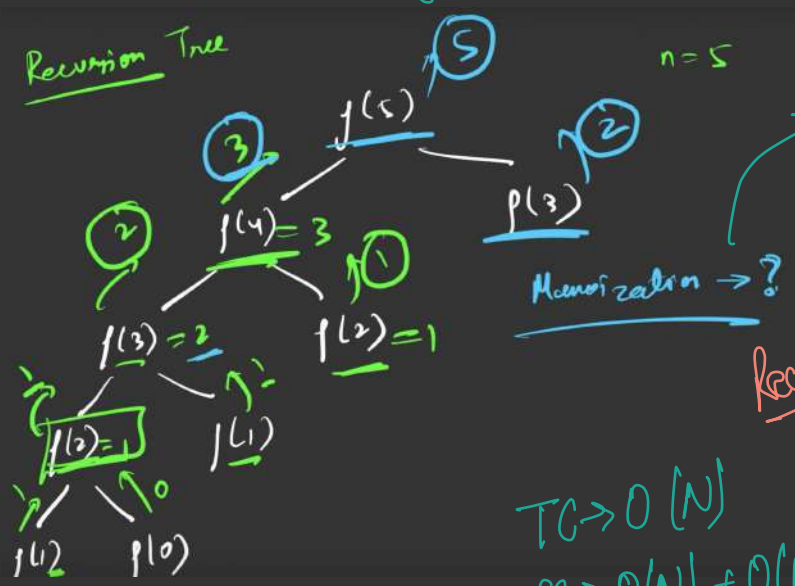
Dynamic Programming

Pre-requisites
Recussion

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] (size)
② base case
dp[0] = 0 → prev2
dp[1] = 1 → prev
③ recussion 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
loop
curr = prev + prev2
prev2 = prev
prev = curr

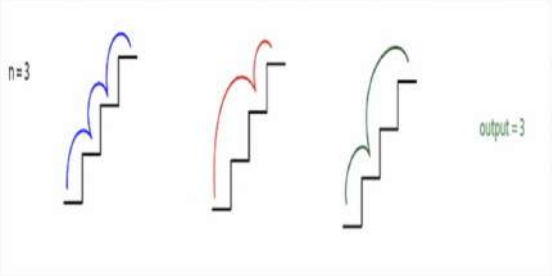
TC → O(N)
SC → O(1)

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

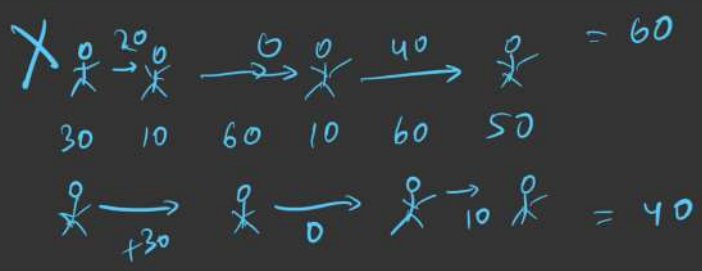
There is a frog on the 1st step of an N stairs long staircase. The frog wants to reach the Nth stair. HEIGHT[i] is the height of the (i+1)th stair.If Frog jumps from ith to jth stair, the energy lost in the jump is given by |HEIGHT[i-1] - HEIGHT[j-1] |.In the Frog is on ith staircase, he can jump either 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	$\boxed{0 \to 3}$
0	1	2	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)
}
```

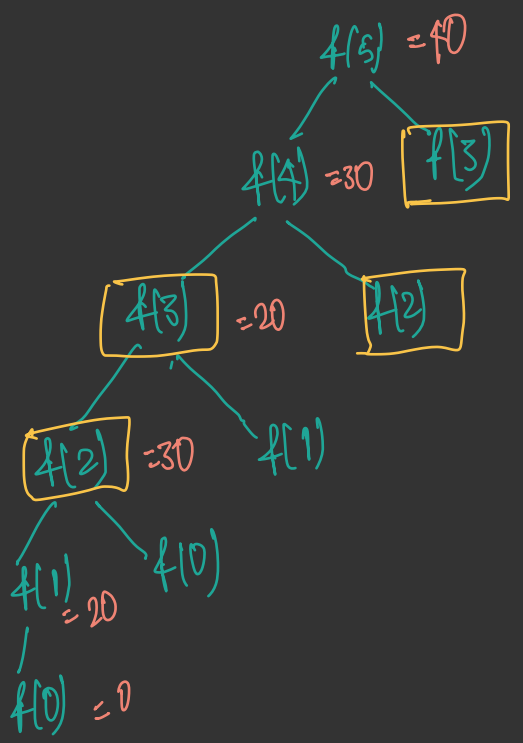
overlapping sub problems

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);
}
```

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];
}
```

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;
}
```

follow-up

$i \to i+1$
$i \to i+2$

K jumps

$i+1,\ i+2,\ i+3, \ldots\ldots, i+K$

```
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];
}
```

$TC \to O(N*K)$
$SC \to O(N)$

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

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

if $K=N$, $SC \to O(N)$
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 6&7 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);
}
```

$TC \to O(2^n)$

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

$f(1) \to$ sum b/w 0 to index 1
$f(2) \to$ sum b/w 0 to index 2

memoization

```javascript
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

```javascript
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)

```javascript
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)

```javascript
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

```javascript
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

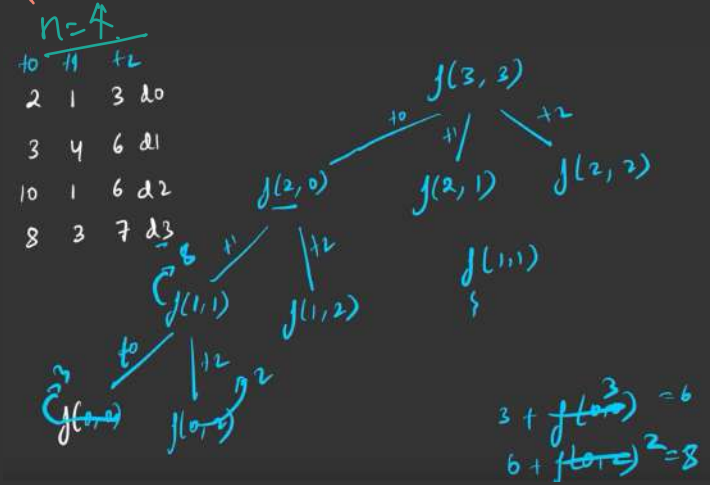
① index

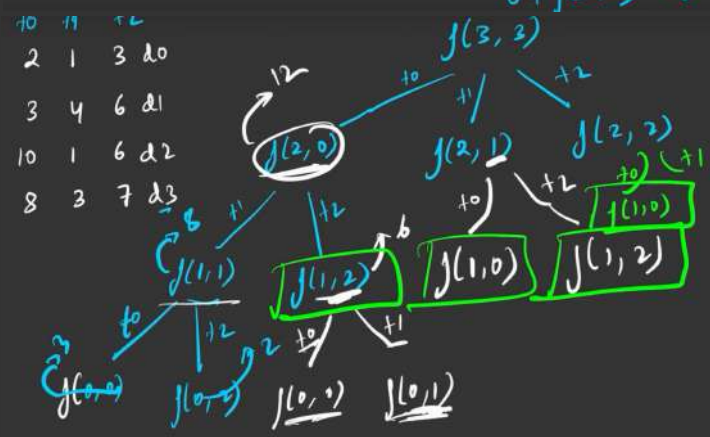
② do stuff on the index

③ find max / all ways

Recursion

n=4





Overlapping sub problems

day → N

last → (0, 1, 2, 3) → 4

dp[N][4] → -1

(N×4) size dp array

TC → O(N×4) × 3

SC → O(N) + O(N×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 recursion (or, the initial call in recursion)

```js
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×4×3)

SC → O(N×4)

Space optimization

```js
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×4×3)

SC → O(4) → O(1)

L8. DP on GRIDS / 2D matrix

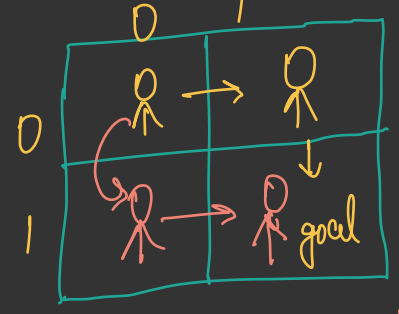
1. count paths
2. count paths with obstacles
3. min path sum
4. max path sum
5. triangle
6. 2 start points

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



$(0,0) \rightarrow (m-1, n-1)$

ways → 2

count all ways → use recursion

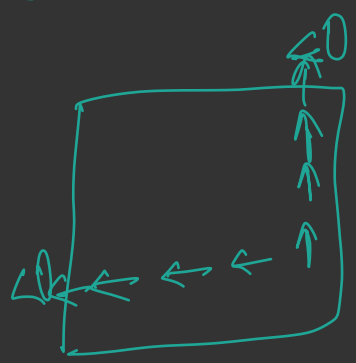
How to write recurrence

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;
}
```



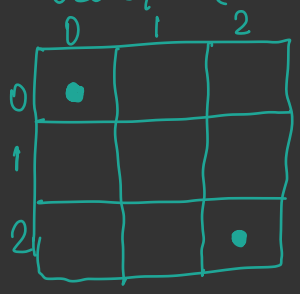
TC → $2^{m \times n}$

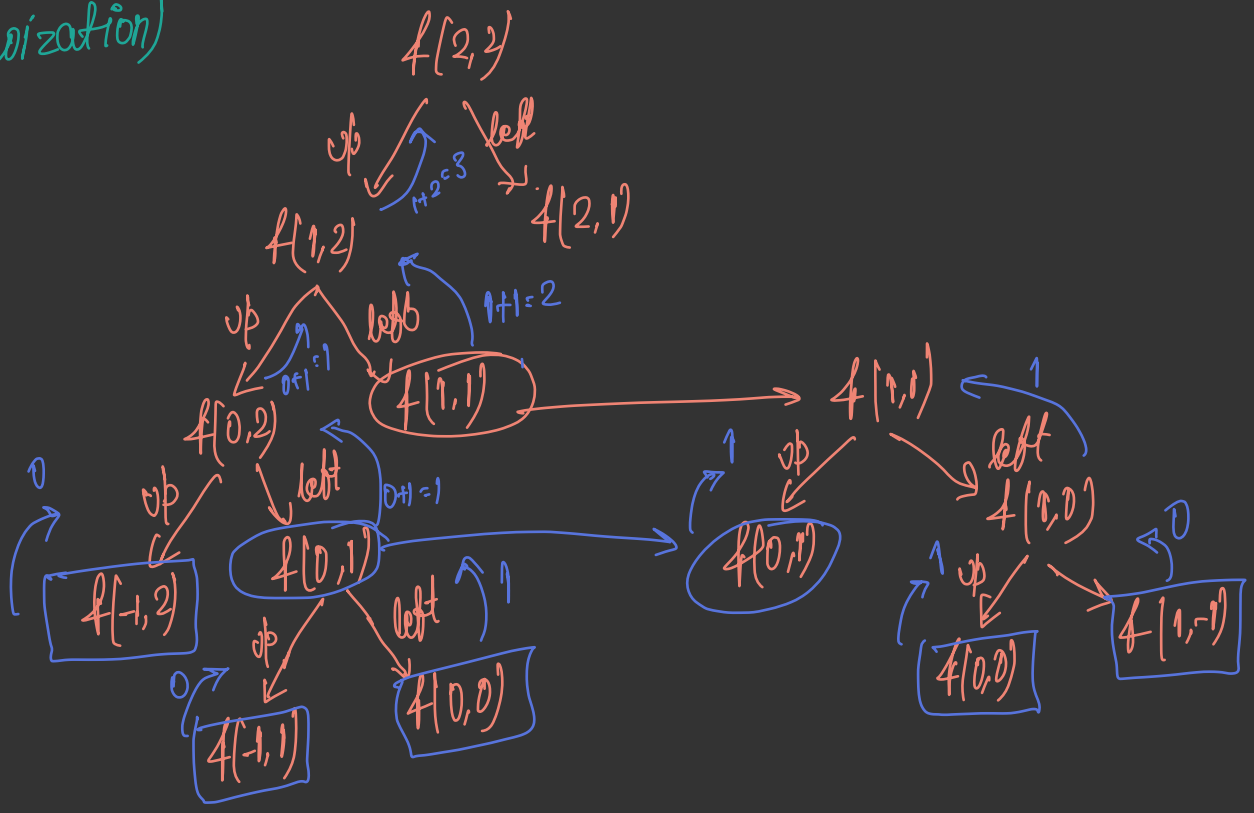
SC → O(path length)

$(m-1) + (n-1)$

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 → $\underbrace{O((N-1)+(M-1))}_{\text{recursion}} + \underbrace{O(N\times M)}_{DP}$

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)

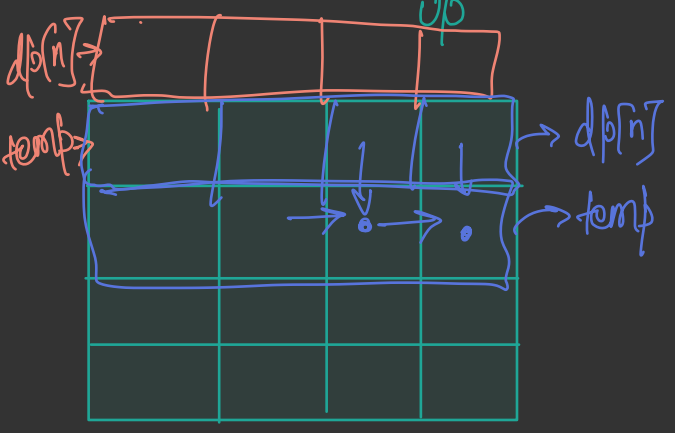
SC → O(M×N)

```
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

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



$dp[n] = 0$

```
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

TC → O(M×N)

SC → O(N)

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.
```

```js
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
```

```js
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
```

```js
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); //?
```

```js
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
```

Q10 Lec12.

Triangle → Fixed starting point & variable ending point

1 → fixed

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

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

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

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

SC → O(N)

```
f(i,j)
{ if (i==n-1) return a[n-1][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

```javascript
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

```javascript
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

Lec13 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,

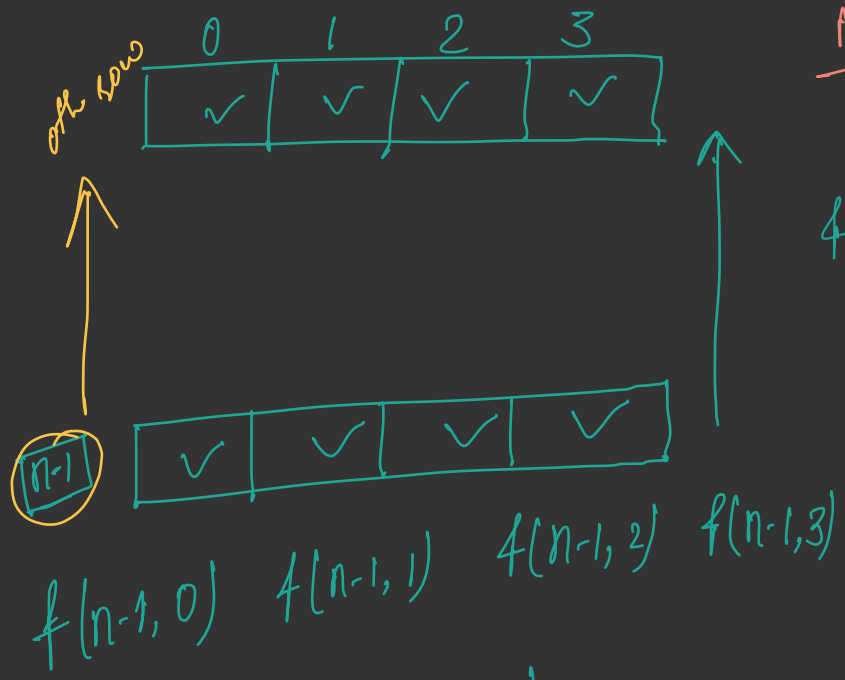
1	2	10	4
100	3	2	1
1	1	20	2
1	2	2	1

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);
  }
```

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

recurrence

base case $\rightarrow$ destination
$\rightarrow$ out of bound cases

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

col < 0
or col ≥ m

```
f(i,j)
{
  if(j<0 || j>=m) return Number.Max_Negative;
  if (i==0) return a[0][j];
  let s = a[i][j] + 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, rd);
}
```

base case

TLE $TC \rightarrow 3^N$ exponential
$SC \rightarrow O(N)$

Memoization

recursion $\rightarrow$ overlapping sub problems $\rightarrow$ memoization

```
f(i,j)  declare dp[N][M]
{
  if(j<0 || j>=m) return Number.Max_Negative;
  if (i==0) return a[0][j];
  if(dp[i][j] != -1) return dp[i][j];
  let s = a[i][j] + 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, rd);
}
```

$TC \rightarrow O(N \times M)$
$SC \rightarrow O(N \times M) + O(N)$

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 → $O(N \times M) + O(M)$

SC → $O(N \times M)$

space optimization

prev[n] , cur[n]

for (j=0 → m-1) ~~dp[0]~~ prev [j] = a[0][j]

for (i=1 → n-1)

{ for (j=0 → m-1)

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

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

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

~~dp[i][j]~~ cur = Max(u, ld, rd);

}

→ 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

lec14 → cherry pick-up, DP on GRIDS, 3D DP

2	3	1	2
3	4	2	2
5	6	3	5

12 + 9 ⇒ 21 ans

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.
```

fixed starting point → variable ending point

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

↓ recursion ↓ recursion

① express everything in terms (i_1, j_1) & (i_2, j_2)
& write base cases → destination, out of bound

② explore all possible paths ↙ ↓ ↘

③ Max Sum

$f(\underbrace{0, 0}_{alice}, \underbrace{0, m-1}_{bob})$

$f(i_1, j_1, i_2, j_2)$

$(i+1, j-1)$ $\quad$ $(i+1, j)$ $\quad$ $(i+1, j+1)$

Alice × Bob

$3 \times 3 \Rightarrow 9$ combos of path

```
{ 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;
}
```

$\rightarrow$ if $dp[i][j_1][j_2] != -1$ return $dp[i][j_1][j_2]$

$\rightarrow$ return $dp[i][j_1][j_2] = maxi$;

TC $\rightarrow$ $(3^N \times 3^N)$ $\rightarrow$ exponential

SC $\rightarrow$ $O(N)$

memoization

(i) (j_1) (j_2)

N M M

$dp \rightarrow [N] \times [M] \times [M]$

$dp[i][j_1][j_2]$

TC $\rightarrow$ $O(N \times M \times M) \times 9$ $\rightarrow$ number of different states

SC $\rightarrow$ $O(N \times M \times M) + O(N)$

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]

base case

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];
}
}

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	subsets
✓	✓
✗	✓
✓	✓

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

TC -> O(N x target)

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

ind target

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

