
                temp[last] = Math.max(temp[last],points[day][task] + prev[task]);
            }
        }
        prev = temp;
    }

    return prev[3];
}

let points = [[1,2,5], [3,1,1], [3,3,3]];
ninjaTraining(3, points); //?
```

TC -> O(N x 4 x 3)

SC -> O(4) -> O(1)

L8. # DP on GRIDS / 2D matrix

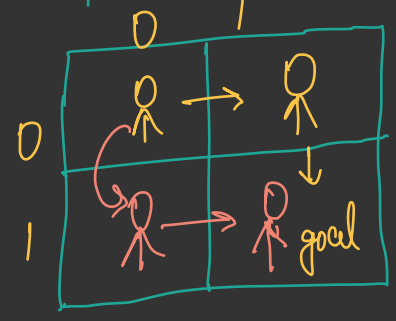
1. count paths
2. count paths with obstacles
3. min path sum
4. max path sum
5. triangle
6. 2 start points

Q7.

You are present at point 'A' which is the top-left cell of an M X N matrix, your destination is point 'B', which is the bottom-right cell of the same matrix. Your task is to find the total number of unique paths from point 'A' to point 'B'.In other words, you will be given the dimensions of the matrix as integers 'M' and 'N', your task is to find the total number of unique paths from the cell MATRIX[0][0] to MATRIX['M' - 1]['N' - 1].

To traverse in the matrix, you can either move Right or Down at each step. For example in a given point MATRIX[i] [j], you can move to either MATRIX[i + 1][j] or MATRIX[i][j + 1].

unique path



(0,0) → (m-1, n-1)

ways → 2

count all ways → use recursion

How to write recurrence

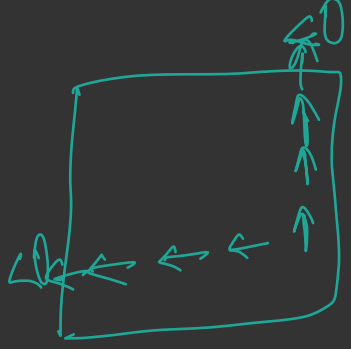
2D grid

① express everything in terms of index (row, column) i, j

② do all stuffs on the index

③ sum of all ways / max or min

```
f(i,j)
{
  if(i==0 && j==0)
    return 1;
  if(i<0 || j<0)
    return 0;
  up = f(i-1, j)
  left = f(i, j-1)
  return up+left;
}
```



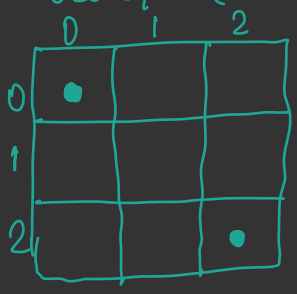
TC → $2^{m \times n}$

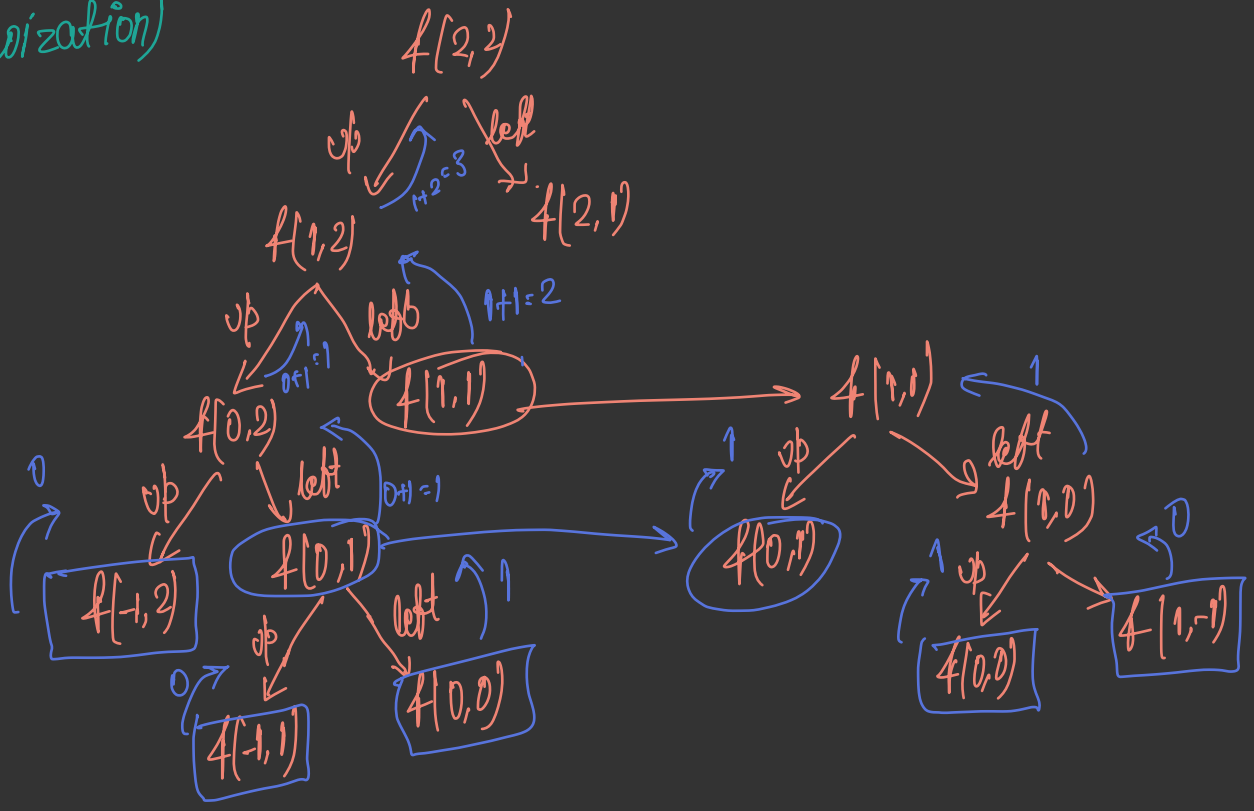
SC → O(path length)

(m-1) + (n-1)

optimize recursion

use DP (memoization)





DP

① declare dp[m][n], initialized with -1

② store the result, that you are returning from recursion call

③ check if answer is already calculated or not

TC → O(N×M)

SC → O[(N-1)+(M-1)] + O(N×M)
recursion DP

Tabulation (bottom-up)

① declare, dp[m][n]

② declare base case
dp[0][0] = 1

③ state all states in for loop

④ copy the recurrence & write

TC → O(M×N)

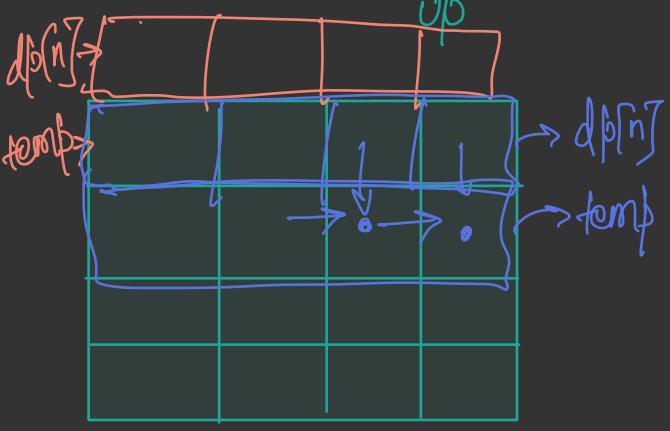
SC → O(M×N)