```
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 not take = f(ind-1, target)
  bool take = false
    if (target >= a[ind])
        take = f(ind-1, target - a[ind]);

  return false or not take;
}
```

dp[ind][target]

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;

    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
```

if(arr[0] <= k) prev[arr[0]] = 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

	0	1	2	3	4	5	6	7
0	T	T						
1	T	✓	✓	✓	✓	✓	✓	✓
2	T	✓	✓	✓	✓	✓	✓	✓
3	T	✓	✓	✓	✓	✓	✓	✓
4	T	✓	✓	✓	✓	✓	✓	T/F

n = 5
target = 7

ans dp[4][7]
↳ dp[n-1][target]

dp[5][8]

Yes | No

Subset Sum → Tabulation

if we check for a target = k

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

original problem

totalSum = 12

S1 =

S2 = totalSum - S1

ex → [3, 2, 7]

if S1 = 0, we are not picking anything in subset 1

	✓	✗	✓	✓	✗	✓	✗	✓	✗	✓	✓	✗	✓
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
```

Lec 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[ind][target]

memoization

dp[n][sum+1]

TC -> O(N x sum)

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

recursion AS

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.
```

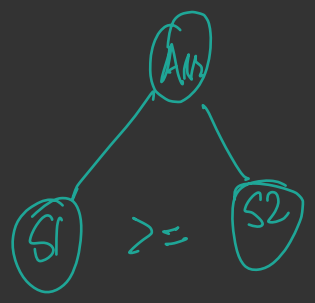
$(S1 - S2 = D)$ $\quad S1 > S2$

$S1 = totalSum - S2$

$TotalSum - S2 - S2 = D$

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

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



$S1 - S2 = D$

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

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

$10 - 7 = 3$

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

edge cases ① $0 \le Num[i]$

so ② $totalSum - D \ge 0$

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

lec20 0/1 knapsack | recursion to single array space optimised approach

Recursion

try out all combinations
↓
take the best total value combination

① express everything in terms of index & write base cases

(ind, w)
↑ bag weight

② explore all possibilities
pick & not pick

③ max (all possibilities)

DP-14

0/1 Knapsack

n = 3

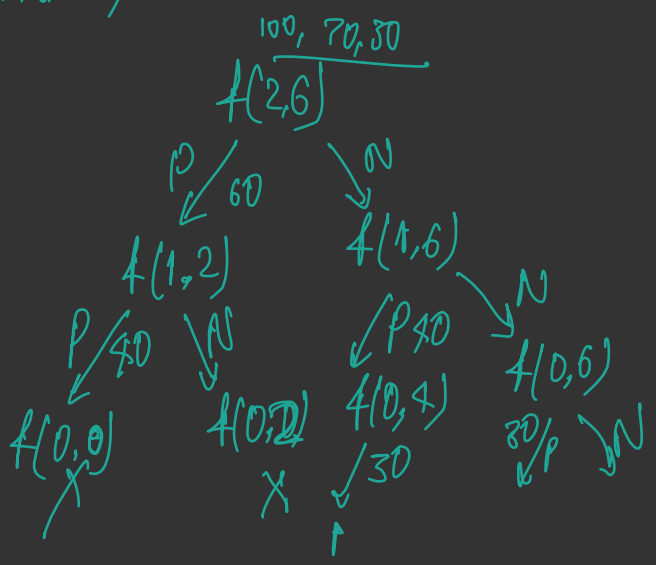
	2	X	1
wt →	3	4	5
val →	30	50	60

Bag → wt = 8 → 3 → 0

Total → 60 + 30 = 90 ↑↑

wt →	3,	2,	4
val →	30	40	60
ind →	0	1	2

f(n-1, 6)




```
f(ind, w)
{ if(ind==0)
  { if (wt[0] ≤ w) return val[0];
    else return 0;
  }
  notTake = 0 + f(ind-1, w)
  take = INT_MAX
  if(wt[ind] ≤ w) pick = val[ind] + f(ind-1, w-wt[ind])
  return max(pick, notpick)
}
```

TC → 2^N

SC → O(N)

memoization

changing params → ind, w

ind: 0 → N-1

w: 0 → W

dp [N] [W+1]

TC → O(N × W)

SC → O(N × W) + O(N)

ASS

Tabulation

① base cases

② changing params in loops

③ copy the recurrence

dp[N][W+1] = 0

ind → 1 to n-1

wt → 0 to w

```
function f(ind, wt, val, W) {
    if(ind == 0) {
        if(wt[ind] <= W) return val[ind];
        else return 0;
    }

    let notTake = 0 + f(ind-1, wt, val, W);
    let take = Number.MIN_SAFE_INTEGER;
    if(wt[ind] <= W) take = val[ind] + f(ind-1, wt, val, W - wt[ind]);

    return Math.max(notTake, take);
}

function knapsack(wt, val, n, W) {
    return f(n-1, wt, val, W);
}

const value = [5,4,8,6];
const weight = [1,2,4,5];

knapsack(weight, value,4,5); ///13
```

Recursion

```
function f(ind, wt, val, W, dp) {
    if(ind == 0) {
        if(wt[ind] <= W) return val[ind];
        else return 0;
    }
    if(dp[ind][W] != -1) return dp[ind][W];

    let notTake = 0 + f(ind-1, wt, val, W, dp);
    let take = Number.MIN_SAFE_INTEGER;
    if(wt[ind] <= W) take = val[ind] + f(ind-1, wt, val, W - wt[ind], dp);

    return dp[ind][W] = Math.max(notTake, take);
}

function knapsack(wt, val, n, W) {
    let dp = Array(n).fill().map(() => Array(W+1).fill(-1));
    return f(n-1, wt, val, W, dp);
}

const value = [5,4,8,6];
const weight = [1,2,4,5];

knapsack(weight, value,4,5); ///13
```

memoization

```
function knapsack(wt, val, n, W) {
    let dp = Array(n)
        .fill()
        .map(() => Array(W + 1).fill(0));

    for (let i = wt[0]; i < W; i++) dp[0][i] = val[0];

    for (let ind = 1; ind < n; ind++) {
        for (let w = 0; w <= W; w++) {
            let notTake = 0 + dp[ind - 1][w];
            let take = Number.MIN_SAFE_INTEGER;
            if (wt[ind] <= w) take = val[ind] + dp[ind - 1][w - wt[ind]];
            dp[ind][w] = Math.max(notTake, take);
        }
    }

    return dp[n - 1][W];
}
```

Tabulation

```
function knapsack(wt, val, n, W) {

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

    for (let i = wt[0]; i < W; i++) prev[i] = val[0];

    for (let ind = 1; ind < n; ind++) {
        for (let w = 0; w <= W; w++) {
            let notTake = 0 + prev[w];
            let take = Number.MIN_SAFE_INTEGER;
            if (wt[ind] <= w) take = val[ind] + prev[w - wt[ind]];
            cur[w] = Math.max(notTake, take);
        }
        prev = [...cur]
    }

    return prev[W];
}
```

space optimization

```javascript
function knapsack(wt, val, n, W) {
    let prev = Array(W+1).fill(0);

    for (let i = wt[0]; i < W; i++) prev[i] = val[0];

    for (let ind = 1; ind < n; ind++) {
        for (let w = 0; w <= W; w++) {
            let notTake = 0 + prev[w];
            let take = Number.MIN_SAFE_INTEGER;
            if (wt[ind] <= w) take = val[ind] + prev[w - wt[ind]];
            prev[w] = Math.max(notTake, take);
        }
    }

    return prev[W];
}
```

now optimization

---

lec21 minimum coins, infinite supplies patterns

trying out all combos to find target

take the combo that has min coins

knapsack.

infinite supply or multiple supply

pick → ind will remain same
target will update

Problem Statement Suggest Edit

You are given an array of 'N' distinct integers and an integer 'X' representing the target sum. You have to tell the minimum number of elements you have to take to reach the target sum 'X'.

Note:

```
You have an infinite number of elements of each type.
```

For Example

```
If N=3 and X=7 and array elements are [1,2,3].
Way 1 - You can take 4 elements  [2, 2, 2, 1] as 2 + 2 + 2 +
1 = 7.
Way 2 - You can take 3 elements  [3, 3, 1] as 3 + 3 + 1 = 7.
Here, you can see in Way 2 we have used 3 coins to reach the
target sum of 7.
Hence the output is 3.
```

Recursion

TC → >> $O(2^N)$ exponential

SC → >> $O(N)$

```
f(ind, target)
{ if (ind == 0)
    { if ((target % num[ind]) == 0) return target/num[ind]
      return MAX_INT;
    }
  notpick = 0 + f(ind-1, target)
  pick = INT_MAX
  if( num[ind] <= target) pick = 1 + f(ind, target - num[ind])
  return Math.min(pick, notPick)
}
```

memoization

ind, target

dp[N][target+1]

TC → O(N × Target)

SC → O(N × Target) + O(Target)
AS

Tabulation

1. base case
2. changing-params ind, T
3. copy recurrence

dp[N][target+1]

for (T=0 → target)
    if (T % a[0] == 0)
        dp[0][T] = T/a[0]
    else
        dp[0][T] = INT_MAX

```
function f(ind, T, num) {
    if (ind == 0) {
        if ((T % num[ind]) == 0) return T/num[ind];
        else return Number.MAX_SAFE_INTEGER;
    }

    let notTake = 0 + f(ind - 1, T, num);
    let take = Number.MAX_SAFE_INTEGER;
    if (num[ind] <= T) take = 1 + f(ind, T - num[ind], num);

    return Math.min(notTake, take);
}

function minimumElements(T, num) {
    let n = num.length;
    return f(n-1, T, num);
}

const num = [1, 2, 3];
minimumElements(7, num); //? 3
```

Recursion

```
function f(ind, T, num, dp) {
    if (ind == 0) {
        if ((T % num[ind]) == 0) return T/num[ind];
        else return Number.MAX_SAFE_INTEGER;
    }
    if(dp[ind][T] != -1) return dp[ind][T];
    let notTake = 0 + f(ind - 1, T, num, dp);
    let take = Number.MAX_SAFE_INTEGER;
    if (num[ind] <= T) take = 1 + f(ind, T - num[ind], num, dp);

    return dp[ind][T] = Math.min(notTake, take);
}

function minimumElements(T, num) {
    let n = num.length;
    let dp = Array(n).fill().map(() => Array(T+1).fill(-1));
    return f(n-1, T, num, dp);
}

const num = [1, 2, 3];
minimumElements(7, num); //? 3
```

memoization

```javascript
function minimumElements(T, num) {
    let n = num.length;
    let dp = Array(n)
        .fill()
        .map(() => Array(T + 1).fill(0));

    for (let t = 0; t <= T; t++) {
        if (t % num[0] == 0) {
            dp[0][t] = t / num[0];
        } else {
            dp[0][t] = Number.MAX_SAFE_INTEGER;
        }
    }

    for (let ind = 1; ind < n; ind++) {
        for (let t = 0; t <= T; t++) {
            let notTake = 0 + dp[ind - 1][t];
            let take = Number.MAX_SAFE_INTEGER;
            if (num[ind] <= t) take = 1 + dp[ind][t - num[ind]];

            dp[ind][t] = Math.min(notTake, take);
        }
    }

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

const num = [1, 2, 3];
minimumElements(7, num); //? 3
```

Tabulation

```javascript
function minimumElements(T, num) {
    let n = num.length;
    let dp = Array(n)
        .fill()
        .map(() => Array(T + 1).fill(0));

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

    for (let t = 0; t <= T; t++) {
        if (t % num[0] == 0) {
            prev[t] = t / num[0];
        } else {
            prev[t] = Number.MAX_SAFE_INTEGER;
        }
    }

    for (let ind = 1; ind < n; ind++) {
        for (let t = 0; t <= T; t++) {
            let notTake = 0 + prev[t];
            let take = Number.MAX_SAFE_INTEGER;
            if (num[ind] <= t) take = 1 + cur[t - num[ind]];

            cur[t] = Math.min(notTake, take);
        }
        prev = [...cur]
    }

    return prev[T];
}

const num = [1, 2, 3];
minimumElements(7, num); //? 3
```

space optimization

---

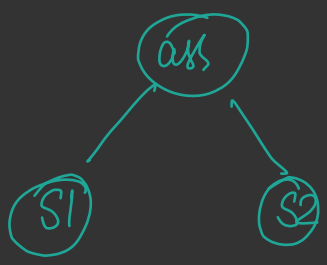
lec22 target Sum subsequence

arr = [1, 2, 3, 1]

target = 3

1st way → 1 2 3 1 (− + + −)

2nd way → 1 2 3 1 (+ − + +)



(2,3) − (1,1) ⇒ 3

(1,3,1) − (2) ⇒ 3

S1 − S2 = target

count (S1 − S2) : Difference

↳ lec18.

Problem Statement Suggest Edit

You are given an array 'ARR' of 'N' integers and a target number, 'TARGET'. Your task is to build an expression out of an array by adding one of the symbols '+' and '-' before each integer in an array, and then by concatenating all the integers, you want to achieve a target. You have to return the number of ways the target can be achieved.

For Example :

```
You are given the array 'ARR' = [1, 1, 1, 1, 1], 'TARGET' =
3. The number of ways this target can be achieved is:
1. -1 + 1 + 1 + 1 + 1 = 3
2. +1 - 1 + 1 + 1 + 1 = 3
3. +1 + 1 - 1 + 1 + 1 = 3
4. +1 + 1 + 1 - 1 + 1 = 3
5. +1 + 1 + 1 + 1 - 1 = 3
These are the 5 ways to make. Hence the answer is 5.
```

```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];
}

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);
}

function targetSum(n, d, arr) {
    return countPartitions(n, d, arr);
}

const num = [1, 2, 3, 1];
targetSum(4, 3, num); //?
```

space optimized

lec23. count change2, infinite supply problems

Ways To Make Coin Change

Contributed by Ankit Kharb
Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 20 mins | Success Rate 80 %

Share | 67 upvotes

Problem Statement Suggest Edit

You are given an infinite supply of coins of each of denominations D = {D0, D1, D2, D3, ... Dn-1}. You need to figure out the total number of ways W, in which you can make a change for value V using coins of denominations from D. Print 0, if a change isn't possible.

Detailed explanation (Input/output format, Notes, Images)

Sample Input 1 :

```
3
1 2 3
4
```

Sample Output 1:

```
4
```

Explanation For Sample Output 1:

```
We can make a change for the value V = 4 in four ways.
1. (1,1,1,1),
2. (1,1, 2), [One thing to note here is, (1, 1, 2) is same as
that of (2, 1, 1) and (1, 2, 1)]
3. (1, 3), and
4. (2, 2)
```

Recursion

1. express in (ind, target)
2. base cases
3. explore all possibilities
4. sum all ways & return

TC → exponential
SC → O(target)

```
f(ind, T)
{   if (ind==0){
        if( T % ass[ind] ==0) return 1 else return 0;

    }
    not take = f(ind-1, T)
    take = 0
    if( ass[ind] ≤ T) take = f(ind, T-a[ind])
    return take + notTake
}
```

memoization

Changing params → ind, Target

0 to n-1 | 0 to T

dp[N][T+1]

TC → O(N×T)
SC → O(N×T) + O(target) ASS

Tabulation

```
dp[N][T+1]
for ( T=0 to target)
    dp[0][T] = ( T% ass[0] ==0) ? 1 : 0;
for ( ind=1; ind<n; ind++)
{   for (T=0; T≤target; T++)
    {   notTake = dp[ind-1][T]
        take =0
        if( a[ind] <=T) take = dp[ind][T-a[ind]]
```

dp[ind][T] = take + notTake

}

}

return dp[n-1][value]

```
function f(ind, T, arr) {
    if(ind == 0) {
        return T % arr[ind] === 0 ? 1 : 0;
    }

    let notTake = f(ind-1, T, arr);
    let take = 0;
    if(arr[ind] <= T) take = f(ind, T - arr[ind], arr);

    return take + notTake;
}

function countWaysToMakeChange(n, target, arr) {
    return f(n-1, target, arr);
}

const num = [1, 2, 3];
countWaysToMakeChange(3, 4, num); //?
```

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

    let notTake = f(ind-1, T, arr, dp);
    let take = 0;
    if(arr[ind] <= T) take = f(ind, T - arr[ind], arr, dp);

    return dp[ind][T] = take + notTake;
}

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

const num = [1, 2, 3];
countWaysToMakeChange(3, 4, num); //?
```

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

    for(let T=0; T<=target; T++) dp[0][T] = (T % arr[0] === 0) ? 1 : 0;

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

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

const num = [1, 2, 3];
countWaysToMakeChange(3, 4, num); //?
```

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

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

    for(let T=0; T<=target; T++) prev[T] = (T % arr[0] === 0) ? 1 : 0;

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

    return prev[target];
}

const num = [1, 2, 3];
countWaysToMakeChange(3, 4, num); //?
```

lec 24 unbounded knapsack

Unbounded Knapsack

Contributed by **Ambuj verma**
Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 15 mins | Success Rate 85 %

Share | 58 upvotes

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.
```

Constraints

```
1 <= T <= 50
1 <= N <= 10^3
1 <= W <= 10^3
1 <= PROFIT[ i ] , WEIGHT[ i ] <= 10^8

Time Limit: 1sec
```

Recursion

f(ind, w)

{ if(ind == 0) {

return $\left(\left(\frac{w}{wt[0]}\right) \times val[0]\right)$

}

notTake = 0 + f(ind-1, w)

take = INT_MIN

if(wt[ind] ≤ w) take = val[ind] + f(ind, w − wt[ind])

return max(take, notTake)

}

TC → exponential

(not 2^N because we are standing at same index)

SC → O(wt)

memoization

ind , wt

dp[N][wt+1]

Tabulation

① Base case

② loops. ind → 1 to n-1

w → 0 to wt

③ copy the recurrence

```
function f(ind, W, profit, weight) {
    if(ind == 0) {
        return ((W/weight[ind]) * profit[ind])
    }

    let notTake = 0 + f(ind-1, W, profit, weight);
    let take = Number.MIN_SAFE_INTEGER;
    if(weight[ind] <= W) take = profit[ind] + f(ind, W - weight[ind], profit, weight);

    return Math.max(take, notTake);
}

function unboundedKnapsack(W, profit, weight) {
    let n = profit.length;
    return f(n-1, W, profit, weight);
}

const profit = [5, 11, 13];
const weight = [2, 4, 6];
unboundedKnapsack(10, profit, weight); ///27
```

recursion

```
function f(ind, W, profit, weight, dp) {
    if(ind == 0) {
        return ((W/weight[ind]) * profit[ind])
    }

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

    let notTake = 0 + f(ind-1, W, profit, weight, dp);
    let take = Number.MIN_SAFE_INTEGER;
    if(weight[ind] <= W) take = profit[ind] + f(ind, W - weight[ind], profit, weight, dp);

    return dp[ind][W] = Math.max(take, notTake);
}

function unboundedKnapsack(W, profit, weight) {
    let n = profit.length;
    let dp = Array(n).fill().map(() => Array(W+1).fill(-1));
    return f(n-1, W, profit, weight, dp);
}

const profit = [5, 11, 13];
const weight = [2, 4, 6];
unboundedKnapsack(10, profit, weight); ///27
```

memoization

```javascript
function unboundedKnapsack(W, profit, weight) {
    let n = profit.length;
    let dp = Array(n).fill().map(() => Array(W+1).fill(0));

    for(let w=0; w<=W; w++) dp[0][w] = (w / weight[0]) * profit[0];

    for(let ind=1; ind<n; ind++) {
        for(let w=0; w<= W; w++) {
            let notTake = 0 + dp[ind - 1][w];
            let take = Number.MIN_SAFE_INTEGER;
            if (weight[ind] <= w)
                take =
                    profit[ind] + dp[ind][w - weight[ind]];

            dp[ind][w] = Math.max(take, notTake);
        }
    }

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

const profit = [5, 11, 13];
const weight = [2, 4, 6];
unboundedKnapsack(10, profit, weight); //?27
```

tabulation

Todo → optimize it to 1 array

```javascript
function unboundedKnapsack(W, profit, weight) {
    let n = profit.length;
    let prev = Array(W+1).fill(0);
    let cur = Array(W+1).fill(0);

    for(let w=0; w<=W; w++) prev[w] = (w / weight[0]) * profit[0];

    for(let ind=1; ind<n; ind++) {
        for(let w=0; w<= W; w++) {
            let notTake = 0 + prev[w];
            let take = Number.MIN_SAFE_INTEGER;
            if (weight[ind] <= w)
                take =
                    profit[ind] + cur[w - weight[ind]];

            cur[w] = Math.max(take, notTake);
        }
        prev = [...cur]
    }

    return prev[W];
}

const profit = [5, 11, 13];
const weight = [2, 4, 6];
unboundedKnapsack(10, profit, weight); //?27
```

space optimization

```javascript
function unboundedKnapsack(W, profit, weight) {
    let n = profit.length;
    let prev = Array(W+1).fill(0);

    for(let w=0; w<=W; w++) prev[w] = (w / weight[0]) * profit[0];

    for(let ind=1; ind<n; ind++) {
        for(let w=0; w<= W; w++) {
            let notTake = 0 + prev[w];
            let take = Number.MIN_SAFE_INTEGER;
            if (weight[ind] <= w)
                take =
                    profit[ind] + prev[w - weight[ind]];

            prev[w] = Math.max(take, notTake);
        }
    }

    return prev[W];
}

const profit = [5, 11, 13];
const weight = [2, 4, 6];
unboundedKnapsack(10, profit, weight); //?
```

lec25 → Rod cutting problem, 1D array space optimised

Rod cutting problem

Contributed by **Mutiur Rehman khan**
Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 40 mins | Success Rate 75 %

Share | 125 upvotes

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.

Sample Input 1:

```
2
5
2 5 7 8 10
8
3 5 8 9 10 17 17 20
```

Sample Output 1:

```
12
24
```

Explanation Of Sample Input 1:

```
Test case 1:

All possible partitions are:
1,1,1,1,1          max_cost=(2+2+2+2+2)=10
1,1,1,2            max_cost=(2+2+2+5)=11
1,1,3              max_cost=(2+2+7)=11
1,4                max_cost=(2+8)=10
5                  max_cost=(10)=10
2,3                max_cost=(5+7)=12
1,2,2              max _cost=(1+5+5)=12

Clearly, if we cut the rod into lengths 1,2,2, or 2,3, we get
the maximum cost which is 12.
```

Recursion

① explore in → (ind, N)

② explore all possibilities → not Take, → Take

③ return max of all possibilities

2	5	7	8	10
0	1	2	3	4

$f(\overset{n-1}{4}, N)$

till index 4, what is the max price obtained?

```
f(ind, N){
    if(ind==0){
        return (N x price[0]);
    }
    notTake = 0 + f(ind-1, N)
    take = INT_MIN
    rodLength = ind+1
    if(rodLength <= N) take = price[ind] + f(ind, N-rodLength)
    return max(take, notTake)
}
```

TC → exponential (greater than 2^N)

SC → O(target)

Memoization

ind, N

dp[N][N+1]

TC → O(N x N)

SC → O(N x N) + O(target)
ASS

Tabulation

① write base case

② changing params

③ recurrence

dp[N][N+1]

for(N => 0 to N) dp[0][N] = N x price[0]

for(ind => 1 to N-1)

    for(N => 0 to N)

        recurrence

```javascript
function f(ind, N, price) {
    if(ind == 0) {
        return N * price[ind];
    }

    let notTake = 0 + f(ind-1, N, price);
    let take = Number.MIN_SAFE_INTEGER;
    let rodLength = ind+1;
    if(rodLength <= N) take = price[ind] + f(ind, N-rodLength, price);

    return Math.max(take, notTake);
}

function cutRod(N, price) {
    return f(N-1, N, price);
}

const price = [2,5,7,8,10];
cutRod(5, price); //? 12
```

recursion

```javascript
function f(ind, N, price, dp) {
    if(ind == 0) {
        return N * price[ind];
    }

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

    let notTake = 0 + f(ind-1, N, price, dp);
    let take = Number.MIN_SAFE_INTEGER;
    let rodLength = ind+1;
    if(rodLength <= N) take = price[ind] + f(ind, N-rodLength, price, dp);

    return dp[ind][N] = Math.max(take, notTake);
}

function cutRod(N, price) {
    let dp = Array(N).fill().map(() => Array(N+1).fill(-1));
    return f(N-1, N, price, dp);
}

const price = [2,5,7,8,10];
cutRod(5, price); //? 12
```

memoization

```javascript
function cutRod(N, price) {
    let dp = Array(N).fill().map(() => Array(N+1).fill(0));

    for(let n=0; n<=N; n++) dp[0][n] = n * price[0];

    for(let ind=1; ind<N; ind++) {
        for(let n=0; n<=N; n++) {
            let notTake = 0 + dp[ind - 1][n];
            let take = Number.MIN_SAFE_INTEGER;
            let rodLength = ind + 1;
            if (rodLength <= n)
                take = price[ind] + dp[ind][n - rodLength];

            (dp[ind][n] = Math.max(take, notTake));
        }
    }

    return dp[N-1][N];
}

const price = [2,5,7,8,10];
cutRod(5, price); //? 12
```

Tabulation

```javascript
function cutRod(N, price) {
    let prev = Array(N+1).fill(0);
    let cur = Array(N+1).fill(0);

    for(let n=0; n<=N; n++) prev[n] = n * price[0];

    for(let ind=1; ind<N; ind++) {
        for(let n=0; n<=N; n++) {
            let notTake = 0 + prev[n];
            let take = Number.MIN_SAFE_INTEGER;
            let rodLength = ind + 1;
            if (rodLength <= n)
                take = price[ind] + cur[n - rodLength];

            (cur[n] = Math.max(take, notTake));
        }
        prev = [...cur]
    }

    return prev[N];
}

const price = [2,5,7,8,10];
cutRod(5, price); //? 12
```

space optimization

```javascript
function cutRod(N, price) {
    let prev = Array(N+1).fill(0);

    for(let n=0; n<=N; n++) prev[n] = n * price[0];

    for(let ind=1; ind<N; ind++) {
        for(let n=0; n<=N; n++) {
            let notTake = 0 + prev[n];
            let take = Number.MIN_SAFE_INTEGER;
            let rodLength = ind + 1;
            if (rodLength <= n)
                take = price[ind] + prev[n - rodLength];

            (prev[n] = Math.max(take, notTake));
        }
    }

    return prev[N];
}

const price = [2,5,7,8,10];
cutRod(5, price); //? 12
```

1D optimized

lec26 longest common subsequences | DP on strings

Longest Common Subsequence

Contributed by **Deep Mavani**
Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 39 mins

Share | 98 upvotes

Problem Statement Suggest Edit

Given two strings, 'S' and 'T' with lengths 'M' and 'N', find the length of the 'Longest Common Subsequence'.
For a string 'str'(per se) of length K, the subsequences are the strings containing characters in the same relative order as they are present in 'str,' but not necessarily contiguous. Subsequences contain all the strings of length varying from 0 to K.

Example :

```
Subsequences of string "abc" are:  ""(empty string), a, b, c,
ab, bc, ac, abc.
```

Detailed explanation (Input/output format, Notes, Images)

Constraints :

```
0 <= M <= 10 ^ 3
0 <= N <= 10 ^ 3

Time Limit: 1 sec
```

Sample Input 1 :

```
adebc
dcadb
```

Sample Output 1 :

```
3
```

Explanation Of The Sample Output 1 :

```
Both the strings contain a common subsequence 'adb', which is
the longest common subsequence with length 3.
```

S1 = 'adebc'
S2 = "dcadb"

DP on strings → comparison
→ replaces/edit

M1
Brute force → exponential

M2 → recursion
generate all subsequences & compare on the way

ex acd / ced

a	c
c	e
d	d
ac	ce
ad	cd
cd	ed
acd	ced

output ② cd

rules to write recursion
① express in index (ind1, ind2)
② explore all possibilities on that index
③ take best among them

f(2,2) → los of str1 (0....2) & str2 (0....2)

f(2,5) → str1 (0...2) & str2 (0....5)

recurrence
① if match on indexes
1 + f(ind1-1, ind2-1)

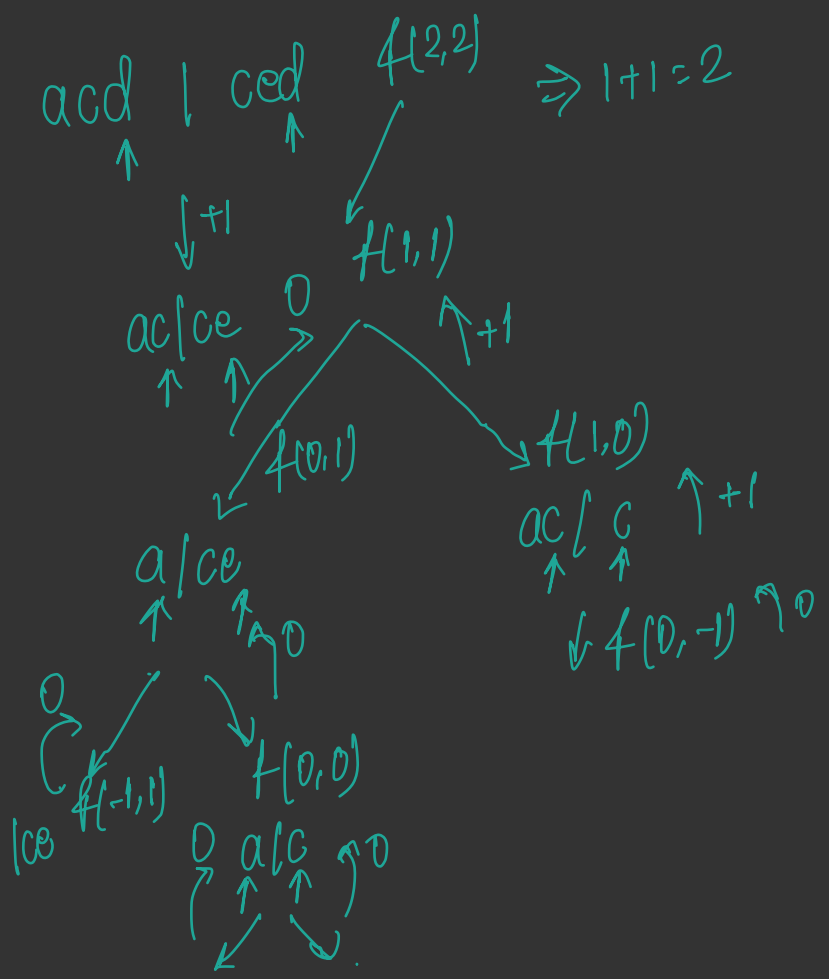
② not match

$$0 + \max(f(ind1-1, ind2), f(ind1, ind2-1))$$

TC → $2^n \times 2^m$ exponential

SC →

```
f(ind1, ind2)
{ if(ind1<0 || ind2<0)
      return 0;
  if (s1[ind1] == s2[ind2])
      return 1+f(ind1-1, ind2-1)
  return 0+ max(f(ind1-1, ind2), f(ind1, ind2-1))
}
```



Memoization

$f(i,j) \rightarrow$ lcs of str1[0.....i] & str2[0.....j]

dp[N][M] = -1

TC → $O(N \times M)$

SC → $O(N \times M) + O(N+M)$ ASS

```javascript
function f(i, j, s1, s2) {
    if(i < 0 || j < 0) {
        return 0
    }

    if(s1[i] == s2[j]) {
        return 1 + f(i-1, j-1, s1, s2);
    }

    return Math.max(f(i-1, j, s1, s2), f(i, j-1, s1, s2));
}

function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    return f(n-1,m-1,s1,s2);
}

const s1 = "adebc";
const s2 = "dcadb";
lcs(s1, s2); //? 3
```

Recursion

```javascript
function f(i, j, s1, s2, dp) {
    if(i < 0 || j < 0) {
        return 0
    }

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

    if(s1[i] == s2[j]) {
        return dp[i][j] = 1 + f(i-1, j-1, s1, s2, dp);
    }

    return dp[i][j] = Math.max(f(i-1, j, s1, s2, dp), f(i, j-1, s1, s2, dp));
}

function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n).fill().map(() => Array(m).fill(-1));
    return f(n-1,m-1,s1,s2,dp);
}

const s1 = "adebc";
const s2 = "dcadb";
lcs(s1, s2); //? 3
```

memoization

Tabulation / bottom up

1. declare dp array
2. copy the base case
3. nested loops for changing parameters
4. copy the recurrence

dp[N+1][M+1] = 0

for (i = 0 to N) dp[i][0] = 0

for (j = 0 to M) dp[0][j] = 0

for (i = 1 to N)

    for (j = 1 to M)

    }

    ⋮

1. dp[N][M] = 0
2. base cases are written for → but in array we can't go to negative index

shifting of index

-1 0 1 2 3 . . . . n-1

f(0) f(1) f(n,m)

Shifting it to right by 1

so the base case

if (i<0 || j<0) return 0

will become

if (i==0 || j==0) return 0

```javascript
function f(i, j, s1, s2, dp) {
    if(i == 0 || j == 0) {
        return 0
    }

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

    if(s1[i-1] == s2[j-1]) {
        return dp[i][j] = 1 + f(i-1, j-1, s1, s2, dp);
    }

    return dp[i][j] = Math.max(f(i-1, j, s1, s2, dp), f(i, j-1, s1, s2, dp));
}

function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n+1).fill().map(() => Array(m+1).fill(-1));
    return f(n,m,s1,s2,dp);
}

const s1 = "adebc";
const s2 = "dcadb";
lcs(s1, s2); //? 3
```

recursion

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n + 1)
        .fill()
        .map(() => Array(m + 1).fill(-1));

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

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i-1][j-1];
            } else {
                dp[i][j] = Math.max(dp[i-1][j], dp[i][j]);
            }
        }
    }

    return dp[n][m];
}

const s1 = "adebc";
const s2 = "dcadb";
lcs(s1, s2); //? 3
```

Tabulation

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);
    let cur = Array(n+1).fill(0);

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

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                cur[j] = 1 + prev[j-1];
            } else {
                cur[j] = Math.max(prev[j], cur[j]);
            }
        }
        prev = [...cur];
    }

    return prev[m];
}

const s1 = "adebc";
const s2 = "dcadb";
lcs(s1, s2); //? 3
```

space optimization

lec27 → print longest common subsequence

j →		0	1	2	3	4	5
			b	d	g	e	k
i ↓	0	0	0	0	0	0	0
	1 a	0	0	0	0	0	0
	2 b	0	1	1	1	1	1
i-1, j-1	3 c	0	1	1	1	1	1
	4 d	0	1	2	2	2	2
	5 e	0	1	2	2	3	3

max, max

(n, m)

abcde

bdqek

b	d	e

len = dp[n][m], index = len-1

s = '', for (i=0 to len-1) s += '$'

i = n ; j = m

while (i>0 && j>0)

{ if (s1[i-1] == s2[j-1])

{

s[index] = s1[i-1];

index--

i--; j--;

}

else if (dp[i-1][j] > dp[i][j-1])

{ i = i-1;

}

else

{ j = j-1;

}

}

if: s1[i-1] == s2[j-1]

dp[i][j] = 1 + dp[i-1][j-1]

else

dp[i][j] = max(dp[i-1][j], dp[i][j-1])

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n + 1)
        .fill()
        .map(() => Array(m + 1).fill(-1));

    for (let j = 0; j <= m; j++) dp[0][j] = 0;
    for (let i=0; i<=n; i++) dp[i][0] =0
    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i-1][j-1];
            } else {
                dp[i][j] = Math.max(dp[i-1][j], dp[i][j]);
            }
        }
    }

    let len = dp[n][m];
    let index = len-1;
    let s = [];
    for(let i=0; i<len; i++) s[i]= '$';

    let i=n; let j=m;
    while(i>0 && j>0) {
        if(s1[i-1] == s2[j-1]) {
            s[index] = s1[i-1];
            index--;
            i--;
            j--;
        }
        else if(dp[i-1][j] > dp[i][j-1]) {
            i = i-1;
        }
        else {
            j = j-1;
        }
    }
    return s.join('');
}

const s1 = "adebc";
const s2 = "dcadb";
lcs(s1, s2); //? adb
```

lec 28. longest common substring (not subsequence)

Longest Common Substring

Contributed by Deep Mavani
Last Updated: 23 Feb, 2023

Medium | 0/80 | Avg time to solve 25 mins | Success Rate 75 %

Share | 53 upvotes

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.
For Example :

```
If 'STR1' = "abcjklp" and 'STR2' = "acjkp"  then the output
will be 3.