```
for( i=0 → m-1)
  for( j=0 → n-1)
  { if( i==0 && j==0) dp[0][0]=1
    else { up=0, left=0
    if(i>0) up = dp[i-1][j]
    if(j>0) left = dp[i][j-1]
      dp[i][j] = up + left
    }
  }
}
return dp[m-1][n-1]
```

Space optimization

→ if there is a prev row & prev column, we can space optimize it

$$dp[i][j] = \underbrace{dp[i-1][j]}_{up} + \underbrace{dp[i][j-1]}_{left}$$



TC → O(N×N)

SC → O(N)

```
dp[n]=0
for(i=0 → m-1)
{ for(j=0 → n-1)
  { if(i==0 && j==0) dp[i][j]=1;
    else {
      dp[i][j] = dp[i-1][j] + dp[i][j-1]
      temp[j] = dp[j] + temp[j-1]
    }
  } → dp = temp
}
```

temp[n]=0

Recursion

```
function f(i, j) {
    if(i == 0 && j == 0) return 1;
    if(i < 0 || j < 0) return 0;
    let up = f(i-1, j);
    let left = f(i, j-1);
    return up+left;
}

function uniquePaths(m, n) {
    return f(m-1, n-1)
}

uniquePaths(3,2); //? 3
```

Memoization

```
function f(i, j, dp) {
    if(i == 0 && j == 0) return 1;
    if(i < 0 || j < 0) return 0;
    if(dp[i][j] != -1) return dp[i][j];
    let up = f(i-1, j, dp);
    let left = f(i, j-1, dp);
    return dp[i][j] = up+left;
}

function uniquePaths(m, n) {
    let dp = Array(m).fill().map(() => Array(n).fill(-1));
    return f(m-1, n-1, dp)
}

uniquePaths(3,2); //? 3
```

Tabulation

```
function uniquePaths(m, n) {
    let dp = Array(m).fill().map(() => Array(n));
    for(let i=0; i<m; i++) {
        for(let j=0; j<n; j++) {
            if(i==0 && j==0) {
                dp[i][j]=1;
            } else {
                let up=0;
                let left=0;
                if(i>0) up = dp[i-1][j]
                if(j>0) left = dp[i][j-1]
                dp[i][j] = up+left;
            }
        }
    }
    return dp[m-1][n-1]
}

uniquePaths(3,2); //?
```

Space Optimization

```
function uniquePaths(m, n) {
    let prev = Array(n).fill(0);
    for(let i=0; i<m; i++) {
        let cur = Array(n).fill(0);
        for(let j=0; j<n; j++) {
            if(i==0 && j==0) {
                cur[j]=1;
            } else {
                let up=0;
                let left=0;
                if(i>0) up = prev[j]
                if(j>0) left = cur[j-1]
                cur[j] = up+left;
            }
        }
        prev = cur;
    }
    return prev[n-1]
}

uniquePaths(3,2); //?
```

08. Lec-10

Problem Statement Suggest Edit

Given a 'N' * 'M' maze with obstacles, count and return the number of unique paths to reach the right-bottom cell from the top-left cell. A cell in the given maze has a value '-1' if it is a blockage or dead-end, else 0. From a given cell, we are allowed to move to cells (i+1, j) and (i, j+1) only. Since the answer can be large, print it modulo 10^9 + 7.

For Example :

```
Consider the maze below :
0 0 0
0 -1 0
0 0 0

There are two ways to reach the bottom left corner -

(1, 1) -> (1, 2) -> (1, 3) -> (2, 3) -> (3, 3)
(1, 1) -> (2, 1) -> (3, 1) -> (3, 2) -> (3, 3)

Hence the answer for the above test case is 2.
```

```javascript
function f(i,j,mat) {
    if(i>=0 && j>=0 && mat[i][j]== -1) return 0;
    if(i == 0 && j == 0) return 1;
    if(i < 0 || j < 0) return 0;
    let up = f(i-1, j, mat);
    let left = f(i, j-1, mat);
    return up+left;
}

function mazeObstacles(m, n, mat) {
    return f(m-1,n-1, mat);
}

const mat = [[0,0,0],[0,-1,0],[0,0,0]]
mazeObstacles(3,3, mat); //? 2
```

```javascript
function f(i,j,mat,dp) {
    if(i>=0 && j>=0 && mat[i][j]== -1) return 0;
    if(i == 0 && j == 0) return 1;
    if(i < 0 || j < 0) return 0;
    if(dp[i][j] != -1) return dp[i][j];
    let up = f(i-1, j, mat, dp);
    let left = f(i, j-1, mat, dp);
    return dp[i][j]=up+left;
}

function mazeObstacles(m, n, mat) {
    let dp = Array(m).fill().map(() => Array(n).fill(-1));
    return f(m-1,n-1, mat, dp);
}

const mat = [[0,0,0],[0,-1,0],[0,0,0]]
mazeObstacles(3,3, mat); //? 2
```

```javascript
function mazeObstacles(m, n, mat) {
    let dp = Array(m).fill().map(() => Array(n));

    for(let i=0; i<m; i++){
        for(let j=0; j<n; j++){
            if(mat[i][j] == -1) {
                dp[i][j]=0;
            }
            else if(i==0 && j==0) {
                dp[i][j] = 1;
            } else {
                let up=0;
                let left=0;
                if(i>0) up = dp[i-1][j];
                if(j>0) left = dp[i][j-1];
                dp[i][j]=up+left;
            }
        }
    }

    return dp[m-1][n-1];
}

const mat = [[0,0,0],[0,-1,0],[0,0,0]]
mazeObstacles(3,3, mat); //?
```

```javascript
function mazeObstacles(m, n, mat) {
    let prev = Array(n).fill(0);

    for(let i=0; i<m; i++){
        let cur = Array(n).fill(0);
        for(let j=0; j<n; j++){
            if(mat[i][j] == -1) {
                cur[j]=0;
            }
            else if(i==0 && j==0) {
                cur[j] = 1;
            } else {
                let up=0;
                let left=0;
                if(i>0) up = prev[j];
                if(j>0) left = cur[j-1];
                cur[j]=up+left;
            }
        }
        prev = cur;
    }

    return prev[n-1];
}

const mat = [[0,0,0],[0,-1,0],[0,0,0]]
mazeObstacles(3,3, mat); //?
```

Q9. Lec11

Problem Statement Suggest Edit

Ninjaland is a country in the shape of a 2-Dimensional grid 'GRID', with 'N' rows and 'M' columns. Each point in the grid has some cost associated with it.
Find a path from top left i.e. (0, 0) to the bottom right i.e. ('N' - 1, 'M' - 1) which minimizes the sum of the cost of all the numbers along the path. You need to tell the minimum sum of that path.

Note:

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

5	9	6
11	5	2

For this the grid the path with minimum value is (0,0) -> (0,1) -> (1,1) -> (1,2). And the sum along this path is 5 + 9 +5 + 2 = 21. So the ans is 21.

In test case 2, The given grid is:

Base cases.

f(i,j) → this path summation is not considered

```
{ if(i==0 && j==0) return a[0][0];
  if(i<0 || j<0) return INT_MAX;
up = a[i][j] + f(i-1, j)
left = a[i][j] + f(i, j-1)
return min(up, left)
}
```

① compare (i,j)
② explore all paths
③ take the min path

Recursion

```
function f(i,j,mat) {
    if(i==0 && j==0) return mat[i][j];
    if(i<0 || j<0) return Number.MAX_SAFE_INTEGER;
    let up = mat[i][j] + f(i-1, j, mat);
    let left = mat[i][j] + f(i, j-1, mat);
    return Math.min(up, left);
}

function minPathSum(m, n, mat) {
    return f(m-1, n-1, mat);
}

const mat = [[5,9,6],[11,5,2]]
minPathSum(2,3, mat); //? 21
```

Memoization

TC → O(N×M)
SC → O(N×M) + O(path length)
(M-1)+(N-1)