Explanation: The longest common substring is "cjk" which is
of length 3.
```

s1 = abcd
s2 = abzd

		j →		a	b	z	d
				0	1	2	3	4
	i ↓							
		0	0	0	0	0	0
a	1		0	1	0	0	0
b	2		0	0	2	0	0
c	3		0	0	0	0	0
d	4		0	0	0	0	1

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n + 1)
        .fill()
        .map(() => Array(m + 1).fill(-1));

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

    let ans = 0;

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i - 1][j - 1];
                ans = Math.max(ans, dp[i][j]);
            } else {
                dp[i][j] = 0;
            }
        }
    }

    return ans;
}

const s1 = "abcd";
const s2 = "abzd";
lcs(s1, s2); //? 2
```

tabulation

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);
    let cur = Array(m+1).fill(0);

    let ans = 0;

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                cur[j] = 1 + prev[j - 1];
                ans = Math.max(ans, cur[j]);
            } else {
                cur[j] = 0;
            }
        }
        prev = [...cur];
    }

    return ans;
}

const s1 = "abcd";
const s2 = "abzd";
lcs(s1, s2); //? 2
```

space optimization

---

lec29 – longest palindromic subsequence

palindrome → if it reads same way forward & backward

		palindrome
s = bbbab →	ab	✗
	bb	✓
	bbb	✓
	bbbb	✓ → 4
	bab	✓

M1 → generate all subsequences, & check for palindrome & take the longest

M2 → s1 → given input
s2 → reverse s1
call lcs(s1, s2)

lec-26

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n+1).fill().map(() => Array(m+1).fill(0));

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i-1][j-1];
            } else {
                dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }

    return dp[n][m];
}

function longestPalindrome(s1) {
    let s2 = s1.split('').reverse().join('');
    return lcs(s1, s2);
}

const s = "bbbab";
longestPalindrome(s); //? 4
```

Tabulation

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);
    let cur = Array(m+1).fill(0);

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                cur[j] = 1 + prev[j-1];
            } else {
                cur[j] = Math.max(prev[j], cur[j-1]);
            }
        }
        prev = [...cur];
    }

    return prev[m];
}

function longestPalindrome(s1) {
    let s2 = s1.split('').reverse().join('');
    return lcs(s1, s2);
}

const s = "bbbab";
longestPalindrome(s); //? 4
```

space optimized

---

lec 30 → min insertion to make string palindrome

you can insert any chars anywhere

s = abcaa

abcaa aacba

reverse of i/p

max operations = len(s)

s = abcaa

↓

aabcbaa ② insertions

How do you approach?

keep the longest palindromic portion intact

ex-1 abcaa len ⇒ 5

longest palindrome length ⇒ 3

min insertion ⇒ 5-3 = 2

ex2 coding ninja

length ⇒ 11

palindrome length ⇒ 11-5 = 6

abcacba

codajn ingninjadoc

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);
    let cur = Array(m+1).fill(0);

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                cur[j] = 1 + prev[j-1];
            } else {
                cur[j] = Math.max(prev[j], cur[j-1]);
            }
        }
        prev = [...cur];
    }

    return prev[m];
}

function longestPalindrome(s1) {
    let s2 = s1.split('').reverse().join('');
    return lcs(s1, s2);
}

function minInsertion(s) {
    let n = s.length;
    let palindromeLength = longestPalindrome(s);
    return n - palindromeLength;
}

const s = "codingninja";
minInsertion(s); //? 6
```

space optimized

---

lec31 → min insertions/deletions to convert string A to string B

s1 = "abcd"
s2 = "anc"

convert s1 to s2

M1
① delete everything from s1 → 4 operations
② insert everything from s2 to s1 → 3 "

total = 7 operations

max operations = n + m
↑ deletions ↑ insertions

a ~~b~~ c ~~d~~ → 2 deletions → ac → 1 insert → a n c

ac → longest common subsequence

lcs = 2
n = 4
m = 3

ans ⇒ 4 − 2 + 3 − 2
= 7 − 4
= 3

⇒ n + m − lcs × 2

```javascript
function lcs(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n+1).fill().map(() => Array(m+1).fill(0));

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i-1][j-1];
            } else {
                dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }

    return dp[n][m];
}

function convert(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let len = lcs(s1,s2);
    return (m+n) - 2* len;
}

const s1 = "abcd";
const s2 = "anc";
convert(s1, s2); //3
```

tabulation

lec 32 → shortest common super sequence

S1 = "brute" S2 = "groot"

"brutegroot" length = 10

"bgruoote" length = 8 → shortest

S1 = bleed S2 = blue

ans → bleued

brute groot

bgruoote

traverse both string if char matches take it once, else take both

$$\boxed{\text{operations} = n + m - \text{len}(\text{lcs})}$$

string = ?

log[m+1][n+1]

i ↓ / j →		g	r	o	o	t
		0	1	2	3	4	5
	0	0	0	0	0	0	0
b	1	0	0	0	0	0	0
r	2	0	0	1	1	1	1
u	3	0	0	1	1	1	1
t	4	0	0	1	1	1	2
e	5	0	0	1	1	1	2

i ↓ / j →		g	r	o	o	t
		0	1	2	3	4	5
	0	0	0	0	0	0	0
b	1	0	0	0	0	0	0
r	2	0	0	1	1	1	1
u	3	0	0	1	1	1	1
t	4	0	0	1	1	1	2
e	5	0	0	1	1	1	2

this will be taken care outside while loop in another while loop

te

ctoudrbg

gbroute

len → 8

```javascript
function shortestSupersequence(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n + 1)
        .fill()
        .map(() => Array(m + 1).fill(0));

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] == s2[j - 1]) {
                dp[i][j] = 1 + dp[i - 1][j - 1];
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    let i=n; j=m;
    let ans = '';

    while(i>0 && j>0) {
        if(s1[i-1] === s2[j-1]) {
            ans += s1[i-1];
            i--;
            j--;
        } else if(dp[i-1][j] > dp[i][j-1]) {
            ans += s1[i-1];
            i--;
        } else {
            ans += s2[j-1];
            j--;
        }
    }

    while(i>0) {
        ans += s1[i-1];
        i--;
    }

    while(j>0) {
        ans += s2[j-1];
        j--;
    }

    return ans.split('').reverse().join('');
}

const s1 = "brute";
const s2 = "groot";
shortestSupersequence(s1, s2); //bgruoote
```

only one will run

tabulation

Shortest Common Supersequence

TA Contributed by **TanishkTonk**
Last Updated: 23 Feb, 2023

Hard 0/120

Share 50 upvotes

Problem Statement Suggest Edit

Given two strings, 'A' and 'B'. Return the shortest supersequence string 'S', containing both 'A' and 'B' as its subsequences. If there are multiple answers, return any of them.
Note: A string 's' is a subsequence of string 't' if deleting some number of characters from 't' (possibly 0) results in the string 's'.
For Example:

```
Suppose 'A' = "brute", and 'B' = "groot"

The shortest supersequence will be "bgruoote". As shown
below, it contains both 'A' and 'B' as subsequences.

A   A A     A A
b g r u o o t e
  B B   B B B

It can be proved that the length of supersequence for this
input cannot be less than 8. So the output will be bgruoote.
```

---

lec 33 → distinct subsequences (1D array optimisation technique.

concept → string matching

s1 = babgbag s2 = bag

ans = 5

babgbag
babgbag
babgbag
babgbag
babgbag

Try out all possible ways

Recursion

1. express everything in (i, j)
2. explore all possibilities on that index
3. return summation of all possibilities

s1 = babgbag (n-1) (n)

s2 = bag (m-1) (m)

start call

f(n-1, m-1) → number of distinct subsequences of s2[0.....j] in s1[0.....i]

```
f(i,j) {
    if(j<0) return 1;
    if(i<0) return 0;

    if (s1[i] == s2[j])
        return f(i-1, j-1) + f(i-1, j)
    else
        return f(i-1, j)
}
```

TC → exponential

$s1 \to 2^n$

$s2 \to 2^m$

SC → $O(N+M)$

memoization

changing params → i j

dp[N][M]

TC → $O(N \times M)$

SC → $O(N \times M) + O(N+M)$

AS

auxiliary stack space

base case

babgbag bag

i = -1 j

or j = -1

↳ everything is matched

if all chars of s2 matched, return 1

if j>0 & i<0 → match not found return 0

115. Distinct Subsequences

Hard 5.2K 199

Apple Google Amazon ...

Given two strings `s` and `t`, return *the number of distinct* **subsequences** *of* `s` *which equals* `t`.

The test cases are generated so that the answer fits on a 32-bit signed integer.

Example 1:

```
Input: s = "rabbbit", t = "rabbit"
Output: 3
Explanation:
As shown below, there are 3 ways you can generate "rabbit" from s.
rabbbit
rabbbit
rabbbit
```

```javascript
function f(i,j,s1,s2) {
    if(j<0) return 1;
    if(i<0) return 0;

    if(s1[i] === s2[j]) {
        return f(i-1, j-1, s1, s2) + f(i-1, j, s1, s2);
    } else {
        return f(i-1, j, s1, s2);
    }
}

var numDistinct = function (s1, s2) {
    let n = s1.length;
    let m = s2.length;
    return f(n-1,m-1,s1,s2)
};

const s1 = "babgbag";
const s2 = "bag";
numDistinct(s1, s2); //?5
```

Recursion

```javascript
function f(i, j, s1, s2, dp) {
    if (j < 0) return 1;
    if (i < 0) return 0;

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

    if (s1[i] === s2[j]) {
        return (dp[i][j] = f(i - 1, j - 1, s1, s2, dp) + f(i - 1, j, s1, s2, dp));
    } else {
        return (dp[i][j] = f(i - 1, j, s1, s2, dp));
    }
}

var numDistinct = function (s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n)
        .fill()
        .map(() => Array(m).fill(-1));
    return f(n - 1, m - 1, s1, s2, dp);
};

const s1 = "babgbag";
const s2 = "bag";
numDistinct(s1, s2); //?5
```

Memoization

Tabulation

bottom-up

1. declare dp array
2. write base cases
3. write loops for changing params
4. copy the recurrence

dp[N][M]

shifting the index by 1 to right

1. dp[N+1][M+1]
2. for (i=0 to n) dp[i][0] = 1
   for (j=0 to m) dp[0][j] = 0
3. for (i=1 to n)
   for (j=1 to m)
4. if(s1[i-1] == s2[j-1])
   dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
   else
   dp[i][j] = dp[i-1][j]

return dp[n][m]

```javascript
var numDistinct = function (s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n+1)
        .fill()
        .map(() => Array(m+1).fill(0));

    for(let i=0; i<=n; i++) dp[i][0] = 1;
    // for(let j=1; j<=m; j++) dp[0][j] = 0;
    ↳ because dp is initialized with 0 already
    for(let i=1; i<=n; i++) {
        for(let j=1; j<=m; j++) {
            if(s1[i-1] === s2[j-1]) {
                dp[i][j] = dp[i-1][j-1] + dp[i-1][j]
            } else {
                dp[i][j] = dp[i-1][j];
            }
        }
    }

    return dp[n][m];
};

const s1 = "babgbag";
const s2 = "bag";
numDistinct(s1, s2); //? 5
```

Tabulation

```javascript
var numDistinct = function (s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);
    let cur = Array(m+1).fill(0);

    prev[0] = cur[0] = 1;

    for(let i=1; i<=n; i++) {
        for(let j=1; j<=m; j++) {
            if(s1[i-1] === s2[j-1]) {
                cur[j] = prev[j-1] + prev[j]
            } else {
                cur[j] = prev[j];
            }
        }
        prev = [...cur]
    }

    return prev[m];
};

const s1 = "babgbag";
const s2 = "bag";
numDistinct(s1, s2); //? 5
```

space optimized

```javascript
var numDistinct = function (s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);

    prev[0] =  1;

    for(let i=1; i<=n; i++) {
        for(let j=m; j>=1; j--) {
            if(s1[i-1] === s2[j-1]) {
                prev[j] = prev[j-1] + prev[j]
            }
        }
    }

    return prev[m];
};

const s1 = "babgbag";
const s2 = "bag";
numDistinct(s1, s2); //?5
```

1D - optimized

lec 34 → Edit Distance | 1D Array optimised

Problem Statement Suggest Edit

You are given two strings 'S' and 'T' of lengths 'N' and 'M' respectively. Find the "Edit Distance" between the strings.
Edit Distance of two strings is the minimum number of steps required to make one string equal to the other. In order to do so, you can perform the following three operations:

```
1. Delete a character
2. Replace a character with another one
3. Insert a character
```

Note:

```
Strings don't contain spaces in between.
```

Sample Input 1 :

```
abc
dc
```

Sample Output 1 :

```
2
```

Explanation For Sample Input 1 :

```
In 2 operations we can make the string T to look like string
S. First, insert the character 'a' to string T, which makes
it "adc".

And secondly, replace the character 'd' of the string T with
'b' from the string S. This would make string T to "abc"
which is also the string S. Hence, the minimum distance.
```

1. insert
2. remove
3. replace

M1 → delete all from s1 → n
insert all from s2 to s1 → m
total operations (n+m)

$f(i,j)$

$f(n-1, m-1)$ → find min operations to convert $s1[0 \ldots i]$ to $s2[0 \ldots j]$

intuition

String Matching

horse
ros

→ insert of the same character
→ delete & try finding somewhere else
→ replace & match

insertion

horse s — no match so insert s
ros

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

after insertion j matches with the inserted char, so $j = j-1$ & i stayed same

replace

horse
ros

doesn't match, update e to s
$i = i-1$
$j = j-1$

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

remove/delete

horse
ros

doesn't match, delete e,
$i = i-1$
$j = j$

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

matches.

$0 + f(i-1, j-1)$

Base cases

i or j exhausted

s1 get exhausted

we need to insert remaining chars from s2

s2 get exhausted

if means match already found

s1 = "hos" s2 = ""

to convert s1 to s2

min 8 delete operations required

```
f(i,j){
    if(i<0) return j+1;
    if(j<0) return i+1;
    if(s1[i]==s2[j]) return 0+f(i-1,j-1)
    return Math.min(1+f(i,j-1)
                    1+f(i-1,j),
                    1+f(i-1,j-1))
}
```

TC → exponential

SC → O(N+M)

Memoization

Changing params i j

N M

dp[N][M]

TC → O(N×M)

SC → O(N×M) + O(N+M)

ASS

```
function f(i, j, s1, s2) {
    if (i < 0) return j+1;
    if (j < 0) return i+1;

    if (s1[i] === s2[j]) {
        return 0 + f(i-1, j-1, s1, s2)
    } else {
        return Math.min((1+f(i, j-1, s1, s2)), (1+f(i-1, j, s1, s2)), (1+f(i-1, j-1, s1, s2)));
    }
}

function editDistance(s1, s2) {
    let n = s1.length;
    let m = s2.length;

    return f(n-1, m-1, s1, s2);
};

const s1 = "abc";
const s2 = "dc";
editDistance(s1, s2); //? 2
```

Recursion

```
function f(i, j, s1, s2, dp) {
    if (i < 0) return j + 1;
    if (j < 0) return i + 1;

    if (s1[i] === s2[j]) {
        return (dp[i][j] = 0 + f(i - 1, j - 1, s1, s2, dp));
    } else {
        return (dp[i][j] = Math.min(
            1 + f(i, j - 1, s1, s2, dp),
            1 + f(i - 1, j, s1, s2, dp),
            1 + f(i - 1, j - 1, s1, s2, dp)
        ));
    }
}

function editDistance(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n)
        .fill()
        .map(() => Array(m).fill(-1));

    return f(n - 1, m - 1, s1, s2, dp);
}

const s1 = "abc";
const s2 = "dc";
editDistance(s1, s2); //? 2
```

Memoization

Tabulation

```javascript
function editDistance(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n+1)
        .fill()
        .map(() => Array(m+1).fill(0));

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

    for(let i=1; i<=n; i++) {
        for(let j=1; j<=m; j++) {
            if (s1[i-1] === s2[j-1]) {
                dp[i][j] = 0 + dp[i - 1][j-1];
            } else {
                dp[i][j] = Math.min(
                    1 + dp[i][j-1],
                    1 + dp[i - 1][j],
                    1 + dp[i - 1][j-1]
                );
            }
        }
    }

    return dp[n][m];
}

const s1 = "abc";
const s2 = "dc";
editDistance(s1, s2); //? 2
```

Tabulation

Shifting index to right by 1

```javascript
function f(i, j, s1, s2, dp) {
    if (i ~~==~~ 0) return ~~j + 1~~; j
    if (j ~~==~~ 0) return ~~i + 1~~; i

    if (s1[~~i~~ i-1] === s2[~~j~~ j-1]) {
        return (dp[i][j] = 0 + f(i - 1, j - 1, s1, s2, dp));
    } else {
        return (dp[i][j] = Math.min(
            1 + f(i, j - 1, s1, s2, dp),
            1 + f(i - 1, j, s1, s2, dp),
            1 + f(i - 1, j - 1, s1, s2, dp)
        ));
    }
}

function editDistance(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(~~n~~ n+1)
        .fill()
        .map(() => Array(~~m~~ m+1).fill(-1));

    return f(~~n - 1~~ n, ~~m - 1~~ m, s1, s2, dp);
}
```

```javascript
function editDistance(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(0);
    let cur = Array(m+1).fill(0);

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

    for(let i=1; i<=n; i++) {
        cur[0] = i;
        for(let j=1; j<=m; j++) {
            if (s1[i-1] === s2[j-1]) {
                cur[j] = 0 + prev[j-1];
            } else {
                cur[j] = Math.min(
                    1 + cur[j-1],
                    1 + prev[j],
                    1 + prev[j-1]
                );
            }
        }
        prev = [...cur]
    }

    return prev[m];
}

const s1 = "abc";
const s2 = "dc";
editDistance(s1, s2); //? 2
```

space optimized

lec 35 → wildcard matching

s1 = "?ay" true
s2 = "say"

s1 = ~~abc*cd~~
s2 = ~~a~~ ~~b~~ de ~~f~~ ~~c~~ ~~d~~ true

s1 = **abcd true
s2 = abcd

s1 = ab?d false
s2 = abcc

? → matches with single character
* → matches with sequence of length 0 or more

input
s1 → lowercase, ?, *
s2 → lowercase
question → check whether s1 matches with s2 or not