```
function f(i,j,mat, dp) {
    if(i==0 && j==0) return mat[i][j];
    if(i<0 || j<0) return Number.MAX_SAFE_INTEGER;
    if(dp[i][j] != -1) return dp[i][j];
    let up = mat[i][j] + f(i-1, j, mat, dp);
    let left = mat[i][j] + f(i, j-1, mat, dp);
    return dp[i][j] = Math.min(up, left);
}

function minPathSum(m, n, mat) {
    let dp = Array(m).fill().map(() => Array(n).fill(-1));
    return f(m-1, n-1, mat, dp);
}

const mat = [[5,9,6],[11,5,2]]
minPathSum(2,3, mat); //? 21
```

Tabulation

```
function minPathSum(n, m, mat) {
    let dp = Array(n).fill().map(() => Array(m).fill(0));

    for(let i=0; i<n; i++){
        for(let j=0; j<m; j++){
            if(i==0 && j==0) {
                dp[i][j] = mat[i][j];
            } else {
                let up=mat[i][j];
                i>0 ? up += dp[i-1][j] : up += Number.MAX_SAFE_INTEGER;

                let left=mat[i][j];
                j>0 ? left += dp[i][j - 1] : left += Number.MAX_SAFE_INTEGER;

                dp[i][j]=Math.min(up, left);
            }
        }
    }

    return dp[n-1][m-1];
}

const mat = [[5,9,6],[11,5,2]]
minPathSum(2,3, mat); //?
```

Space optimization

```
function minPathSum(n, m, mat) {
    let prev = Array(m).fill(0);

    for(let i=0; i<n; i++){
        let cur = Array(m).fill(0);
        for(let j=0; j<m; j++){
            if(i==0 && j==0) {
                cur[j] = mat[i][j];
            } else {
                let up=mat[i][j];
                i>0 ? up += prev[j] : up += Number.MAX_SAFE_INTEGER;

                let left=mat[i][j];
                j>0 ? left += cur[j - 1] : left += Number.MAX_SAFE_INTEGER;

                cur[j]=Math.min(up, left);
            }
        }
        prev = cur;
    }

    return prev[m-1];
}

const mat = [[5,9,6],[11,5,2]]
minPathSum(2,3, mat); 21
```

DP Lec 12.

Triangle → Fixed starting point & variable ending point

2 3

3 6 7

8 9 6 10

try out all the paths

recursion → min or sum

① represent in (i,j)

② explore all paths (↓ ↘)

③ min of all paths

recursion will start from (0,0), (f(0,0))

f(0,0) → min path sum from (0,0) to last row(any column)

TC → $2^{1+2+3+\ldots+N}$ ← num of cols

SC → O(N)

```
f(i,j)
{ if (i==n-1) return a[n-i][j];
  if (dp[i][j]!=-1) return dp[i][j]
  down = a[i][j] + f(i+1,j);
  diagonal = a[i][j] + f(i+1,j+1);
  return min(down, diagonal)
                ↑
              dp[i][j]=
}
```

memoization

dp[n][n] TC → O(N×N)

SC → O(N×N) + O(N)

Tabulation

declare dp[n][n]

```
for(j=0; j<n; j++)
    dp[n-1][j] = a[n-1][j]

for(i=n-2; i>=0; i--)
  {  for(j=i; j>=0; j--)
     {  d = a[i][j] + dp[i+1][j]
        dg = a[i][j] + dp[i+1][j+1]
        dp[i][j] = min(d,dg)
     }
  }
return dp[0][0]
```

down ← d

diagonal ← dg

TC → O(N×N)

SC → O(N×N)

Space optimisation

1

2 3

3 6 7

8 9 6 10

→ we need only prev row & current row

Problem Statement Suggest Edit

You are given a triangular array/list 'TRIANGLE'. Your task is to return the minimum path sum to reach from the top to the bottom row.
The triangle array will have N rows and the i-th row, where 0 <= i < N will have i + 1 elements.
You can move only to the adjacent number of row below each step. For example, if you are at index j in row i, then you can move to i or i + 1 index in row j + 1 in each step.

For Example :

```
If the array given is 'TRIANGLE' = [[1], [2,3], [3,6,7],
[8,9,6,1]] the triangle array will look like:

1
2,3
3,6,7
8,9,6,10

For the given triangle array the minimum sum path would be
1->2->3->8. Hence the answer would be 14.
```

Recursion

```
function f(i,j,triangle, n) {
    if(i == n-1) {
        return triangle[n-1][j];
    }
    let d = triangle[i][j] + f(i+1, j, triangle, n);
    let dg = triangle[i][j] + f(i+1, j+1, triangle, n);
    return Math.min(d, dg);
}

function minimumPathSum(triangle, n) {
    return f(0,0, triangle, n);
}

const triangle = [[1], [2,3], [3,6,7], [8,9,6,10]];
minimumPathSum(triangle, 4) //? 14
```

TLE

memoization

```
function f(i,j,triangle, n, dp) {
    if(i == n-1) {
        return triangle[n-1][j];
    }
    if(dp[i][j] != -1) return dp[i][j];
    let d = triangle[i][j] + f(i+1, j, triangle, n, dp);
    let dg = triangle[i][j] + f(i+1, j+1, triangle, n, dp);
    return dp[i][j] = Math.min(d, dg);
}

function minimumPathSum(triangle, n) {
    const dp = Array(n).fill().map(() => Array(n).fill(-1));
    return f(0,0, triangle, n, dp);
}

const triangle = [[1], [2,3], [3,6,7], [8,9,6,10]];
minimumPathSum(triangle, 4) //?
```

TLE

```javascript
function minimumPathSum(triangle, n) {
    const dp = Array(n).fill().map(() => Array(n).fill(0));

    for(let j=0; j<n; j++) dp[n-1][j] = triangle[n-1][j];

    for(let i=n-2; i>=0; i--) {
        for(let j=i; j>=0; j--) {
            let d = triangle[i][j] + dp[i+1][j];
            let dg = triangle[i][j] + dp[i+1][j+1];
            dp[i][j] = Math.min(d, dg)
        }
    }

    return dp[0][0];
}

const triangle = [[1], [2,3], [3,6,7], [8,9,6,10]];
minimumPathSum(triangle, 4) //?
```

Tabulization

```javascript
function minimumPathSum(triangle, n) {
    let front = Array(n).fill(0);
    let cur = Array(n).fill(0);

    for(let j=0; j<n; j++) front[j] = triangle[n-1][j];

    for(let i=n-2; i>=0; i--) {
        for(let j=i; j>=0; j--) {
            let d = triangle[i][j] + front[j];
            let dg = triangle[i][j] + front[j+1];
            cur[j] = Math.min(d, dg)
        }
        front = cur;
    }

    return front[0];
}

const triangle = [[1], [2,3], [3,6,7], [8,9,6,10]];
minimumPathSum(triangle, 4) //?
```

Space optimization

Lec 13 Min/Max Falling path sum / variable starting & ending points.

Maximum Path Sum in the matrix

Contributed by **Pankaj Sharma**
Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 35 mins | Success Rate 70 %

Share | 117 upvotes

Problem Statement

Suggest Edit

You have been given an N*M matrix filled with integer numbers, find the maximum sum that can be obtained from a path starting from any cell in the first row to any cell in the last row.
From a cell in a row, you can move to another cell directly below that row, or diagonally below left or right. So from a particular cell (row, col), we can move in three directions i.e.

```
Down: (row+1,col)
Down left diagonal: (row+1,col-1)
Down right diagonal: (row+1, col+1)
```

Explanation Of Input 1 :

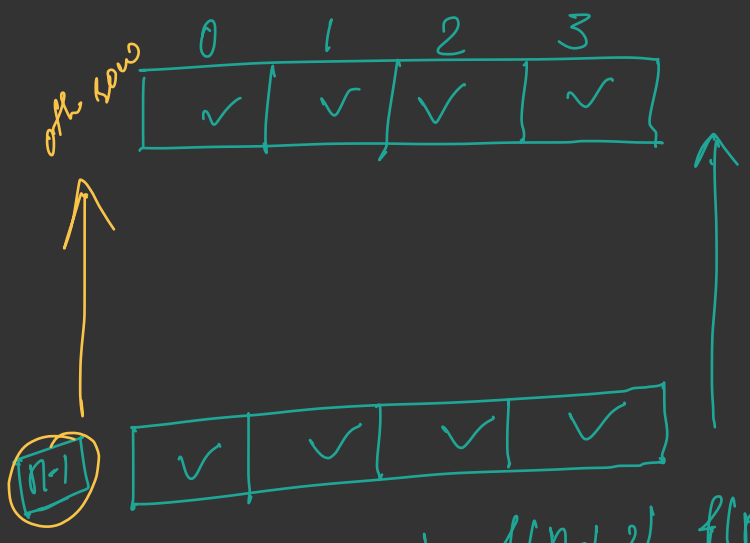
```
In the first test case for the given matrix,
```

1	2	10	4
100	3	2	1
1	1	20	2
1	2	2	1

```
The maximum path sum will be 2->100->1->2, So the sum is
105(2+100+1+2).

In the second test case for the given matrix, the maximum
path sum will be 10->7->8, So the sum is 25(10+7+8).
```

Q. max path sum
from any cell → 1st row
to
any cell → last row

(M1) → try out all the paths
recursion
① express everything in terms of (i,j)
& write base case
② explore all the paths ↙↓↘
③ max among all paths



f(n-1, 0) f(n-1, 1) f(n-1, 2) f(n-1, 3)

```
for (j=0 → m-1)
  { f(n-1, j);
  }
```

f(i,j) → max path sum to reach (i,j) from any cell in the first row

recurrence

base case → destination
→ out of bound cases

col < 0 or col ≥ m

```
f(i,j)
{
  if(j<0 || j>=m) return Numbes.. Max.Negative;
  if (i==0) return a[0][i];
  let s = a[i][i] + f(i-1, j)
  let ld = a[i][j] + f(i-1, j-1);
  let rd = a[i][j] + f(i-1, j+1);
  return Math.max(s,ld,sd);
}
```

base case

TLE TC → 3^N exponential
SC → O(N)

Memoization

recursion → overlapping sub problems → memoization

```
f(i,j) declare dp[N][M]
{
  if(j<0 || j>=m) return Numbes.. Max.Negative;
  if (i==0) return a[0][i];
→ if(dp[i][j] != -1) return dp[i][j];
  let s = a[i][i] + f(i-1, j)
  let ld = a[i][j] + f(i-1, j-1);
  let rd = a[i][j] + f(i-1, j+1);
  return dp[i][j] = Math.max(s,ld,sd);
}
```

TC → O(N×M)
SC → O(N×M) + O(N)

Tabulation → (bottom up)

1. declare dp array
2. write base cases
3. observe (i,j) & convert to loops

```
dp[N][M]
for (j=0 → m-1) dp[0][j] = a[0][j]
for (i=1 → n-1)
{ for (j=0 → m-1)
  { u = a[i][j] + dp[i-1][j]
if(j-1>0)  ld = a[i][j] + dp[i-1][j-1]
if(j+1<m)  rd = a[i][j] + dp[i-1][j+1]
    dp[i][j] = Max(u, ld, rd);
  }
}
maxi = dp[n-1][0]
for(j=1 → m-1)
  maxi = max(maxi, dp[n-1][j])
return maxi;
```

TC → O(N×M) + O(M)

SC → O(N×M)

space optimization

prev[n] , cur[n]

for (j=0 → m-1) dp[0][j] = a[0][j]

for (i=1 → n-1)

{ for (j=0 → m-1)

{ u = a[i][j] + dp[i-1][j]

if(j-1>0) ld = a[i][j] + dp[i-1][j-1]

if(j+1<m) rd = a[i][j] + dp[i-1][j+1]

dp[i][j] = max(u, ld, rd);

cur

} → prev = cur

}

maxi = dp[n-1][0]

for(j=1 → m-1)

maxi = max(maxi, dp[n-1][j])

return maxi;

TC -> O(N×M) + O(M)

SC -> O(N)

```
function f(i, j, matrix) {
    if(j < 0 || j >= matrix[0].length) return Number.MIN_SAFE_INTEGER;
    if(i == 0) return matrix[0][j];

    let u = matrix[i][j] + f(i-1, j, matrix);
    let ld = matrix[i][j] + f(i-1, j-1, matrix);
    let rd = matrix[i][j] + f(i-1, j+1, matrix);

    return Math.max(u, ld, rd);
}

function getMaxPathSum(matrix) {
    let n = matrix.length;
    let m = matrix[0].length;
    let maxi = Number.MIN_SAFE_INTEGER;

    for(let j=0; j<m; j++) {
        maxi = Math.max(maxi, f(n-1, j, matrix));
    };

    return maxi;
}

const matrix = [
    [1, 2, 10, 4],
    [100, 3, 2, 1],
    [1, 1, 20, 2],
    [1, 2, 2, 1],
];
getMaxPathSum(matrix); //? 105
```

Recursion

```
function f(i, j, matrix, dp) {
    if(j < 0 || j >= matrix[0].length) return Number.MIN_SAFE_INTEGER;
    if(i == 0) return matrix[0][j];
    if(dp[i][j] != -1) return dp[i][j];

    let u = matrix[i][j] + f(i-1, j, matrix, dp);
    let ld = matrix[i][j] + f(i-1, j-1, matrix, dp);
    let rd = matrix[i][j] + f(i-1, j+1, matrix, dp);

    return dp[i][j] = Math.max(u, ld, rd);
}

function getMaxPathSum(matrix) {
    let n = matrix.length;
    let m = matrix[0].length;
    let dp = Array(n).fill().map(() => Array(m).fill(-1));
    let maxi = Number.MIN_SAFE_INTEGER;

    for(let j=0; j<m; j++) {
        maxi = Math.max(maxi, f(n-1, j, matrix, dp));
    };

    return maxi;
}

const matrix = [
    [1, 2, 10, 4],
    [100, 3, 2, 1],
    [1, 1, 20, 2],
    [1, 2, 2, 1],
];
getMaxPathSum(matrix); //? 105
```

memoization

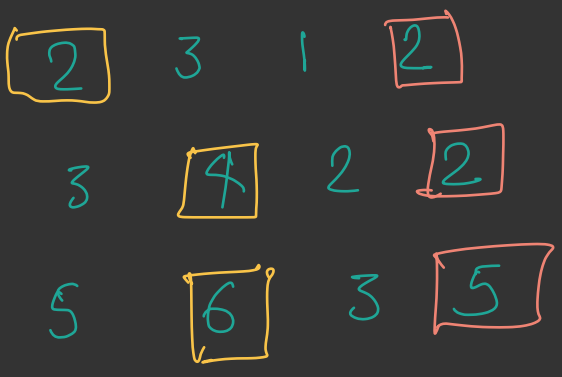
```javascript
function getMaxPathSum(matrix) {
    let n = matrix.length;
    let m = matrix[0].length;
    let dp = Array(n).fill().map(() => Array(m).fill(0));
    let maxi = Number.MIN_SAFE_INTEGER;