TC → exponential
SC → O(N+M) ASS

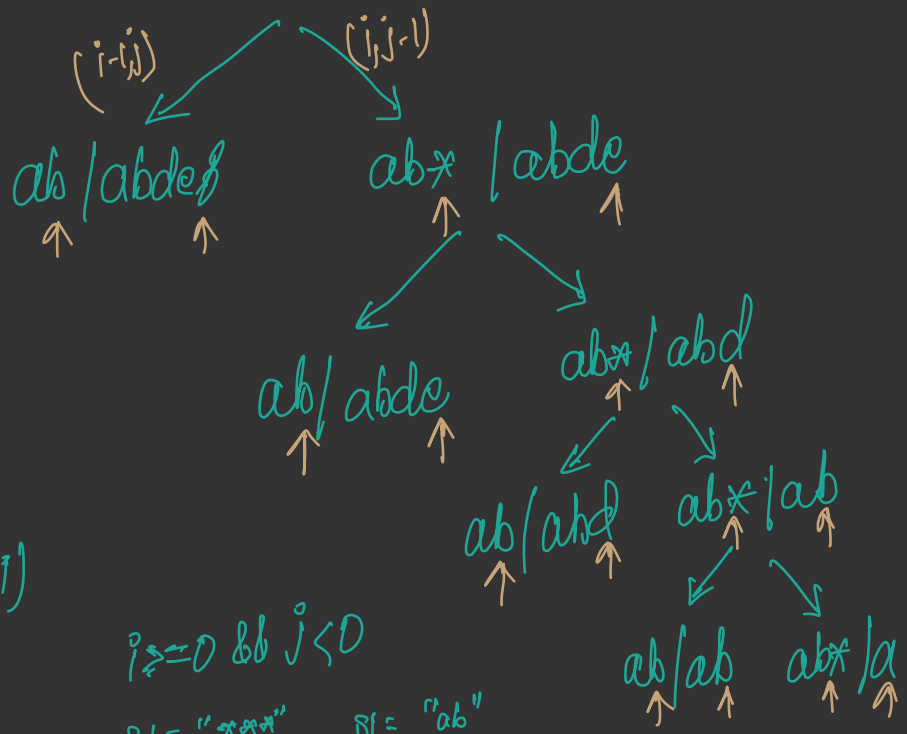
Recussion

① express in (i, j)
② explore all comparisons
③ out of all comparisons, if anyone can, return true

```
f(i,j)
{ if(i<0 && j<0) return true;
  if(i<0 && j>=0) return false;
  if(i>=0 && j<0) code ...
  if(s1[i] == s2[j] || s1[i] == '?')
      return f(i-1, j-1)
  if(s1[i] == '*')
      return f(i-1, j) or f(i, j-1)
  return false;
}
```

for * case

ex ab* abdef



i>=0 && j<0

s1 = "***"
s2 = " " true

s1 = "ab"
s2 = " " false

Memoization

i j

n-1 m-1

dp[N][M]

TC → O (N×M)

SC → O (N×M) + O (N+M)

```
function f(i, j, s1, s2) {
    if (i < 0 && j<0) return true;
    if (i < 0 && j>=0) return false;
    if(i>=0 && j<0) {
        for(let x=i; x>=0; x--) {
            if(s1[x] != '*') return false;
        }
        return true;
    }

    if(s1[i] === s2[j] || s1[i] == '?') return f(i-1, j-1, s1, s2);
    if(s1[i] === '*') return f(i-1, j, s1, s2) || f(i, j-1, s1, s2);

    return false;
}

function wildcardMatching(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    return f(n-1, m-1, s1, s2);
}

const s1 = "*ay";
const s2 = "ray";
wildcardMatching(s1, s2); //? true
```

Recursion

```
function f(i, j, s1, s2, dp) {
    if (i < 0 && j<0) return true;
    if (i < 0 && j>=0) return false;
    if(i>=0 && j<0) {
        for(let x=i; x>=0; x--) {
            if(s1[x] != '*') return false;
        }
        return true;
    }

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

    if(s1[i] === s2[j] || s1[i] == '?') return dp[i][j] = f(i-1, j-1, s1, s2, dp);
    if(s1[i] === '*') return dp[i][j] = f(i - 1, j, s1, s2, dp) || f(i, j - 1, s1, s2, dp);

    return dp[i][j] = false;
}

function wildcardMatching(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n).fill().map(() => Array(m).fill(-1));
    return f(n-1, m-1, s1, s2, dp);
}

const s1 = "*ay";
const s2 = "ray";
wildcardMatching(s1, s2); //? true
```

memoization

Tabulation

```
function wildcardMatching(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n + 1)
        .fill()
        .map(() => Array(m + 1).fill(false));

    dp[0][0] = true;
    for (let j = 1; j <= m; j++) dp[0][j] = false;
    for (let i = 1; i <= n; i++) {
        let flag = true;
        for (let x = i; x >= 1; x--) {
            if (s1[x - 1] != "*") flag = false;
        }
        dp[i][0] = flag;
    }

    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] === s2[j - 1] || s1[i - 1] == "?")
                dp[i][j] = dp[i-1][j-1];
            else if (s1[i - 1] === "*")
                dp[i][j] =
                    dp[i-1][j] || dp[i][j-1];
            else {
                dp[i][j] = false;
            }
        }
    }

    return dp[n][m];
}

const s1 = "*ay";
const s2 = "ray";
wildcardMatching(s1, s2); //? true
```

Tabulation

```
function f(i, j, s1, s2, dp) {
    if (i == 0 && j == 0) return true;
    if (i == 0 && j > 0) return false;
    if (i > 0 && j == 0) {
        for (let x = i; x >= 1; x--) {
            if (s1[x - 1] != "*") return false;
        }
        return true;
    }

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

    if (s1[i - 1] === s2[j - 1] || s1[i - 1] == "?")
        return (dp[i][j] = f(i - 1, j - 1, s1, s2, dp));
    if (s1[i - 1] === "*")
        return (dp[i][j] = f(i - 1, j, s1, s2, dp) || f(i, j - 1, s1, s2, dp));

    return (dp[i][j] = false);
}

function wildcardMatching(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let dp = Array(n + 1)
        .fill()
        .map(() => Array(m + 1).fill(-1));
    return f(n, m, s1, s2, dp);
}

const s1 = "*ay";
const s2 = "ray";
wildcardMatching(s1, s2); //? true
```

1 based indexing

```cpp
function wildcardMatching(s1, s2) {
    let n = s1.length;
    let m = s2.length;
    let prev = Array(m+1).fill(false);
    let cur = Array(m+1).fill(false);

    prev[0] = true;
    for (let j = 1; j <= m; j++) prev[j] = false;

    for (let i = 1; i <= n; i++) {
        // cur is the current's row's column
        // and that cur's 0th row has to be assigned everytime
        let flag = true;
        for (let x = i; x >= 1; x--) {
            if (s1[x - 1] != "*") flag = false;
        }
        cur[0] = flag;

        for (let j = 1; j <= m; j++) {
            if (s1[i - 1] === s2[j - 1] || s1[i - 1] == "?")
                cur[j] = prev[j-1];
            else if (s1[i - 1] === "*")
                cur[j] =
                    prev[j] || cur[j-1];
            else {
                cur[j] = false;
            }
        }

        prev = [...cur];
    }

    return prev[m];
}

const s1 = "*ay";
const s2 = "ray";
wildcardMatching(s1, s2); //?true
```

space optimized

---

lec 86 best time to buy and sell stocks | DP on stocks

arr[] → [7, 1, 5, 3, 6, 4] n = 6

buy → 1
sell → 6
profit ⇒ 5

intuition
if you are selling on ith day, you buy on the min price from 1st → (i-1)

code

```
min = a[0], profit = 0
for (i = 1; i < n; i++)
  { cost = a[i] - min;
    profit = max (profit, cost);
    min = min( min, a[i])
  }
return profit;
```

T.C → O(N)
S.C → O(1)

```cpp
function maxProfit(arr) {
    let n = arr.length;
    let min = arr[0];
    let profit = 0;

    for(let i=1; i<n; i++){
        let cost = arr[i] - min;
        profit = Math.max(profit, cost);
        min = Math.min(min, arr[i]);
    }

    return profit;
}

const arr = [7,1,5,3,6,4]
maxProfit(arr); //?5
```

lec37 → Buy & sell stock II

arr[] → [7, 1, 5, 3, 6, 4] n = 6

buy & sell as many times as you want

Problem Statement Suggest Edit

You have been given stock values/prices for N number of days. Every i-th day signifies the price of a stock on that day. Your task is to find the maximum profit which you can make by buying and selling the stocks.

Note :

```
You may make as many transactions as you want but can not
have more than one transaction at a time i.e, if you have the
stock, you need to sell it first, and then only you can buy
it again.
```

Detailed explanation (Input/output format, Notes, Images)

Sample Input 1 :

```
1
7
1 2 3 4 5 6 7
```

Sample Output 1 :

```
6
```

Explanation :

```
We can make the maximum profit by buying the stock on the
first day and selling it on the last day.
```

Try out all ways.

① Recursion

1. express everything in terms of (ind, buy) a flag to see if we can buy current stock or not
2. explore possibilities on that day
3. take the max all profits made

④ Base case

0 to n-1 n is out of bound

buying → -price → -7

selling → +price → +5-7 ⇒ -2

```
f(i,j) { if(ind==n)
            return 0;
```

```
if(buy) {
    profit = Max( (-prices[ind] + f(ind+1,0)), (0 + f(ind+1,1)) )
}
else {
    profit = Max( (prices[ind] + f(ind+1,1)), (0 + f(ind+1,0)) )
}
return profit;
```

take notTake

TC → 2^n

SC → $O(N)$

Memoization

ind buy

dp [N] [2]

```
function f(ind, arr, n, buy) {
    if(ind === n) return 0;
    let profit = 0;
    if(buy) {
        profit = Math.max((-arr[ind] + f(ind+1, arr, n, 0)), (0 + f(ind+1, arr, n, 1)))
    } else {
        profit = Math.max((arr[ind] + f(ind + 1, arr, n, 1)), (0 + f(ind+1, arr, n, 0)));
    }
    return profit;
}

function getMaximumProfit(arr) {
    let n = arr.length;
    return f(0, arr, n, 1);
}

const arr = [1,2,3,4,5,6,7]
getMaximumProfit(arr); //?6
```

Recursion

```
function f(ind, arr, n, buy, dp) {
    if(ind === n) return 0;
    if(dp[ind][buy] != -1) return dp[ind][buy];

    let profit = 0;
    if(buy) {
        profit = Math.max((-arr[ind] + f(ind+1, arr, n, 0, dp)), (0 + f(ind+1, arr, n, 1, dp)))
    } else {
        profit = Math.max((arr[ind] + f(ind + 1, arr, n, 1, dp)), (0 + f(ind+1, arr, n, 0, dp)));
    }
    return dp[ind][buy] = profit;
}

function getMaximumProfit(arr) {
    let n = arr.length;
    let dp = Array(n).fill().map(() => Array(2).fill(-1));
    return f(0, arr, n, 1, dp);
}

const arr = [1,2,3,4,5,6,7]
getMaximumProfit(arr); //?6
```

Memoization

TC → $O(N \times 2)$

SC → $O(N \times 2) + O(N+2)$

RBS

Tabulation

dp [N+1] [2]

base case

dp[n][0] = 0

dp[n][1] = 0

loops

i → n-1 to 0

buy → 0 to 1 or 1 to 0

```
function getMaximumProfit(arr) {
    let n = arr.length;
    let dp = Array(n+1).fill().map(() => Array(2).fill(0));

    dp[n][0] = dp[n][1] = 0;

    for(let ind=n-1; ind>=0; ind--) {
        for(let buy=0; buy<=1; buy++){
        let profit = 0;
        if (buy) {
            profit = Math.max(
                -arr[ind] + dp[ind+1][0],
                0 + dp[ind+1][1]
            );
        } else {
            profit = Math.max(
                arr[ind] + dp[ind+1][1],
                0 + dp[ind+1][0]
            );
        }
        dp[ind][buy] = profit;
        }
    }

    return dp[0][1];
}

const arr = [1,2,3,4,5,6,7]
getMaximumProfit(arr); //?6
```

Tabulation

```
function getMaximumProfit(arr) {
    let n = arr.length;
    let ahead = Array(2).fill(0);
    let cur = Array(2).fill(0);

    ahead[0] = ahead[1] = 0;

    for (let ind = n - 1; ind >= 0; ind--) {
        for (let buy = 0; buy <= 1; buy++) {
            let profit = 0;
            if (buy) {
                profit = Math.max(-arr[ind] + ahead[0], 0 + ahead[1]);
            } else {
                profit = Math.max(arr[ind] + ahead[1], 0 + ahead[0]);
            }
            cur[buy] = profit;
        }
        ahead = [...cur];
    }

    return ahead[1];
}

const arr = [1, 2, 3, 4, 5, 6, 7];
getMaximumProfit(arr); //? 6
```

space optimization

```javascript
function getMaximumProfit(arr) {
    let n = arr.length;
    let aheadBuy = 0, aheadNotBuy = 0;
    let curBuy, curNotBuy;

    for (let ind = n - 1; ind >= 0; ind--) {
        curBuy = Math.max(-arr[ind] + aheadNotBuy, 0 + aheadBuy);
        curNotBuy = Math.max(arr[ind] + aheadBuy, 0 + aheadNotBuy);

        aheadNotBuy = curNotBuy;
        aheadBuy = curBuy;
    }

    return aheadBuy;
}

const arr = [1, 2, 3, 4, 5, 6, 7];
getMaximumProfit(arr); //? 6
```

space optimization

lec 38 → buy & sell stocks III

prices = [3, 3, 5, 0, 0, 3, 1, 4]

ex-1 2 3 ⇒ 5

ex-2 3 3 ⇒ 6

max Profit = 6.

max 2 transactions ← initial call

f(0, 1, 2)

f(ind, buy, cap)

{ if (ind == 0 || cap == 0) return 0

let profit = 0;

if (buy)

profit = Max(−price[ind] + f(ind+1, 0, cap), 0 + f(ind+1, 1, cap))

else

profit = Max(price[ind] + f(ind+1, 1, cap−1), 0 + f(ind+1, 0, cap))

return profit;

}

TC → exponential

SC → O(N)

ASS

Problem Statement Suggest Edit

You are Harshad Mehta's friend. He told you the price of a particular stock for the next 'N' days. You can either buy or sell a stock. Also, you can only complete at most 2-transactions. Find the maximum profit that you can earn from these transactions. You can take help from Mr. Mehta as well.

Note

1. Buying a stock and then selling it is called one transaction.
2. You are not allowed to do multiple transactions at the same time. This means you have to sell the stock before buying it again.

buy = 1 → we can buy a stock today
if we buy, We will invest money so money will be reduced from profit
if we are not buying today, we can buy tomorrow. profit remain same

buy = 0 → we can sell today the previously bought stock
if sell, profit will increase & we can buy tomorrow
if not sell, profit remain same & we can not buy tommorow
if we sell, 1 transaction will complete & now cap = cap−1

memoization

ind	buy	cap
0 to n-1	0/1	0/1/2
N	2	3

dp[N][2][5]

TC -> O(N×2×3)

SC -> O(N×2×3) + O(N) ASS

```
function f(ind, buy, cap, arr, n) {
    if (ind === n || cap === 0) return 0;

    let profit = 0;
    if (buy) {
        profit = Math.max(-arr[ind] + f(ind + 1, 0, cap, arr, n),
                          0 + f(ind + 1, 1, cap, arr, n));
    } else {
        profit = Math.max(arr[ind] + f(ind + 1, 1, cap-1, arr, n),
                          0 + f(ind + 1, 0, cap, arr, n));
    }
    return profit;
}

function maxProfit(arr) {
    let n = arr.length;
    return f(0, 1, 2, arr, n);
}

const arr = [3, 3, 5, 0, 3, 1, 4];
maxProfit(arr); //?6
```

Recursion

```
function f(ind, buy, cap, arr, n, dp) {
    if (ind === n || cap === 0) return 0;

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

    let profit = 0;
    if (buy) {
        profit = Math.max(-arr[ind] + f(ind + 1, 0, cap, arr, n, dp),
                          0 + f(ind + 1, 1, cap, arr, n, dp));
    } else {
        profit = Math.max(arr[ind] + f(ind + 1, 1, cap-1, arr, n, dp),
                          0 + f(ind + 1, 0, cap, arr, n, dp));
    }
    return dp[ind][buy][cap] = profit;
}

function maxProfit(arr) {
    let n = arr.length;
    let dp = Array(n).fill().map(() => Array(2).fill().map(() => Array(3).fill(-1)));
    return f(0, 1, 2, arr, n, dp);
}

const arr = [3, 3, 5, 0, 3, 1, 4];
maxProfit(arr); //?6
```

Memoization

Tabulation

dp[N][2][3] = 0

Base cases

① cap==0 ind & buy can be anything

for (ind = 0 to n-1)
    for(buy = 0 to 1)
        dp[ind][buy][0] = 0

② ind==n buy & cap can be anything

for(buy = 0 to 1)
    for (cap = 0 to 2)
        dp[n][buy][cap] = 0

```
function maxProfit(arr) {
    let n = arr.length;
    let dp = Array(n+1).fill().map(() => Array(2).fill().map(() => Array(3).fill(0)));

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

    for(let buy=0; buy<=1; buy++){
        for(let cap=0; cap<=2; cap++){
            dp[n][buy][cap] = 0;
        }
    }

    for(let ind=n-1; ind>=0; ind--){
        for(let buy=0; buy<=1; buy++){
            for(let cap=1; cap<=2; cap++){
                let profit = 0;
                if (buy) {
                    profit = Math.max(
                        (-arr[ind] + dp[ind + 1][0][cap]),
                        0 + dp[ind + 1][1][cap]
                    );
                } else {
                    profit = Math.max(
                        arr[ind] + dp[ind + 1][1][cap - 1],
                        0 + dp[ind + 1][0][cap]
                    );
                }
                dp[ind][buy][cap] = profit;
            }
        }
    }

    return dp[0][1][2];
}

const arr = [3, 3, 5, 0, 3, 1, 4];
maxProfit(arr); //?6
```

not necessary since array is initialized with 0

at cap=0, it's always 0 so start from 1

Tabulation

```javascript
function maxProfit(arr) {
    let n = arr.length;

    let after = Array(2).fill().map(() => Array(3).fill(0));
    let cur = Array(2).fill().map(() => Array(3).fill(0));

    for(let ind=n-1; ind>=0; ind--){
        for(let buy=0; buy<=1; buy++){
            for(let cap=1; cap<=2; cap++){
                let profit = 0;
                if (buy) {
                    profit = Math.max(
                        (-arr[ind] + after[0][cap]),
                        0 + after[1][cap]
                    );
                } else {
                    profit = Math.max(
                        arr[ind] + after[1][cap - 1],
                        0 + after[0][cap]
                    );
                }
                cur[buy][cap] = profit;
            }
            after = cur
        }
    }

    return after[1][2];
}

const arr = [3, 3, 5, 0, 3, 1, 4];
maxProfit(arr); //? 6
```

space optimized

---

lec 39 buy & sell stocks IV

atmost k transactions are allowed

prices = [3, 2, 6, 5, 0, 3] k=2

```javascript
function maxProfit(arr, k) {
    let n = arr.length;

    let after = Array(2).fill().map(() => Array(k+1).fill(0));
    let cur = Array(2).fill().map(() => Array(k+1).fill(0));

    for(let ind=n-1; ind>=0; ind--){
        for(let buy=0; buy<=1; buy++){
            for(let cap=1; cap<=k; cap++){
                let profit = 0;
                if (buy) {
                    profit = Math.max(
                        (-arr[ind] + after[0][cap]),
                        0 + after[1][cap]
                    );
                } else {
                    profit = Math.max(
                        arr[ind] + after[1][cap - 1],
                        0 + after[0][cap]
                    );
                }
                cur[buy][cap] = profit;
            }
            after = cur
        }
    }

    return after[1][k];
}

const arr = [3, 3, 5, 0, 3, 1, 4];
maxProfit(arr, 3); //? 8
```

lec 40 buy & sell stocks with cooldown

cooldown → can't buy right after sell

unlimited transactions

```
function f(ind, buy, arr, n, dp) {
    if (ind >= n) return 0;

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

    let profit = 0;
    if (buy) {
        profit = Math.max(-arr[ind] + f(ind + 1, 0, arr, n, dp),
                          0 + f(ind + 1, 1,arr, n, dp));
    } else {
        profit = Math.max(arr[ind] + f(ind + 2, 1, arr, n, dp),
                          0 + f(ind + 1, 0, arr, n, dp));
    }
    return dp[ind][buy] = profit;
}

function stockProfit(arr) {
    let n = arr.length;
    let dp = Array(n).fill().map(() => Array(2).fill(-1));
    return f(0, 1, arr, n, dp);
}

const arr = [4, 9, 0, 4, 10];
stockProfit(arr); //? 11
```

memoization

Problem Statement Suggest Edit

You are given a list of stock prices, 'prices'. Where 'prices[i]' represents the price on 'i'th day. Your task is to calculate the maximum profit you can earn by trading stocks if you can only hold one stock at a time. After you sell your stock on the 'i'th day, you can only buy another stock on 'i + 2' th day or later.