    for(let j=0; j<m; j++) dp[0][j] = matrix[0][j];

    for(let i=1; i<n; i++) {
        for(let j=0; j<m; j++) {
            let u = matrix[i][j] + dp[i - 1][j];
            let ld = j-1>= 0 ?  matrix[i][j] + dp[i - 1][j-1] : Number.MIN_SAFE_INTEGER;
            let rd = j+1 < m ? matrix[i][j] + dp[i - 1][j+1] : Number.MIN_SAFE_INTEGER;
            dp[i][j] = Math.max(u, ld, rd);
        }
    }

    for(let j=0; j<m; j++) {
        maxi = Math.max(maxi, dp[n-1][j]);
    };

    return maxi;
}

const matrix = [
    [1, 2, 10, 4],
    [100, 3, 2, 1],
    [1, 1, 20, 2],
    [1, 2, 2, 1],
];
getMaxPathSum(matrix); //?
```

Tabulation

```javascript
function getMaxPathSum(matrix) {
    let n = matrix.length;
    let m = matrix[0].length;
    let prev = Array(m).fill(0);
    let cur = Array(m).fill(0);
    let maxi = Number.MIN_SAFE_INTEGER;

    for(let j=0; j<m; j++) prev[j] = matrix[0][j];

    for(let i=1; i<n; i++) {
        for(let j=0; j<m; j++) {
            let u = matrix[i][j] + prev[j];
            let ld = j-1>= 0 ?  matrix[i][j] + prev[j-1] : Number.MIN_SAFE_INTEGER;
            let rd = j+1 < m ? matrix[i][j] + prev[j+1] : Number.MIN_SAFE_INTEGER;
            cur[j] = Math.max(u, ld, rd);
        }
        prev = [...cur];
    }

    for(let j=0; j<m; j++) {
        maxi = Math.max(maxi, prev[j]);
    };

    return maxi;
}

const matrix = [
    [1, 2, 10, 4],
    [100, 3, 2, 1],
    [1, 1, 20, 2],
    [1, 2, 2, 1],
];
getMaxPathSum(matrix); //?
```

space optimization

lec 14 → chessy pick up, DP on GRIDS, 3D DP



12 + 9 ⇒ 21 ans

fixed starting point → variable ending point

Max (All paths by Alice + All paths by Bob)
↓ recursion ↓ recursion

① express everything in terms (i1, j1) & (i2, j2) → destination → out of bound
& write base cases

② explore all possible paths ↙ ↓ ↘

③ Max sum

f(0, 0, 0, m-1)
alice bob

Problem Statement Suggest Edit

Ninja has a 'GRID' of size 'R' X 'C'. Each cell of the grid contains some chocolates. Ninja has two friends Alice and Bob, and he wants to collect as many chocolates as possible with the help of his friends.

Initially, Alice is in the top-left position i.e. (0, 0), and Bob is in the top-right place i.e. (0, 'C' - 1) in the grid. Each of them can move from their current cell to the cells just below them. When anyone passes from any cell, he will pick all chocolates in it, and then the number of chocolates in that cell will become zero. If both stay in the same cell, only one of them will pick the chocolates in it.

If Alice or Bob is at (i, j) then they can move to (i + 1, j), (i + 1, j - 1) or (i + 1, j + 1). They will always stay inside the 'GRID'.

Your task is to find the maximum number of chocolates Ninja can collect with the help of his friends by following the above rules.

Example:

```
Input: 'R' = 3, 'C' = 4
       'GRID' = [[2, 3, 1, 2], [3, 4, 2, 2], [5, 6, 3, 5]]
Output: 21

Initially Alice is at the position (0,0) he can follow the
path (0,0) -> (1,1) -> (2,1) and will collect 2 + 4 + 6 = 12
chocolates.

Initially Bob is at the position (0, 3) and he can follow the
path (0, 3) -> (1,3) -> (2, 3) and will colllect 2 + 2 + 5 =
9 chocolates.

Hence the total number of chocolates collected will be 12 + 9
= 21. there is no other possible way to collect a greater
number of chocolates than 21.
```

```
f(i1, j1, i2, j2)
{
  if (j1 < 0 || j1 >= m || j2 < 0 || j2 > m)
        return Negative Min;
  if (i == n-1)
    { if (j1 == j2) return a[i][j1];
      else return a[i][j1] + a[i][j2];
    }
  maxi = 0
  for (dj1 -> -1 to +1)
   { for (dj2 -> -1 to +1)
      {
        if (j1 == j2) maxi = max(maxi, a[i][j1] + f(i+1, j1+dj1, j2+dj2))
        else maxi = max(maxi, a[i][j1] + a[i][j2] + f(i+1, j1+dj1, j2+dj2))
      }
   }
  return maxi;
}
```

(i+1, j-1) (i+1, j) (i+1, j+1)

Alice × Bob
3 × 3 ⇒ 9 combos of path

if (dp[i][j1][j2] != -1) return dp[i][j1][j2]

return dp[i][j1][j2] = maxi;

TC → $(3^N \times 3^N)$ → exponential

SC → O(N)

memoization

(i) (j1) (j2)

N M M

dp → [N] × [M] × [M]

dp[i][j1][j2]

TC → O(N × M × M) × 9 → number of different states

SC → O(N × M × M) + O(N)

↑ recursion

```javascript
function f(i, j1, j2, r, c, a) {
    if (j1 < 0 || j2 < 0 || j1 >= c || j2 >= c) return Number.MIN_SAFE_INTEGER;

    if(i == r-1) {
        return (j1 == j2) ? a[i][j1] : a[i][j1] + a[i][j2];
    }

    let maxi = 0;
    for(let dj1 = -1; dj1 <= 1; dj1++) {
        for(let dj2 = -1; dj2 <= 1; dj2++) {
            let value = 0;
            value = (j1 == j2) ? a[i][j1] : a[i][j1] + a[i][j2];
            value += f(i+1, j1+dj1, j2+dj2, r, c, a)
            maxi = Math.max(maxi, value);
        }
    }
    return maxi;
}

function maximumChocolates(r, c, grid) {
    return f(0, 0, c - 1, r, c, grid);
}

const grid = [
    [2, 3, 1, 2],
    [3, 4, 2, 2],
    [5, 6, 3, 5],
];
maximumChocolates(3, 4, grid); //? 21
```

Recursion

```javascript
function f(i, j1, j2, r, c, a, dp) {
    if (j1 < 0 || j2 < 0 || j1 >= c || j2 >= c) return Number.MIN_SAFE_INTEGER;

    if(dp[i][j1][j2] != -1) return dp[i][j1][j2];

    if(i == r-1) {
        return (j1 == j2) ? a[i][j1] : a[i][j1] + a[i][j2];
    }

    let maxi = 0;
    for(let dj1 = -1; dj1 <= 1; dj1++) {
        for(let dj2 = -1; dj2 <= 1; dj2++) {
            let value = 0;
            value = (j1 == j2) ? a[i][j1] : a[i][j1] + a[i][j2];
            value += f(i+1, j1+dj1, j2+dj2, r, c, a, dp)
            maxi = Math.max(maxi, value);
        }
    }
    return dp[i][j1][j2] = maxi;
}

function maximumChocolates(r, c, grid) {
    let dp = Array(r).fill().map(() => Array(c).fill().map(() => Array(c).fill(-1)));
    return f(0, 0, c - 1, r, c, grid, dp);
}

const grid = [
    [2, 3, 1, 2],
    [3, 4, 2, 2],
    [5, 6, 3, 5],
];
maximumChocolates(3, 4, grid); //? 21
```

Memoization

Tabulation

dp[n][m][m]

for(j1 -> 0 to m-1)
{ for(j2 -> 0 to m-1)
{ if(j1 == j2) dp[n-1][j1][j2] = grid[n-1][j1];
else dp[n-1][j1][j2] = grid[n-1][j1] + grid[n-1][j2];
}
}

base case

for(i = n-2 to 0)
{ for(j1 = 0 to m-1)
{ for(j2 = 0 to m-1)
{
}
}
}

```javascript
function maximumChocolates(n, m, grid) {
    let dp = Array(n)
        .fill()
        .map(() =>
            Array(m)
                .fill()
                .map(() => Array(m).fill(0))
        );

    for (let j1 = 0; j1 < m; j1++) {
        for (let j2 = 0; j2 < m; j2++) {
            dp[n - 1][j1][j2] =
                j1 == j2 ? grid[n - 1][j1] : grid[n - 1][j1] + grid[n - 1][j2];
        }
    }

    for (let i = n - 2; i >= 0; i--) {
        for (let j1 = 0; j1 < m; j1++) {
            for (let j2 = 0; j2 < m; j2++) {
                let maxi = 0;
                for (let dj1 = -1; dj1 <= 1; dj1++) {
                    for (let dj2 = -1; dj2 <= 1; dj2++) {
                        let value = 0;
                        value = j1 == j2 ? grid[i][j1] : grid[i][j1] + grid[i][j2];
                        if(j1+dj1 >= 0 && j1+dj1 < m && j2+dj2 >= 0 && j2+dj2 < m) {
                            value += dp[i + 1][j1 + dj1][j2 + dj2]
                        } else {
                            value = Number.MIN_SAFE_INTEGER;
                        }
                        maxi = Math.max(maxi, value);
                    }
                }
                dp[i][j1][j2] = maxi;
            }
        }
    }

    return dp[0][0][m-1];
}

const grid = [
    [2, 3, 1, 2],
    [3, 4, 2, 2],
    [5, 6, 3, 5],
];
maximumChocolates(3, 4, grid); //? 21
```

Space optimization

1D dp → two variables
2D dp → 1D dp
3D dp → 2D dp

```javascript
function maximumChocolates(n, m, grid) {
    let front = Array(m).fill().map(() => Array(m).fill(0))
    let cur = Array(m).fill().map(() => Array(m).fill(0))

    for (let j1 = 0; j1 < m; j1++) {
        for (let j2 = 0; j2 < m; j2++) {
            front[j1][j2] =
                j1 == j2 ? grid[n - 1][j1] : grid[n - 1][j1] + grid[n - 1][j2];
        }
    }

    for (let i = n - 2; i >= 0; i--) {
        for (let j1 = 0; j1 < m; j1++) {
            for (let j2 = 0; j2 < m; j2++) {
                let maxi = 0;
                for (let dj1 = -1; dj1 <= 1; dj1++) {
                    for (let dj2 = -1; dj2 <= 1; dj2++) {
                        let value = 0;
                        value = j1 == j2 ? grid[i][j1] : grid[i][j1] + grid[i][j2];
                        if(j1+dj1 >= 0 && j1+dj1 < m && j2+dj2 >= 0 && j2+dj2 < m) {
                            value += front[j1 + dj1][j2 + dj2]
                        } else {
                            value = Number.MIN_SAFE_INTEGER;
                        }
                        maxi = Math.max(maxi, value);
                    }
                }
                cur[j1][j2] = maxi;
            }
        }
        front = cur;
        cur = Array(m)
            .fill()
            .map(() => Array(m).fill(0));
    }

    return front[0][m-1];
}

const grid = [
    [2, 3, 1, 2],
    [3, 4, 2, 2],
    [5, 6, 3, 5],
];
maximumChocolates(3, 4, grid); //?
```

lec 15 subset sum equals to target / identify DP on subsequences

subsequences / subset

contagious / non-contagious

[1, 3, 2] → [1, 2]
[2, 1]
[3, 2]

	subsequence/subarray	subset
[1, 2]	✓	✓
[2, 1]	✗	✓
[3, 2]	✓	✓

Subset Sum Equal To K

Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 30 mins | Success Rate 65 %

Share | 220 upvotes

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.
```

→ generate all subsequences & check if any of them gives a sum of K

M1 → power set

M2 → recursion

Recursion

① express everything in (index, target)

② explore possibilities of that index

- a[index] part of the subsequence
- a[index] not part of the subsequence

③ return true/false

```
f(ind, target)
{
  if(target == 0) return true;
  if(ind == 0) return (a[0] == target);
bool notTake = f(ind-1, target)
bool take = false
  if (target >= a[ind])
      take = f(ind-1, target - a[ind]);
  return take || notTake;
}
```

TC → $O(2^N)$

SC → $O(N)$

Memoization

① figure out the changing state

ind <= 10^3 , target <= 10^5

ind target

dp [10^3+1] x [10^5+1]

TC -> O(N x target)

SC -> O(N x target) + O(N)

```
f(ind, target)
{
  if(target == 0) return true;
  if(ind == 0) return (a[0] == target);
  if(dp[ind][target] != -1) return dp[ind][target]
bool notTake = f(ind-1, target)
bool take = false
  if (target >= a[ind])
      take = f(ind-1, target - a[ind]);
  return dp[ind][target] = take || notTake;
}
```

Tabulation

① dp[N][target]

② base cases

for(i=0 -> n-1) dp[i][0] = true

dp[0][a[0]] = true

② explore

```
for(i=0 -> n-1)
{  for(target=1 -> k)
   {

   }
}
```

```javascript
function f(ind, target, arr) {
    if(target == 0) return true;
    if(ind == 0) return arr[0] === target;

    let notTake = f(ind-1, target, arr);
    let take = false;
    if(arr[ind] <= target) take = f(ind-1, target-arr[ind], arr);

    return Boolean(take || notTake);
}

function subsetSumToK(n, k, arr) {
    return f(n-1, k, arr);
}

const arr = [4, 3, 2,1]
subsetSumToK(4, 1, arr); //?true
```

recursion

```javascript
function f(ind, target, arr, dp) {
    if(target == 0) return true;
    if(ind == 0) return arr[0] === target;

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

    let notTake = f(ind-1, target, arr, dp);
    let take = false;
    if(arr[ind] <= target) take = f(ind-1, target-arr[ind], arr, dp);

    return dp[ind][target] = Boolean(take || notTake);
}

function subsetSumToK(n, k, arr) {
    let dp = Array(n).fill().map(() => Array(k+1).fill(-1));
    return f(n-1, k, arr, dp);
}

const arr = [4, 3, 2,1]
subsetSumToK(4, 4, arr); //?true
```

memoization

```javascript
function subsetSumToK(n, k, arr) {
    let dp = Array(n)
        .fill()
        .map(() => Array(k + 1).fill(0));

    for (let i = 0; i < n; i++) dp[i][0] = true;
    dp[0][arr[0]] = true;

    for (let ind = 1; ind < n; ind++) {
        for (let target = 0; target <= k; target++) {
            let notTake = dp[ind - 1][target];
            let take = false;
            if (arr[ind] <= target) take = dp[ind - 1][target - arr[ind]];
            dp[ind][target] = Boolean(take || notTake)
        }
    }

    return dp[n-1][k];
}

const arr = [4, 3, 2, 1];
subsetSumToK(4, 10, arr); //?true
```

tabulation

```javascript
function subsetSumToK(n, k, arr) {
    let prev = Array(k+1).fill(0);
    let cur = Array(k+1).fill(0);
    prev[0] = cur[0] = true;

    prev[arr[0]] = true;

    for (let ind = 1; ind < n; ind++) {
        for (let target = 0; target <= k; target++) {
            let notTake = prev[target];
            let take = false;
            if (arr[ind] <= target) take = prev[target - arr[ind]];
            cur[target] = Boolean(take || notTake)
        }
        prev = [...cur]
    }

    return prev[k];
}

const arr = [4, 3, 2, 1];
subsetSumToK(4, 4, arr); //?true
```

space optimization

lec16 partition equal subset sum | DP on subsequences

Partition Equal Subset Sum

Contributed by **Ashwani**
Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 25 mins | Success Rate 65 %