For Example:

```
You are given prices = {4, 9, 0, 4, 10}, To get maximum
profits you will have to buy on day 0 and sell on day 1 to
make a profit of 5, and then you have to buy on day 3  and
sell on day 4 to make the total profit of 11. Hence the
maximum profit is 11.
```

```
function stockProfit(arr) {
    let n = arr.length;
    let dp = Array(n + 2)
        .fill()
        .map(() => Array(2).fill(0));

    for (let ind = n - 1; ind >= 0; ind--) {
        for (let buy = 0; buy <= 1; buy++) {
            let profit = 0;
            if (buy) {
                profit = Math.max(
                    -arr[ind] + dp[ind + 1][0],
                    0 + dp[ind + 1][1]
                );
            } else {
                profit = Math.max(
                    arr[ind] + dp[ind + 2][1],
                    0 + dp[ind + 1][0]
                );
            }
            dp[ind][buy] = profit;
        }
    }

    return dp[0][1];
}

const arr = [4, 9, 0, 4, 10];
stockProfit(arr); //? 11
```

Tabulation

loc 41 → buy & sell stocks with transaction fee

unlimited transaction
but for every transaction a fee will be taken

prices = [1, 3, 2, 8, 4, 9]

7 − 2 = 5

5 − 2 = 3

5 + 3 = 8

fee = 2

from loc37

```
function f(ind, arr, n, buy, dp, fee
    if(ind === n) return 0;
    if(dp[ind][buy] != -1) return dp[ind][buy];

    let profit = 0;
    if(buy) {
        profit = Math.max((-arr[ind] + f(ind+1, arr, n, 0, dp)), (0 + f(ind+1, arr, n, 1, dp)))
    } else {
        profit = Math.max((arr[ind] + f(ind + 1, arr, n, 1, dp)), (0 + f(ind+1, arr, n, 0, dp)));
    }
    return dp[ind][buy] = profit;
}

function getMaximumProfit(arr) {
    let n = arr.length;
    let dp = Array(n).fill().map(() => Array(2).fill(-1));
    return f(0, arr, n, 1, dp);
}

const arr = [1,2,3,4,5,6,7]
getMaximumProfit(arr); //?
```

−fee

lec 42 → longest increasing subsequence (LIS pattern)

arr[] = [10, 9, 2, 5, 3, 7, 101, 18]

2, 3, 7, 18 → len 4

2, 3, 7, 101 → len 4

various subsequences → print all subsequences

① power set

② recursion

brute force

TC → 2^n

Recursion

1. express everything in term of (ind)
2. explore all possibilities
   take or not Take
3. take the max length b/w take & not Take

f(ind, prev) → index of prev element which was picked

f(0, -1) → start → length of LIS starting from 0th index, whose prev index is -1

```
f(ind, prev)
{   if(ind == n) return 0;
    if(dp[ind][prev+1] != -1) return dp...
    let notTakeLen = 0 + f(ind+1, prev)
    if(prev == -1 || arr[ind] > arr[prev])
        let takeLen = 1 + f(ind+1, ind)
    return max(notTakeLen, takeLen);
}   dp[ind][prev+1] =
```

TC → 2^n → exponential (for each item we are doing take & not take)

SC → O(N)

memoization

ind → 0 to n-1

prev → -1 to n-1

dp[N][N+1]

co-ordinate change (shifting the index to right by 1)

TC → O(N×N)

SC → O(N×N) + O(N)

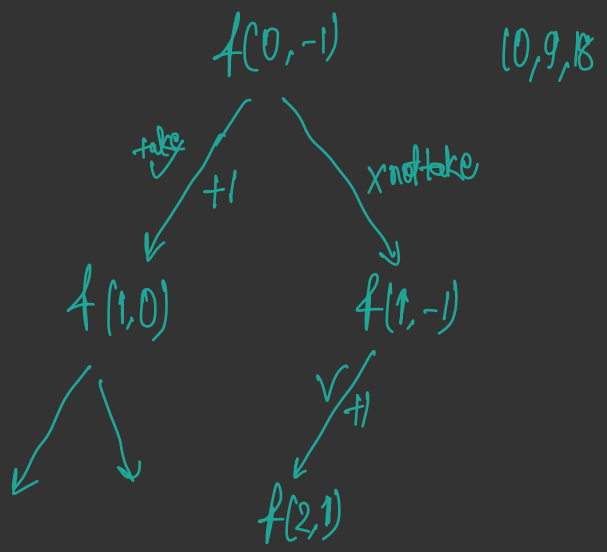
Problem Statement Suggest Edit

For a given array with N elements, you need to find the length of the longest subsequence from the array such that all the elements of the subsequence are sorted in strictly increasing order.

Strictly Increasing Sequence is when each term in the sequence is larger than the preceding term.

For Example:

```
[1, 2, 3, 4] is a strictly increasing array, while [2, 1, 4, 3] is not.
```




```javascript
function f(ind, prev, arr, n) {
    if (ind == n) return 0;

    let notTakeLen = 0 + f(ind+1, prev, arr, n);
    let takeLen=Number.MIN_SAFE_INTEGER;
    if(prev == -1 || arr[ind] > arr[prev]) {
        takeLen = 1 + f(ind+1, ind, arr, n);
    }
    return Math.max(notTakeLen, takeLen);
}

function longestIncreasingSubsequence(arr) {
    let n = arr.length;
    return f(0, -1, arr, n);
}

const arr = [5, 4, 11, 1, 16, 8];
longestIncreasingSubsequence(arr); //?3
```

Recursion

```javascript
function f(ind, prev, arr, n, dp) {
    if (ind == n) return 0;
    if(dp[ind][prev+1] != -1) return dp[ind][prev+1];

    let notTakeLen = 0 + f(ind+1, prev, arr, n, dp);
    let takeLen=Number.MIN_SAFE_INTEGER;
    if(prev == -1 || arr[ind] > arr[prev]) {
        takeLen = 1 + f(ind+1, ind, arr, n, dp);
    }
    return dp[ind][prev+1] = Math.max(notTakeLen, takeLen);
}

function longestIncreasingSubsequence(arr) {
    let n = arr.length;
    let dp = Array(n).fill().map(() => Array(n+1).fill(-1));
    return f(0, -1, arr, n, dp);
}

const arr = [5, 4, 11, 1, 16, 8];
longestIncreasingSubsequence(arr); //?3
```

Memoization

---

lec 43 → print longest increasing subsequence

previous question

Tabulation

① dp[n][n+1] = 0

② since dp is already initialized with 0, we dont need to write base case

③ changing params, in opposite direction

ind = n-1 to 0

prev = ind-1 to -1

④ copy the recurrence, follow co-ordinate shift

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

    for (let ind = n - 1; ind >= 0; ind--) {
        for (let prev = ind-1; prev >= -1; prev--) {
            let notTakeLen = 0 + dp[ind + 1][prev+1];
            let takeLen = Number.MIN_SAFE_INTEGER;
            if (prev == -1 || arr[ind] > arr[prev]) {
                takeLen = 1 + dp[ind+1][ind+1];
            }
            dp[ind][prev + 1] = Math.max(notTakeLen, takeLen);
        }
    }

    return dp[0][0];
}

const arr = [5, 4, 11, 1, 16, 8];
longestIncreasingSubsequence(arr); //? 3
```

tabulation

```javascript
function longestIncreasingSubsequence(arr) {
    let n = arr.length;
    let next = Array(n+1).fill(0);
    let cur = Array(n+1).fill(0);

    for (let ind = n - 1; ind >= 0; ind--) {
        for (let prev = ind-1; prev >= -1; prev--) {
            let notTakeLen = 0 + next[prev+1];
            let takeLen = Number.MIN_SAFE_INTEGER;
            if (prev == -1 || arr[ind] > arr[prev]) {
                takeLen = 1 + next[ind+1];
            }
            cur[prev + 1] = Math.max(notTakeLen, takeLen);
        }
        next = [...cur]
    }

    return next[0];
}

const arr = [5, 4, 11, 1, 16, 8];
longestIncreasingSubsequence(arr); //? 3
```

space optimization

```javascript
function longestIncreasingSubsequence(arr) {
    let n = arr.length;
    let dp = Array(n).fill(1);
    let max = 1;

    for(let i=0; i<n; i++){
        for(let prev=0; prev<i; prev++){
            if(arr[prev] < arr[i]) {
                dp[i] = Math.max(dp[i], 1 + dp[prev]);
            }
        }
        max = Math.max(max, dp[i])
    }

    return max;
}

const arr = [5, 4, 11, 1, 16, 8];
longestIncreasingSubsequence(arr); //? 3
```

→ TC → $O(N^2)$

SC → $O(N)$

it's same whether we take 5 or 4 & update 16 correspondent to 2

5	4	11	1	16	8

(5,8)

dp (initially every LIS is 1 length)

1	1	2 ~~1~~	1	~~1~~ 2 3	~~1~~ 2

hash (initially the index of each element)

0	1	0 ~~2~~	3	~~0~~ ~~4~~ 2	5

max value in dp → 3

index = 4

value → val[4] 16 → val[2] → val[0]

hash → 16 → 11 → 5

0 → index of 5 (5, 11)

0 → index of 5 (5, 16)

2 → index of 11 (11, 16)

```javascript
function longestIncreasingSubsequence(arr) {
    let n = arr.length;
    let dp = Array(n).fill(1);
    let hash = Array(n);
    let max = 1;
    let lastIndex = 0;

    for(let i=0; i<n; i++){
        hash[i] = i;
        for(let prev=0; prev<i; prev++){
            if(arr[prev] < arr[i] && 1 + dp[prev] > dp[i]) {
                dp[i] = 1 + dp[prev];
                hash[i] = prev;
            }
        }
        if(dp[i] > max) {
            max = dp[i];
            lastIndex = i;
        }
    }

    let lis = Array(max);
    lis[0] = arr[lastIndex];
    let ind = 1;

    while(hash[lastIndex] != lastIndex) {
        lastIndex = hash[lastIndex];
        lis[ind++] = arr[lastIndex]
    }

    return lis.reverse();
}

const arr = [5, 4, 11, 1, 16, 8];
longestIncreasingSubsequence(arr); //? 5,11,16
```

TC → $O(N^2) + O(N)$

SC → $O(N)$

if $N = 10^5$

TC → $(10^5)^2$

TLE

→ use binary search to optimize

---

lec44 → LIS (binary search)

→ we can not find LIS by binary search

[1, 7, 8, 4, 5, 6, -1, 9]

[1, 7, 8, 9] → len = 4

[1, 4, 5, 6, 9] → len = 5

{-1, 9} → len = 2

LIS → 5

Note → In interview its not preferable to rewrite an array, create a new instead

1, 7, 8, 4, 5, 6, -1, 9

-1, 4, 5, 6, 9

TC → $O(N * \log N)$

SC → $O(N)$

```cpp
#include<bits/stdc++.h>
int longestIncreasingSubsequence(int arr[], int n)
{
    vector<int> temp;
    temp.push_back(arr[0]);
    int len = 1;
    for(int i = 1;i < n; i++) {
        if(arr[i] > temp.back()) {
            temp.push_back(arr[i]);
            len++;
        }
        else {
            int ind
                = lower_bound(temp.begin(), temp.end(), arr[i]
            temp[ind] = arr[i];
        }
    }
    return len;
}
```

lec45 → Largest divisible subset

Largest Divisible Subset → arr[i] % arr[j] == 0 or arr[j] % arr[i] == 0

arr[] = {1, 16, 7, 8, 4}

{16, 8, 4} (16, 8) (8, 4)
(16, 4)

answer ⇒ {1, 16, 8, 4} → size 4

print the answer in any order

sorted 1, 4, 7, 8, 16

ex → 1, 4, 8

if 4 is divisible by 1
& 8 " " " 4
then 8 will be divisible by 1

ans [1, 4, 8, 16]

Problem Statement Suggest Edit

You are given an array of distinct numbers 'arr'. Your task is to return the largest subset of numbers from 'arr', such that any pair of numbers 'a' and 'b' from the subset satisfies the following: a % b == 0 or b % a == 0.

For Example:

```
You are given 'arr' = [1, 16, 7, 8, 4] as you can the set {1, 4, 8, 16}, here you can take any pair from the set and either one of the elements from the pair will divide the other.
```

Detailed explanation (Input/output format, Notes, Images)

Constraints:

```
1 <= T <= 10
1 <= N <= 10^3
0 <= arr[i] <= 10^8

Time Limit: 1 sec
```

```javascript
function longestIncreasingSubsequence(arr) {
    let n = arr.length;
    let dp = Array(n).fill(1);
    let hash = Array(n);
    let max = 1;
    let lastIndex = 0;
    arr.sort((a,b) => a-b);

    for(let i=0; i<n; i++){
        hash[i] = i;
        for(let prev=0; prev<i; prev++){
            if(arr[i] % arr[prev] === 0 && 1 + dp[prev] > dp[i]) {
                dp[i] = 1 + dp[prev];
                hash[i] = prev;
            }
        }
        if(dp[i] > max) {
            max = dp[i];
            lastIndex = i;
        }
    }

    let lis = Array(max);
    lis[0] = arr[lastIndex];
    let ind = 1;

    while(hash[lastIndex] != lastIndex) {
        lastIndex = hash[lastIndex];
        lis[ind++] = arr[lastIndex]
    }

    return lis;
}

const arr = [1, 16, 7, 8,4];
longestIncreasingSubsequence(arr); //? 16,8,4,1
```

TC → $O(N^2) + O(N)$
SC → $O(N)$

lec 46 → longest string chain

words[] = ['a', b, ba, bca, bda, bdca]

sequence
not subsequence

Problem Statement Suggest Edit

You are given an array 'ARR' of strings, where each string consists of English lowercase letters.
Let's say a string 'A' is a predecessor of string 'B' if and only if we can add exactly one letter **anywhere** in 'A' to make it equal to 'B'. For example "abd" is a predecessor of "abcd", we can add "c" in "abd" to make a string "abcd"
A string chain is a sequence of strings where for every 'i' in [1 . . . (K - 1)], 'S_i' is the predecessor of 'S_{i+1}'. And the length of such a string chain is 'K'.
Now your task is to find the length of the longest possible string chain.

For Example :

```
Let 'ARR' = ["x", "xx", "y", "xyx"]
The longest possible string chain is "x" -> "xx" -> "xyx"
The length of the given chain is 3, hence the answer is 3.
```

Detailed explanation (Input/output format, Notes, Images)

Constraints :

```
1 ≤ T ≤ 10
1 ≤ N ≤ 300
1 ≤ WORDS[i].length ≤ 16

Time limit: 1 sec
```

Sample Input 1 :

```
2
4
x xx y xyx
3
m nm mmm
```

Sample Output 1 :

```
3
2
```

Explanation For Sample Input 1 :

```
For test case 1 :
We will print 3 because:
The longest possible string chain is "x" -> "xx" -> "xyx".
The length of the given chain is 3, hence the answer is 3.