Share | 69 upvotes

Problem Statement Suggest Edit

You are given an array 'ARR' of 'N' positive integers. Your task is to find if we can partition the given array into two subsets such that the sum of elements in both subsets is equal.
For example, let's say the given array is [2, 3, 3, 3, 4, 5], then the array can be partitioned as [2, 3, 5], and [3, 3, 4] with equal sum 10.
Follow Up:

```
Can you solve this using not more than O(S) extra space,
where S is the sum of all elements of the given array?
```

Sample Input 1:

```
2
6
3 1 1 2 2 1
5
5 6 5 11 6
```

Sample Output 1:

```
true
false
```

Explanation Of Sample Input 1:

```
For the first test case, the array can be partitioned as
([2,1,1,1] and [3, 2]) or ([2,2,1], and [1,1,3]) with sum 5.

For the second test case, the array can't be partitioned.
```

S1 = S2 = totalSum/2 → true

if totalSum → odd → false

```javascript
function subsetSumToK(n, k, arr) {
    let prev = Array(k+1).fill(0);
    let cur = Array(k+1).fill(0);
    prev[0] = cur[0] = true;

    prev[arr[0]] = true; // if(arr[0] <=k) prev[arr[0]] = true;

    for (let ind = 1; ind < n; ind++) {
        for (let target = 0; target <= k; target++) {
            let notTake = prev[target];
            let take = false;
            if (arr[ind] <= target) take = prev[target - arr[ind]];
            cur[target] = Boolean(take || notTake)
        }
        prev = [...cur]
    }

    return prev[k];
}

function canPartition(n, arr) {
    let totalSum = 0;
    for(let item of arr) totalSum += item;

    if(totalSum % 2) return false;
    let target = totalSum/2;

    return subsetSumToK(n, target, arr);
}

const arr = [3, 1, 1, 2, 2, 1];
canPartition(6, arr); //? true
```

lec17 partition a set into two subsets with min absolute sum difference

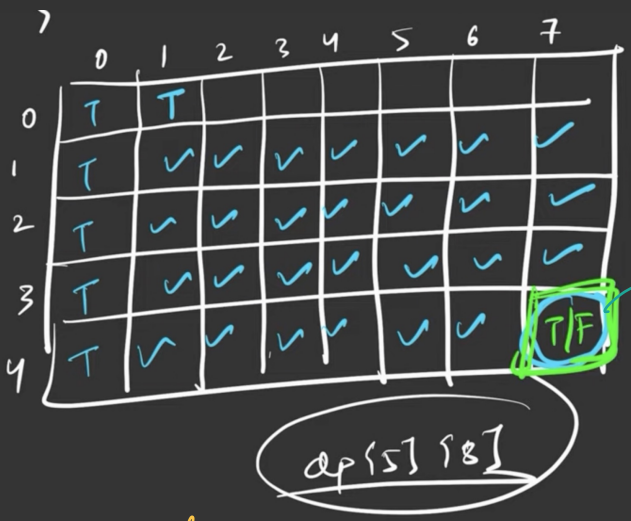
[1, 2, 3, 4]

1,2 & 3,4 → |7-3| = 4

1,3 & 2,4 → |4-6| = 2

1,4 & (2,3) → |5-5| = 0 ✓ ans

tabulation for subset sum & target



n = 5
target = 7

ans → dp[4][7]
↳ dp[n-1][target]

Yes / No

Subset Sum → Tabulation

If we check for a target = k

We can decide if every possible target between 0 (1 & k) is ✓/✗

arr → [3, 2, 7]

if s1=0, we are not picking anything in subset 1

original problem

totalSum = 12

s1 =

s2 = totalSum - s1

	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓	✓	✗	✓
s1 →	0	1	2	3	4	5	6	7	8	9	10	11	12
s2 →	12		10	9		7		5		3	2		0
abs diff →	12		8	6		2		2		6	8		12

ans = 2

Partition a set into two subsets such that the difference of subset sums is minimum.

DS Contributed by **Dhruv Sharma**
Last Updated: 23 Feb, 2023

Hard | 0/120 | Avg time to solve 10 mins | Success Rate 85 %

Share | 140 upvotes

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.
```

```javascript
function minSubsetSumDifference(n, arr) {
    let totalSum = 0;
    for (let item of arr) totalSum += item;
    let k = totalSum;

    let prev = Array(k + 1).fill(0);
    let cur = Array(k + 1).fill(0);

    prev[0] = cur[0] = true;
    if(arr[0] <= k) prev[arr[0]] = true;

    for (let ind = 1; ind < n; ind++) {
        for (let target = 0; target <= k; target++) {
            let notTake = prev[target];
            let take = false;
            if (arr[ind] <= target) take = prev[target - arr[ind]];
            cur[target] = Boolean(take || notTake);
        }
        prev = [...cur];
    }

    // dp[n-1][col--> 0 to totalSum]
    let min = Number.MAX_SAFE_INTEGER;
    for(let s1=0; s1<=totalSum/2; s1++) {
        if(prev[s1] == true) {
            min = Math.min(min, Math.abs((totalSum-s1) - s1));
        }
    }
    return min;
}

const arr = [3,2,7];
minSubsetSumDifference(3, arr); //?2
```

Loc 18 counts subsets with sum k

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.
```

[1, 2, 2, 3] target -> 3
ans -> 3

(1, 2)
(1, 2)
(3)

① express in terms of (ind, target)
② explore all possibilities
③ sum all of the possibilities & return

question asks count
-> in base case, always return 0 or 1

f(ind, target)
{ if(target == 0) return 1;
if(ind == 0) return (a[ind] == target) ? 1 : 0;
notPick = f(ind-1, target)
pick = 0;
if(a[ind] <= target) pick = f(ind-1, target - a[ind])
return pick + notpick;
}

Recursion TC -> O(2^N) SC -> O(N)

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

memoization

dp[n][sum+1]

TC -> O(N x sum)

SC -> O(N x sum) + O(N)

recursion stack space

tabulation

1. write base case
2. look at changing parameters & write nested loops
3. copy recurrence

s -> target

dp[n][sum+1]

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

if(a[0] <= s) dp[0][a[0]] = 1

for(i=1 -> n-1)

{ for(s=0 -> sum)

{

recurrence

}

}

TC -> O(N x sum)

SC -> O(N x sum)

```javascript
function f(ind, sum, arr) {
    if(sum == 0) return 1;
    if(ind == 0) return (arr[0] == sum) ? 1 : 0;

    let notPick = f(ind-1, sum, arr);
    let pick = 0;
    if(arr[ind] <= sum) pick = f(ind-1, sum-arr[ind], arr);

    return notPick + pick;
}

function findWays(target, arr) {
    let n = arr.length;
    return f(n-1, target, arr);
}

const arr = [1,2,2,3];
findWays(3, arr); //?3
```