For test case 2 :
We will print 2 because:
The longest possible string chain is "m" -> "nm".
The length of the given chain is 2, hence the answer is 2.
```

```
for (i=1 to n)
{ for (j=0; j<i; j++)
  {
    if (compare(ass[i], ass[j]) &&
        dp[j]+1 > dp[i])
          dp[i] = dp[j]+1;
  }
  max = max(max, dp[i]
}
return max;
```

compare(ass[i], ass[j])

ass[i] = bdca
↑↑↑↑↑

ass[j] = bda
↑↑↑↑

if both the pointers exhaust together
it's a match
i can have one skip in it

TC → O(N² × length of string) + O(NlogN) ↑sorting

SC → O(N)

```
function checkPossible(s1, s2) {
    if(s1.length != s2.length+1) return false;
    let first = 0;
    let second = 0;

    while(first < s1.length) {
        if(second < s2.length && s1[first] == s2[second]) {
            first++;
            second++;
        } else {
            first++
        }
    }

    return first === s1.length && second === s2.length ? true : false;
}

function longestIncreasingSubsequence(arr) {
    let n = arr.length;
    let dp = Array(n).fill(1);
    let max = 1;
    arr.sort((a,b) => a.length - b.length);

    for(let i=0; i<n; i++){
        for(let prev=0; prev<i; prev++){
            if(checkPossible(arr[i], arr[prev]) && 1 + dp[prev] > dp[i]) {
                dp[i] = 1 + dp[prev];
            }
        }
        if(dp[i] > max) {
            max = dp[i];
        }
    }

    return max;
}

const arr = ['x', 'xx', 'y', 'xyx'];
longestIncreasingSubsequence(arr); //3
```

lec47 → longest bitonic subsequence

bitonic → increase then decrease
or, decrease then increase
or, just increasing
or, just decreasing

1 2 1 2 1
1 2 2 3 3
3 3 2 2 1

len = 4

ex 1, 3, 7, 12, 9, 4, 2
9, 6, 2, 1, 4, 7, 10

arr[] = [1, 11, 2, 10, 4, 5, 2, 1]
dp1[] = 1 2 2 3 3 4 2 1 → LIS from left
dp2[] = 1 5 2 4 3 3 2 1 → LIS from right

bitonic LIS length = 1, 6, 3, 6, 5, 6, 3, 1

4+3 = 7-1=6 (bcoz, 10 was added twice)

```
function longestIncreasingSubsequence(arr) {
    let n = arr.length;
    let dp1 = Array(n).fill(1);
    let dp2 = Array(n).fill(1);

    for(let i=0; i<n; i++){
        for(let prev=0; prev<i; prev++){
            if(arr[i] > arr[prev] && 1 + dp1[prev] > dp1[i]) {
                dp1[i] = 1 + dp1[prev];
            }
        }
    }

    for (let i = n-1; i >= 0; i--) {
        for (let prev = n-1; prev > i; prev--) {
            if (arr[i] > arr[prev] && 1 + dp2[prev] > dp2[i]) {
                dp2[i] = 1 + dp2[prev];
            }
        }
    }

    let max = 0;
    for(let i=0; i<n; i++){
        max = Math.max(max, dp1[i] + dp2[i] -1);
    }

    return max;
}

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

≥ to consider equal value in LIS

TC → $O(N^2) + O(N^2) + O(N)$
SC → $O(N)$

Lec48 → number of longest increasing subsequences

Number of longest increasing subsequence → DP 47

arr[] → {1, 3, 5, 4, 7} → lis = 4

① {1, 3, 4, 7}
② {1, 3, 5, 7}

→ ② subsequences

	1	3	5	4	7
dp[]	1	2	3	3	4
count[]	1	1	1	1	1

→ LIS

1 + 1 = 2 → ans

1,3 | 1,5 | 1,3,5 | 1,4 | 1,3,4 | 1,7 | 1,3,7 | 1,3,5,7 | 1,3,4,7

if 4

1 → 2+1 = 3

```
function longestIncreasingSubsequence(arr) {
    let n = arr.length;
    let dp = Array(n).fill(1);
    let count = Array(n).fill(1);
    let max = 1;

    for(let i=0; i<n; i++){
        for(let prev=0; prev<i; prev++){
            if(arr[i] > arr[prev] && 1 + dp[prev] > dp[i]) {
                dp[i] = 1 + dp[prev];
                // inherit
                count[i] = count[prev]
            } else if(arr[i] > arr[prev] && 1 + dp[prev] == dp[i]) {
                // increase the count
                count[i] += count[prev];
            }
        }
        max = Math.max(max, dp[i]);
    }

    let ans = 0;
    for(let i=0; i<n; i++){
        if(dp[i] === max) ans += count[i]
    }

    return ans;
}

const arr = [1, 3, 5, 4, 7];
longestIncreasingSubsequence(arr); //2
```

Problem Statement Suggest Edit

Given an integer array 'ARR' of length 'N', return the number of longest increasing subsequences in it.
The longest increasing subsequence(LIS) is the longest subsequence of the given sequence such that all elements of the subsequence are in increasing order.
Note :

```
The subsequence should be a strictly increasing sequence.
```

For Example :

```
Let 'ARR' = [50, 3, 90, 60, 80].
In this array the longest increasing subsequences are [50,
60, 80] and [3, 60, 80]. There are other increasing
subsequences as well but we need the number of longest ones.
Hence the answer is 2.
```

Detailed explanation (Input/output format, Notes, Images)

Constraints :

```
1 ≤ T ≤ 10
1 ≤ N ≤ 2000
1 ≤ ARR[i] ≤ 10 ^ 6

Time limit: 1 sec
```

Sample Input 1 :

```
2
5
50 3 90 60 80
4
3 7 4 6
```

Sample Output 1 :

```
2
1
```

Explanation For Sample Input 1 :

```
For test case 1 :
The length of LIS is 3 and there are two such LIS, which
are[50, 60, 80] and [3, 60, 80].

For test case 2 :
The length of LIS is 3 and there is only one such LIS, which
is [3, 4, 6].
```

lec 49 - matrix chain multiplication | MCM | Partition DP

Sample Output Explanation 1:

```
In the first test case, there are three matrices of
dimensions A = [4 5], B = [5 3] and C = [3 2]. The most
efficient order of multiplication is A * ( B * C).
Cost of ( B * C ) = 5 * 3 * 2 = 30  and (B * C) = [5 2] and A
* (B * C) = [ 4 5] * [5 2] = 4 * 5 * 2 = 40. So the overall
cost is equal to 30 + 40 =70.

In the second test case, there are two ways to multiply the
chain - A1*(A2*A3) or (A1*A2)*A3.

If we multiply in order- A1*(A2*A3), then the number of
multiplications required is 11250.

If we multiply in order- (A1*A2)*A3, then the number of
multiplications required is 8000.

Thus a minimum number of multiplications required is 8000.
```

Problem Statement Suggest Edit

Given a chain of matrices A1, A2, A3,.....An. Your task is to find out the minimum cost to multiply these matrices. The cost of matrix multiplication is defined as the number of scalar multiplications. A Chain of matrices A1, A2, A3,.....An is represented by a sequence of numbers in an array 'arr' where the dimension of 1st matrix is equal to arr[0] * arr[1] , 2nd matrix is arr[1] * arr[2], and so on.
For example:

```
For arr[ ] = { 10, 20, 30, 40}, matrix A1 = [10 * 20], A2 =
[20 * 30], A3 = [30 * 40]

Scalar multiplication of matrix with dimension 10 * 20 is
equal to 200.
```

Sample Input 1:

```
2
4
4 5 3 2
4
10 15 20 25
```

Sample Output 1:

```
8000
70
```

ABC

A = 10×30
B = 30×5
C = 5×60

① (AB) C → 10×30×5 + 10×5×60 ⇒ 4500

[10×5] [5×60]

min DP → 4500 (AB) C ✓

② A (BC) → 30×5×60 + 10×30×60 ⇒ 27,000

[10×30] [30×60]

given arr[] → [10, 20, 30, 40, 50] (indices 0, 1, 2, 3, 4)

N = 5

A → 10×20
B → 20×30
C → 30×40
D → 40×50

1st → A[0] * A[1]
2nd → A[1] * A[2]
3rd → A[2] * A[3]
ith → A[i-1] * A[i]

ABCD

(AB)(CD) (A)(BCD) (ABC)(D)

i j i j i j i j i j i j

i → start point
j → end point

Partition DP → Rules.

1. Start with entire block/array $f(i,j)$
2. try all partitions
   → Run a loop to try all partitions
3. Return the best possible 2 partitions

$$\begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \\ [10, & 20, & 30, & 40, & 50] \\ & A & B & C & D \\ & i & & & j \\ & \downarrow & & & \downarrow \\ & 1 & & & n-1 \end{array}$$

$f(1,4)$ → return the min multiplications to multiply matrix1 to matrix4

$\downarrow$

$f(1, n-1)$

note → we have to figure out where does i, j start based on the question

```
f(i,j) {
    if (i==j) return 0;
    min = INT_MIN;
    for (k = i to j-1)
    {
        steps = (arr[i-1] * arr[k] * arr[j]) + f(i,k) + f(k+1,j)
        min = min(min, steps)
    }
return min;
}
```

TC → exponential

SC →

memoization

dp[i][j] → IDE $O(N^3)$

i, j loop

TC → $O(N \times N \times N)$ → $O(N^3)$

SC → $O(N \times N) + O(N)$

```javascript
function f(i, j, arr) {
    if(i == j) return 0;
    let min = Number.MAX_SAFE_INTEGER;
    for(let k=i; k<j; k++){
        let steps = (arr[i-1] * arr[k] * arr[j]) + f(i, k, arr) + f(k+1, j, arr);
        min = Math.min(min, steps);
    }
    return min;
}

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

const arr = [10, 15, 20, 25];
matrixMultiplication(arr); //? 8000
```

Recursion

```javascript
function f(i, j, arr, dp) {
    if(i == j) return 0;
    if(dp[i][j] != -1) return dp[i][j];

    let min = Number.MAX_SAFE_INTEGER;
    for(let k=i; k<j; k++){
        let steps = (arr[i-1] * arr[k] * arr[j]) + f(i, k, arr, dp) + f(k+1, j, arr, dp);
        min = Math.min(min, steps);
    }
    return dp[i][j] = min;
}

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

const arr = [10, 15, 20, 25];
matrixMultiplication(arr); //? 8000
```

memoization

lec50 → continue from lecture

Tabulation

dp[N][N]

for(i=1; i<N; i++)

dp[i][i] = 0

i = n-1 to 1

j = i+1 to n-1

```javascript
function matrixMultiplication(arr) {
    let n = arr.length;
    let dp = Array(n).fill().map(() => Array(n).fill(0));

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

    for(let i=n-1; i>=1; i--) {
        for(let j=i+1; j<n; j++) {
         let min = Number.MAX_SAFE_INTEGER;
            for (let k = i; k < j; k++) {
                let steps = arr[i - 1] * arr[k] * arr[j] + dp[i][k] +dp[k+1][j];
                min = Math.min(min, steps);
            }
            dp[i][j] = min;
        }
    }

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

const arr = [10, 15, 20, 25];
matrixMultiplication(arr); //? 8000
```

tabulation

lec51 – min cost to cut the stick

given

cuts = [1, 3, 4, 5] n = 7

in cuts, we can start from anywhere

sort the array, so that we can solve sub problems independently

Problem Statement Suggest Edit

You are given chocolate of 'N' length. The chocolate is labeled from 0 to 'N'. You are also given an array 'CUTS' of size 'C', denoting the positions at which you can do a cut. The order of cuts can be changed. The cost of one cut is the length of the chocolate to be cut. Therefore, the total cost is the sum of all the cuts. Print the minimum cost to cut the chocolate.

Note:

```
All the integers in the 'CUTS' array are distinct.
```

For Example:

```
Let 'N' be: 4
Let the 'CUTS' array be: [1, 3].

Let the order of doing the cut be [1, 3].
The first cut of 1 on length 4 results in a cost of 4, and
chocolate is split into two parts of the length of 1 and 3.
The second cut of 3 on length 3 results in a cost of 3, and
chocolate is split into two parts again of the length of 2
and 1. So the total cost is 7.

The cost will remain the same if we change the order of
cutting. So the result is 7.
```

index	0	1	2	3	4	5
insert	0	1	3	4	5	7

insert n=7

i, j

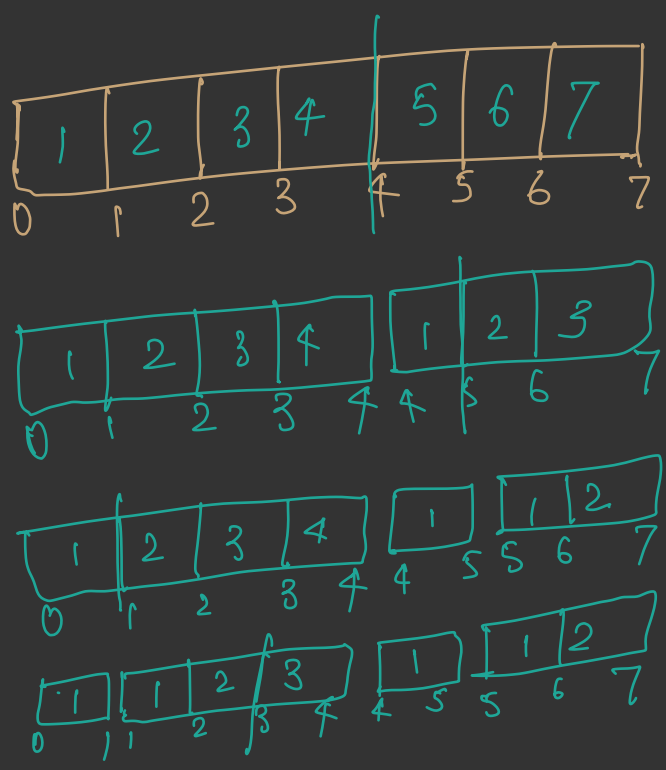
f(1, 4) ← initial call

TC → exponential

SC → O(N)

```
f(i,j){
    if(i>j) return 0;  → min = INT_MAX
    for (ind = i to j)
    { cost = cuts[j+1] - cuts[i-1] + f(i, ind-1) + f(ind+1, j)
      min = min(min, cost)
    }
    return min;
}
```

~~1,3,4,5~~



7+3+4+3 = 17

```javascript
function f(i, j, cuts) {
    if(i > j) return 0;

    let min = Number.MAX_SAFE_INTEGER;
    for(let ind=i; ind<=j; ~~i++~~) {  ind++
        let cost = cuts[j+1] - cuts[i-1] + f(i, ind-1, cuts) + f(ind+1, j, cuts);
        min = Math.min(min, cost);
    }
    return min;
}

function cost(n, cuts) {
    let c = cuts.length;
    cuts.push(n);
    cuts.unshift(0);
    cuts.sort((a,b) => a-b);
    return f(1, c, cuts);
}

const cuts = [1, 3, 4];
cost(5, cuts); //? 10
```

Recursion

```javascript
function f(i, j, cuts, dp) {
    if(i > j) return 0;
    if(dp[i][j] != -1) return dp[i][j];

    let min = Number.MAX_SAFE_INTEGER;
    for(let ind=i; ind<=j; ~~i++~~) {  ind++
        let cost = cuts[j+1] - cuts[i-1] + f(i, ind-1, cuts, dp) + f(ind+1, j, cuts, dp);
        min = Math.min(min, cost);
    }
    return dp[i][j] = min;
}

function cost(n, cuts) {
    let c = cuts.length;
    cuts.push(n);
    cuts.unshift(0);
    cuts.sort((a,b) => a-b);
    let dp = Array(c+1).fill().map(() => Array(c+1).fill(-1));
    return f(1, c, cuts, dp);
}

const cuts = [1, 3, 4];
cost(5, cuts); //?
```

memoization

Memoization

i j

1 to cuts 1 to cuts

dp[c+1][c+1]

c → size of cuts

TC → O(c × c × c) ≈ O(c³) (i, j, loop)

SC → O(c × c) + O(c)

Tabulation

```javascript
function cost(n, cuts) {
    let c = cuts.length;
    cuts.push(n);
    cuts.unshift(0);
    cuts.sort((a,b) => a-b);
    let dp = Array(c+2).fill().map(() => Array(c+2).fill(0));

    for(let i=c; i>=1; i--){
        for(let j=1; j<=c; j++){
            if (i > j) continue;
            let min = Number.MAX_SAFE_INTEGER;
            for (let ind = i; ind <= j; ind++) {
                let cost =cuts[j + 1] - cuts[i - 1] + dp[i][ind-1] + dp[ind+1][j];
                min = Math.min(min, cost);
            }
            dp[i][j] = min;
        }
    }

    return dp[1][c];
}

const cuts = [1, 3, 4];
cost(5, cuts); //? 10
```

tabulation

lec52 → burst balloons | partition DP

return max coins?

[~~3~~, 1, 5, 8] 1×3×1 = 3

[~~1~~, 5, 8] 1×1×5 = 5

[~~5~~, 8] 1×5×8 = 40

[~~8~~] 1×8×1 = 8

total coins = 56

[3, ~~1~~, 5, 8] 3×1×5 = 15

[3, ~~5~~, 8] 3×5×8 = 120

[~~3~~, 8] 1×3×8 = 24

[~~8~~] 1×8×1 = 8

167

Problem Statement Suggest Edit

There are 'N' diamonds in a mine. The size of each diamond is given in the form of integer array 'A'. If the miner mines a diamond, then he gets 'size of previous unmined diamond * size of currently mined diamond * size of next unmined diamond' number of coins. If there isn't any next or previous unmined diamond then their size is replaced by 1 while calculating the number of coins.
Vladimir, a dumb miner was assigned the task to mine all diamonds. Since he is dumb he asks for your help to determine the maximum number of coins that he can earn by mining the diamonds in an optimal order.

For Example:

```
Suppose 'N' = 3, and 'A' = [7, 1, 8]

The optimal order for mining diamonds will be [2, 1, 3].
State of mine -    [7, 1, 8]    [7, 8]    [8]
Coins earned -    (7*1*8) + (1*7*8) + (1*8*1)  = 56 + 56 + 8
= 120
Hence output will be 120.
```

Sample Input 1 :

```
2
3
7 1 8
2
9 1
```

Sample Output 1 :

```
120
18
```

Explanation For Sample Input 1 :

```
For First Case - Same as explained in above example.

For the second case -

'N' = 2, and 'A' = [9, 1]

The optimal order for mining diamonds will be [2, 1].
State of mine -    [9, 1]    [9]
Coins earned -    (1*9*1) + (1*9*1) = 9 + 9 = 18
Hence output will be 18..
```

→ go in reverse, start from last element

1 3 1 5 8 1

8 → 1×8×1 = 8

3 → 1×3×8 = 24

5 → 3×5×8 = 120

1 → 3×1×5 = 15

[b1 b2 b3 b4 b5 b6]

i ind j

$a[i-1] \times a[ind] \times a[j+1] + f(i, ind-1) + f(ind+1, j)$

f(i, j)

{ if(i>j) return 0;

→ max = INT min

for(ind = i to j) {

cost = a[i-1] x a[ind] x a[j+1]

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

max = max (max, cost)

}

return max;

}

TC → exponential

SC → O(N)

Memoization

i j

1 to N 1 to N

dp[N+1][N+1]

i j loop

TC → O(N x N x N) ≈ O(N³)

SC → O(N x N) + O(N)

AGS

```
function f(i, j, coins) {
    if(i > j) return 0;

    let max = Number.MIN_SAFE_INTEGER;
    for(let ind=i; ind<=j; ind++) {
        let cost = coins[i-1] * coins[ind] * coins[j+1] +
            f(i, ind-1, coins) +
            f(ind+1, j, coins);
        max = Math.max(max, cost);
    }
    return max;
}

function maxCoins(coins) {
    let n = coins.length;
    coins.push(1);
    coins.unshift(1);
    return f(1, n, coins);
}

const coins = [7, 1, 8];
maxCoins(coins); //? 120
```

Recursion

```
function f(i, j, coins, dp) {
    if(i > j) return 0;
    if(dp[i][j] != -1) return dp[i][j];

    let max = Number.MIN_SAFE_INTEGER;
    for(let ind=i; ind<=j; ind++) {
        let cost = coins[i-1] * coins[ind] * coins[j+1] +
            f(i, ind-1, coins, dp) +
            f(ind+1, j, coins, dp);
        max = Math.max(max, cost);
    }
    return dp[i][j] = max;
}

function maxCoins(coins) {
    let n = coins.length;
    coins.push(1);
    coins.unshift(1);
    let dp = Array(n+1).fill().map(() => Array(n+1).fill(-1));
    return f(1, n, coins, dp);
}

const coins = [7, 1, 8];
maxCoins(coins); //? 120
```

memoization

Tabulation

dp[n+1][n+1]

base case

for(i = 0 to n)

for(j = 0 to n)

if(i>j) dp[i][j] = 0

changing variable (write in opposite)

i → n to 1

j → 1 to n

copy recurrence

```
function maxCoins(coins) {
    let n = coins.length;
    coins.push(1);
    coins.unshift(1);
    let dp = Array(n+2).fill().map(() => Array(n+2).fill(0));

    for(let i=n; i>=1; i--){
        for(let j=1; j<=n; j++){
            if(i>j) continue;
            let max = Number.MIN_SAFE_INTEGER;
            for (let ind = i; ind <= j; ind++) {
                let cost =
                    coins[i - 1] * coins[ind] * coins[j + 1] +
                    dp[i][ind-1] +
                    dp[ind+1][j];
                max = Math.max(max, cost);
            }
            dp[i][j] = max;
        }
    }

    return dp[1][n];
}

const coins = [7, 1, 8];
maxCoins(coins); //? 120
```

Tabulation

lec53 → evaluate boolean expression to true

expression = 'T|T&F'

| → |
& → and
XOR → ^

find number of ways we can evaluate the expression to true?

Problem Statement Suggest Edit

You are given an expression 'EXP' in the form of a string where operands will be : (TRUE and FALSE) and operators will be : (AND, OR, XOR). Now you have to find the number of ways we can parenthesize the expression such that it will evaluate to TRUE.

Note :

```
'T' will represent the operand TRUE.
'F' will represent the operand FALSE.
'|' will represent the operator OR.
'&' will represent the operator AND.
'^' will represent the operator XOR.
```

For example :

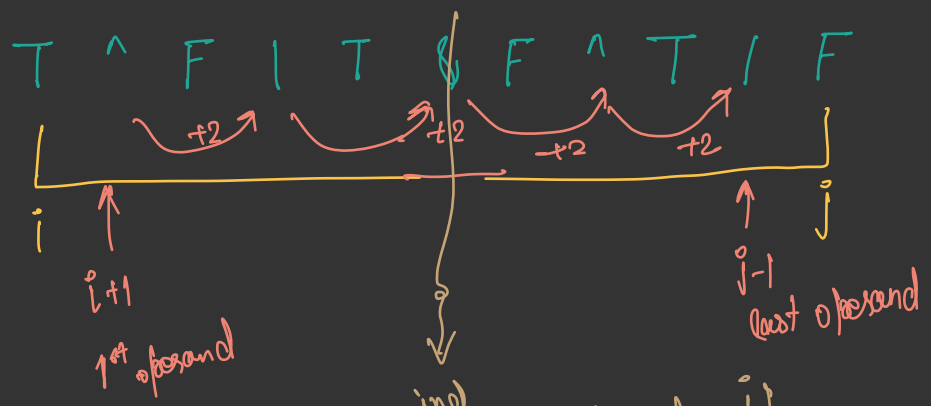
```
Input:
'EXP' = T|T & F
There are total 2  ways to parenthesize this expression:
(i) (T | T) & (F) = F
(ii) (T) | (T & F) = T
Out of 2 ways, one will result in True, so we will return 1.

Output :
1
```

intuition

→ at every operand we can do a partition & break it into independent sub-problems

→ operand are at difference of 2



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

left & right
x y

total ways = x × y

x → num of ways where x = true

y → num of ways where y = true

left | right → when it will be true

left: T = x1, F = x4
right: T = x2, F = x3

⇒ x1 × x2
⇒ x1 × x3
⇒ x2 × x4

total ways =

T^F = F
F^F = F
T^F = T
F^T = T

left ^ right

left: T = x1, F = x3
right: T = x2, F = x4

x1 × x4
x3 × x2

$f(0, n-1, 1)$ → num of ways to make true

```
f(i, j, isTrue) {
    if (i>j) return 0;
    if (i==j)
     { if (isTrue ==1) return a[i] == 'T';
       else return a[i] == 'F';
     }
ways=0
    for (ind = i+1; ind <= j-1 ; ind = ind+2)
     { LT = f(i, ind-1, 1)
       LF = f(i, ind-1, 0)
       RT = f(ind+1, j, 1)
       RF = f(ind+1, j, 0)
      if (a[ind] == '&')
   if (isTrue) ways += LT * RT;
       else ways += (LT x RF) + (LF x RT) + (RF x TF)

      else if (a[ind] == '|')

        ways
      else

 return ways
}
```

TC → exponential

SC → $O(N)$

<u>memoization</u>

i j 1/0

dp[N][N][2]

TC → $(N \times N \times 2) \times N \approx N^3$

SC → $(N \times N \times 2) + O(N)$

```javascript
function f(i, j, isTrue, exp) {
    if(i > j) return 0;
    if(i == j) return isTrue ? exp[i] == 'T' : exp[i] == 'F';

    let ways = 0;
    for(let ind=i+1; ind<=j-1; ind=ind+2) {
        let lt = f(i, ind-1, 1, exp);
        let lf = f(i, ind-1, 0, exp);
        let rt = f(ind+1, j, 1, exp);
        let rf = f(ind+1, j, 0, exp);

        if(exp[ind] == '&') {
            if(isTrue) {
                ways += lt * rt;
            } else {
                ways += lt * rf + lf * rt + lf * rf;
            }
        } else if(exp[ind] == '|') {
            if(isTrue) {
                ways += lt * rt + lt * rf + lf * rt;
            } else {
                ways += lf * rf;
            }
        } else {
            if(isTrue) {
                ways += lt * rf + lf * rt;
            } else {
                ways += lt * rt + lf * rf;
            }
        }
    }
    return ways;
}

function evaluateExp(exp) {
    let n = exp.length;
    return f(0, n-1, 1, exp);
}

const exp = "F|T^F";
evaluateExp(exp); /// 2
```

Recursion

```javascript
function f(i, j, isTrue, exp, dp) {
    if(i > j) return 0;
    if(i == j) return isTrue ? exp[i] == 'T' : exp[i] == 'F';

    if(dp[i][j][isTrue] != -1) return dp[i][j][isTrue];

    let ways = 0;
    for(let ind=i+1; ind<=j-1; ind=ind+2) {
        let lt = f(i, ind-1, 1, exp, dp);
        let lf = f(i, ind-1, 0, exp, dp);
        let rt = f(ind+1, j, 1, exp, dp);
        let rf = f(ind+1, j, 0, exp, dp);

        if(exp[ind] == '&') {
            if(isTrue) {
                ways += lt * rt;
            } else {
                ways += lt * rf + lf * rt + lf * rf;
            }
        } else if(exp[ind] == '|') {
            if(isTrue) {
                ways += lt * rt + lt * rf + lf * rt;
            } else {
                ways += lf * rf;
            }
        } else {
            if(isTrue) {
                ways += lt * rf + lf * rt;
            } else {
                ways += lt * rt + lf * rf;
            }
        }
    }
    return dp[i][j][isTrue] = ways;
}

function evaluateExp(exp) {
    let n = exp.length;
    let dp = Array(n).fill().map(() => Array(n).fill().map(() => Array(2).fill(-1)));
    return f(0, n-1, 1, exp, dp);
}

const exp = "F|T^F";
evaluateExp(exp); /// 2
```

memoization

lec 54. Palindrome partitioning - II | Front partition

every single char is a palindrome in itself → cuts (n-1)

string = bab | abcba | d | c | e d e

ans = 4 4 patterns

bababcbadcede (front partition)

1 + (ababcbadcede) | 1 + (abcbadcede)

1 + (cbadcede)

Problem Statement Suggest Edit

Given a string 'str'. Find the minimum number of partitions to make in the string such that every partition of the string is a palindrome.

Given a string, make cuts in that string to make partitions containing substrings with size at least 1, and also each partition is a palindrome. For example, consider "AACCB" we can make a valid partition like A | A | CC | B. Among all such valid partitions, return the minimum number of cuts to be made such that the resulting substrings in the partitions are palindromes.

The minimum number of cuts for the above example will be AA | CC | B. i.e 2 cuts

Note :

1) We can partition the string after the first index and before the last index.

2) Each substring after partition must be a palindrome.

3) For a string of length 'n', if the string is a palindrome, then a minimum 0 cuts are needed.

4) If the string contains all different characters, then 'n-1' cuts are needed.

5) The string consists of upper case English alphabets only with no spaces.

```
f(i)
{ if (i==n) return 0;
    temp = " ";
    min = INT_MAX;
  for (j=i; j<n; j++)
      temp += s[j]
      if (isPalindrome(temp))
          cost = 1 + f(j+1);
          min = min(min, cost)
return min;
```

i, loop → palindromecheck

TC → O[N × N] + O(N) exponential

SC → O(N)

memoization

i

1 to N

dp[N]

state loop

TC → O[N × N]

SC → O(N) + O(N)

```javascript
function isPalindrom(str) {
    return str.split('').reverse().join('') === str;
}

function f(i,n,s) {
    if(i > n) return 0;

    let temp = '';
    let min = Number.MAX_SAFE_INTEGER;
    for(let ind=i; ind<n; ind++){
        temp += s[ind];
        if(isPalindrom(temp)){
            let cost = 1 + f(ind, n, s);
            min = Math.min(min, cost);
        }
    }

    return min;
}

function palindromePartitioning(s) {
    let n = s.length;
    return f(0, n, s) - 1;
}

const s = "ABBAC";
palindromePartitioning(s); //? 1
```

alpk! remove this

Recursion

both are exceeding max call stack

```javascript
function isPalindrom(str) {
    return str.split('').reverse().join('') === str;
}

function f(i,n,s, dp) {
    if(i > n) return 0;
    if(dp[i] != -1) return dp[i];

    let temp = '';
    let min = Number.MAX_SAFE_INTEGER;
    for(let ind=i; ind<n; ind++){
        temp += s[ind];
        if(isPalindrom(temp)){
            let cost = 1 + f(ind, n, s, dp);
            min = Math.min(min, cost);
        }
    }

    return dp[i] = min;
}

function palindromePartitioning(s) {
    let n = s.length;
    let dp = Array(n).fill(-1);
    return f(0, n, s, dp) - 1;
}

const s = "ABBAC";
palindromePartitioning(s); //? 1
```

memoization

Tabulation

1. dp[n]
2. base case dp[n] = 0
3. i = n-1 to 0
4. copy recurrence

```javascript
function isPalindrom(str) {
    return str.split('').reverse().join('') === str;
}

function palindromePartitioning(s) {
    let n = s.length;
    let dp = Array(n).fill(0);
    dp[n] = 0;

    for(let i=n-1; i>= 0; i--){
        let temp = "";
        let min = Number.MAX_SAFE_INTEGER;
        for (let ind = i; ind < n; ind++) {
            temp += s[ind];
            if (isPalindrom(temp)) {
                let cost = 1 + dp[ind+1];
                min = Math.min(min, cost);
            }
        }

        dp[i] = min;
    }

    return dp[0] - 1;
}

const s = "ABBAC";
palindromePartitioning(s); //? 1
```

tabulation

lec 55 → Partition array for max sum / Isort partition

arr[] = [1, 15, 7, 9, 2, 5, 10] k = 3

[1, 15 | 7 9 2 | 5 10] [1 15 7 | 9 | 2 5 10]

[15 15 15 | 9 | 10 10 10]

[15, 15 | 9 9 9 | 10 10] Sm = 84

$\boxed{Sm = 77}$

max 3 elements in a partition

Rules

1. express everything in ind
2. try every partition possible from that index
3. take the best partition

[1 15 | 15 15 7 | 9 2 5 10]

i j K

len=1 len2 len3

```
f(ind)
{ if(ind == n) return 0;
  maxAns = INTMIN;
  len=0; max = INTMIN;
  for (j = ind; j < min(n, ind+k); j++) {
      len++;
      max = max(max, arr[j])
      sum = len x max + f(j+1)
      maxAns = max(sum, maxAns)
  }
  return maxAns;
}
```

TC → exponential

SC →

Memoization

i → 0 to n-1

dp[n]

TC → O(N) × O(K)

SC → O(N) + O(N)
ASS

Tabulation

dp[n]

Base case dp[n] = 0

ind = n-1 to 0

copy recurrence

```javascript
function f(i, arr, k, n) {
    if(i == n) return 0;

    let max = Number.MIN_SAFE_INTEGER;
    let maxAns = Number.MIN_SAFE_INTEGER;
    let len = 0;

    for(let ind=i; ind<Math.min(ind+k, n); ind++){
        len++;
        max = Math.max(max, arr[ind]);
        let sum = len * max + f(ind+1, arr, k, n);
        maxAns = Math.max(maxAns, sum);
    }

    return maxAns;
}

function maximumSubarray(arr, k) {
    let n = arr.length;
    return f(0, arr, k, n);
}

const arr = [1, 20, 13, 4, 4, 1];
maximumSubarray(arr, 3); //? 120
```

Recursion

```javascript
function f(i, arr, k, n, dp) {
    if(i == n) return 0;
    if(dp[i] != -1) return dp[i];

    let max = Number.MIN_SAFE_INTEGER;
    let maxAns = Number.MIN_SAFE_INTEGER;
    let len = 0;

    for(let ind=i; ind<Math.min(ind+k, n); ind++){
        len++;
        max = Math.max(max, arr[ind]);
        let sum = len * max + f(ind+1, arr, k, n, dp);
        maxAns = Math.max(maxAns, sum);
    }

    return dp[i] = maxAns;
}

function maximumSubarray(arr, k) {
    let n = arr.length;
    let dp = Array(n).fill(-1);
    return f(0, arr, k, n, dp);
}

const arr = [1, 20, 13, 4, 4, 1];
maximumSubarray(arr, 3); //? 120
```

memoization

```javascript
function maximumSubarray(arr, k) {
    let n = arr.length;
    let dp = Array(n).fill(0);
    dp[n] = 0;

    for(let i=n-1; i>=0; i--){
        let max = Number.MIN_SAFE_INTEGER;
        let maxAns = Number.MIN_SAFE_INTEGER;
        let len = 0;

        for (let ind = i; ind < Math.min(ind + k, n); ind++) {
            len++;
            max = Math.max(max, arr[ind]);
            let sum = len * max + dp[ind+1];
            maxAns = Math.max(maxAns, sum);
        }

        dp[i] = maxAns;
    }

    return dp[0];
}

const arr = [1, 20, 13, 4, 4, 1];
maximumSubarray(arr, 3); //? 120
```

Tabulation

lec 56 → max Rectangle area with all 1's | DP on rectangles.

2x3 = 6

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

4x1

1x3

Histogram

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

QBX → max area of Histogram

4x1 → max sum

1

2

6

4

ans = 6

C++ Code Java Code

```cpp
#include <bits/stdc++.h>

using namespace std;
class Solution {
  public:
    int largestRectangleArea(vector < int > & histo) {
      stack < int > st;
      int maxA = 0;
      int n = histo.size();
      for (int i = 0; i <= n; i++) {
        while (!st.empty() && (i == n || histo[st.top()] >= histo[i])) {
          int height = histo[st.top()];
          st.pop();
          int width;
          if (st.empty())
            width = i;
          else
            width = i - st.top() - 1;
          maxA = max(maxA, width * height);
        }
        st.push(i);
      }
      return maxA;
    }
};
int main() {
  vector < int > histo = {2, 1, 5, 6, 2, 3, 1};
  Solution obj;
  cout << "The largest area in the histogram is " << obj.largestRectangleArea(histo
  return 0;
}
```

Output: The largest area in the histogram is 10

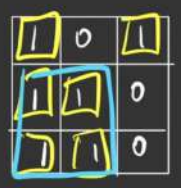
Time Complexity: O(N) + O (N)

```cpp
int maximalAreaOfSubMatrixOfAll1(vector<vector<int>> &mat,
                                 int n, int m){
    int maxArea = 0;
    vector<int> height(m, 0);
    for(int i = 0; i< n; i++) {
        for(int j = 0;j<m;j++) {
            if(mat[i][j] == 1) height[j]++;
            else height[j] = 0;
        }
        int area = largestRectangleArea(height);
        maxArea = max(area, maxArea);
    }
    return maxArea;
    // Write your code here.
}
```

TC → O(N*(M+N))

SC → O(N)

lec 57 → count square sub matrices with all ones | DP on rectangles.

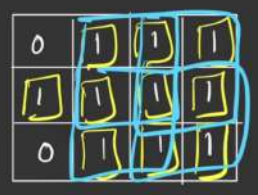


M×M

6 size 1 squares

1 size 2 squares

in Total → 7 squares



10 size 1 squares

4 size 2 squares

1 size 3 squares

15 total squares

1	1	1
1	1	1
1	1	1

right bottom is at (i,j)

	0	1	2
0	1	1	1
1	1	2	2
2	1	2	3

dp[i][j]

how many squares end at (i,j)

1	1	1	1
1	1	1	1
1	1	1	1

→

1	1	1	1
1	2	2	2
1	2	3	3

minimal of all three + himself

1+1 = 2

$$dp[i][j] = \min(dp[i-1][j], dp[i-1][j-1], dp[i][j-1]) + 1$$

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

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

    for(let i=1; i<n; i++){
        for(let j=1; j<m; j++){
            if(matrix[i][j] == 0) {
                dp[i][j] = 0;
            } else {
                dp[i][j] = 1 + Math.min(dp[i-1][j], dp[i-1][j-1], dp[i][j-1]);
            }
        }
    }

    let sum = 0;
    for(let i=0; i<n; i++){
        for(let j=0; j<m; j++) {
            sum += dp[i][j];
        }
    }

    return sum;
}
const matrix = [[0, 1, 1, 0],
[1, 1, 1, 0],
[0, 0, 1, 0]]
countSquares(matrix); //? 7
```

TC -> O(N×M)
SC -> O(N×M)