Recursion

```javascript
function f(ind, sum, arr, dp) {
    if(sum == 0) return 1;
    if(ind == 0) return (arr[0] == sum) ? 1 : 0;
    if(dp[ind][sum] != -1) return dp[ind][sum];

    let notPick = f(ind-1, sum, arr, dp);
    let pick = 0;
    if(arr[ind] <= sum) pick = f(ind-1, sum-arr[ind], arr, dp);

    return dp[ind][sum] = notPick + pick;
}

function findWays(target, arr) {
    let n = arr.length;
    let dp = Array(n).fill().map(() => Array(target+1).fill(-1));
    return f(n-1, target, arr, dp);
}

const arr = [1,2,2,3];
findWays(3, arr); //?3
```

memoization

```javascript
function findWays(target, arr) {
    let n = arr.length;
    let dp = Array(n).fill().map(() => Array(target+1).fill(0));

    for(let i=0; i<n; i++) dp[i][0] = 1;
    if(arr[0] <= target) dp[0][arr[0]] = 1;

    for(let ind=1; ind<n; ind++) {
        for(let sum=0; sum<=target; sum++) {
            let notPick = dp[ind - 1][sum]
            let pick = 0;
            if (arr[ind] <= sum) pick = dp[ind - 1][sum - arr[ind]]

            dp[ind][sum] = notPick + pick;
        }
    }

    return dp[n-1][target]
}

const arr = [1,2,2,3];
findWays(3, arr); //?
```

Tabulation

```javascript
function findWays(target, arr) {
    let n = arr.length;

    let prev = Array(target+1).fill(0);
    let cur = Array(target+1).fill(0);

    for(let i=0; i<n; i++) prev[0] = 1;
    if(arr[0] <= target) prev[arr[0]] = 1;

    for(let ind=1; ind<n; ind++) {
        for(let sum=0; sum<=target; sum++) {
            let notPick = prev[sum]
            let pick = 0;
            if (arr[ind] <= sum) pick = prev[sum - arr[ind]]

            cur[sum] = notPick + pick;
        }
        prev = [...cur]
    }

    return prev[target]
}

const arr = [1,2,2,3];
findWays(3, arr); //?
```

space optimization

lec 18. count partitions with given difference

in the above question, the constraints are $1 \le nums[i] \le 1000$

$1 \le target \le 1000$

if $0 \le nums[i] \le 1000$

we need to modify the base case

```
if (ind == 0) {
    if (sum == 0 && num[0] == 0) return 2;
    if (sum == 0 || sum == num[0]) return 1;
    return 0;
}
```

```javascript
function findWays(target, arr) {
    let n = arr.length;

    let prev = Array(target+1).fill(0);
    let cur = Array(target+1).fill(0);

    if(arr[0] == 0) prev[0] = 2;
    else prev[0] = 1

    for(let i=0; i<n; i++) prev[0] = 1;
    if(arr[0] != 0 && arr[0] <= target) prev[arr[0]] = 1;

    for(let ind=1; ind<n; ind++) {
        for(let sum=0; sum<=target; sum++) {
            let notPick = prev[sum]
            let pick = 0;
            if (arr[ind] <= sum) pick = prev[sum - arr[ind]]

            cur[sum] = notPick + pick;
        }
        prev = [...cur]
    }

    return prev[target]
}
```

```javascript
function f(ind, target, arr) {
    if(ind == 0) {
        if(target == 0 && arr[ind] == 0) return 2;
        if(target == 0 || target == arr[ind]) return 1;
        return 0;
    }

    let notPick = f(ind-1, target, arr);
    let pick = 0;
    if(arr[ind] != 0 && arr[ind] <= target) pick = f(ind-1, target-arr[ind], arr);
    return (notPick + pick);
}

function findWays(target, arr) {
    let n = arr.length;
    return f(n-1, target, arr);
}
```

Partitions With Given Difference

Contributed by **abhi_abhi**
Last Updated: 2 Apr, 2023

Medium | 0/80

Share | 122 upvotes

Problem Statement

Suggest Edit

Given an array 'ARR', partition it into two subsets (possibly empty) such that their union is the original array. Let the sum of the elements of these two subsets be 'S1' and 'S2'.
Given a difference 'D', count the number of partitions in which 'S1' is greater than or equal to 'S2' and the difference between 'S1' and 'S2' is equal to 'D'. Since the answer may be too large, return it modulo '10^9 + 7'.
If 'Pi_Sj' denotes the Subset 'j' for Partition 'i'. Then, two partitions P1 and P2 are considered different if:

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

```
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.
```

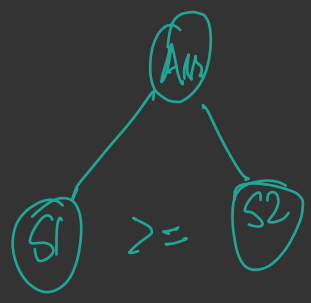
$(S1 - S2 = D)$ $S1 \geq S2$

$S1 = totalSum - S2$

$TotalSum - S2 - S2 = D$

$\Rightarrow TotalSum - D = 2 \times S2$

$\Rightarrow S2 = \frac{TotalSum - D}{2}$



$S1 - S2 = D$

$(5, 2, 6, 4)$ $D = 3$

$\{6, 4\}$ $\{5, 2\}$

$10 - 7 = 3$

$\Rightarrow$ count of subset whose target $= \frac{TotalSum - D}{2}$

edge cases ① $0 \leq Num[i]$

so ② $totalSum - D \geq 0$

③ totalSum has to be even

```
function f(ind, target, arr) {
    if(ind == 0) {
        if(target == 0 && arr[ind] == 0) return 2;
        if(target == 0 || target == arr[ind]) return 1;
        return 0;
    }

    let notPick = f(ind-1, target, arr);
    let pick = 0;
    if(arr[ind] != 0 && arr[ind] <= target) pick = f(ind-1, target-arr[ind], arr);
    return (notPick + pick);
}

function findWays(target, arr) {
    let n = arr.length;
    return f(n-1, target, arr);
}

function countPartitions(n, d, arr) {
    let totalSum = 0;
    for(let item of arr) totalSum += item;
    if(totalSum-d < 0 || (totalSum-d)%2) return 0;
    return findWays((totalSum-d)/2, arr);
}
```

recursion

```
function findWays(target, arr) {
    let n = arr.length;

    let prev = Array(target+1).fill(0);
    let cur = Array(target+1).fill(0);

    if(arr[0] == 0) prev[0] = 2;
    else prev[0] = 1

    for(let i=0; i<n; i++) prev[0] = 1;
    if(arr[0] != 0 && arr[0] <= target) prev[arr[0]] = 1;

    for(let ind=1; ind<n; ind++) {
        for(let sum=0; sum<=target; sum++) {
            let notPick = prev[sum]
            let pick = 0;
            if (arr[ind] <= sum) pick = prev[sum - arr[ind]]

            cur[sum] = notPick + pick;
        }
        prev = [...cur]
    }

    return prev[target]
}

function countPartitions(n, d, arr) {
    let totalSum = 0;
    for(let item of arr) totalSum += item;

    if(totalSum - d < 0 || (totalSum -d) % 2) return 0;
    return findWays((totalSum-d)/2 , arr);
}

const arr = [5,2,6,4];
countPartitions(4, 3, arr); //?
```

space optimization