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

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 x 2)

SC -> O(N x 2) + O(N+2)

RSS

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

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.

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;

}

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

TC → exponential

SC → O(N)
AS

memoization

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

dp[N][2][3]

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

```
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

```
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); //? 6
```

lec 40 buy & sell stocks with cooldown

cooldown → can't buy right after sell

unlimited transactions

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 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

```
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]

fee = 2

(from loc 37)

7 5

-2 -2

5 + 3 = 8

```
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 function)

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 → 0 to n-1 (ind), -1 to n-1 (prev)

dp[N][N+1]

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

TC → O(N×N)

SC → O(N×N) + O(N)

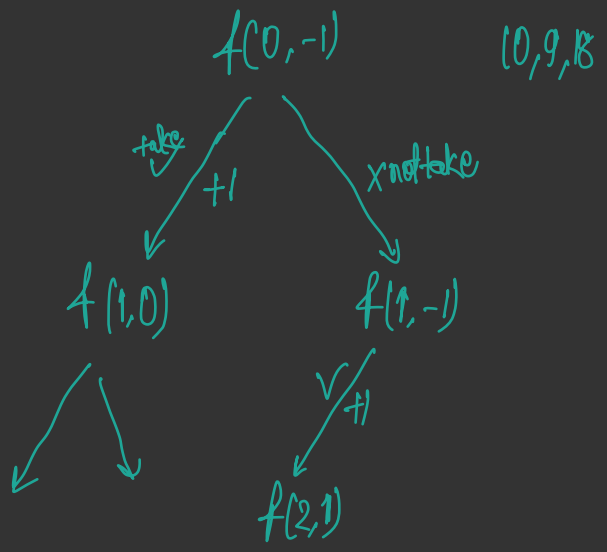
Problem Statement Suggest Edit

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.
```




```
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

```
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
```

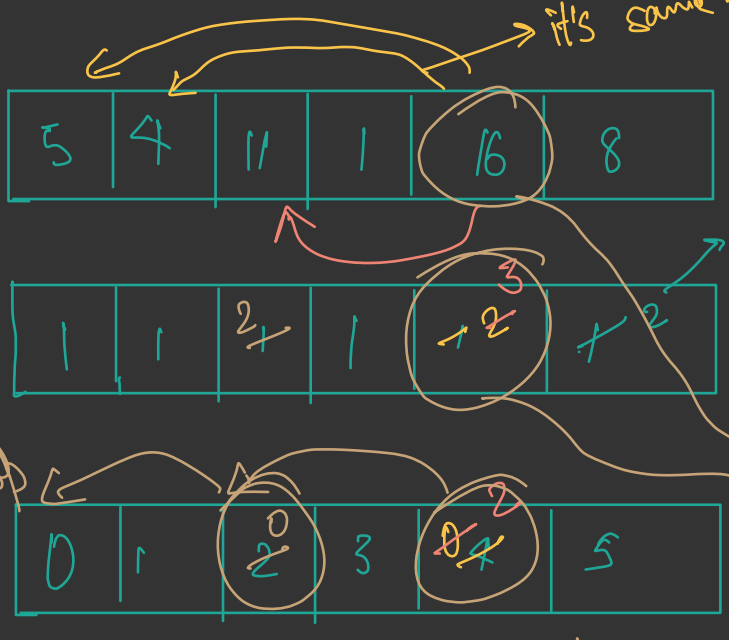
tabulation

```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(N²)
SC → O(N)



```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 print 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[j] = bdca
ass[i] = 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
```

loc47 -> 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)

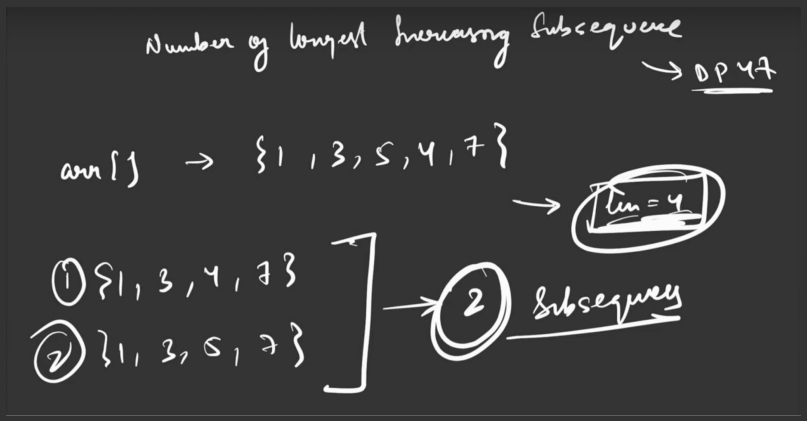
```
function longestIncreasingSubsequence(arr) {
    let n = arr.length;
    let dp1 = Array(n).fill(1);
    let dp2 = Array(n).fill(1);

    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
```

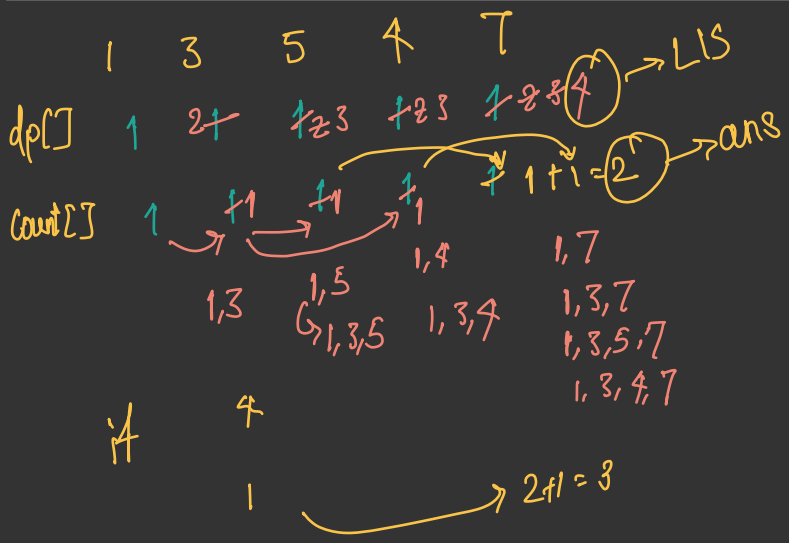
2 -> 1

>= -> to consider equal value in LIS

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

ler48 → number of longest increasing subsequences






```javascript
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] (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 → 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

```
   0    1    2    3    4
 [10,  20,  30,  40,  50]
       A    B    C    D

       i              j
       ↓              ↓
       1             n-1
```

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

i, j
dp [N][N] → 1DE $O(N^3)$

i j for loop
TC → O(N x N x N) → $O(N^3)$

SC → O(N x 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

loc50 -> continue from last 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

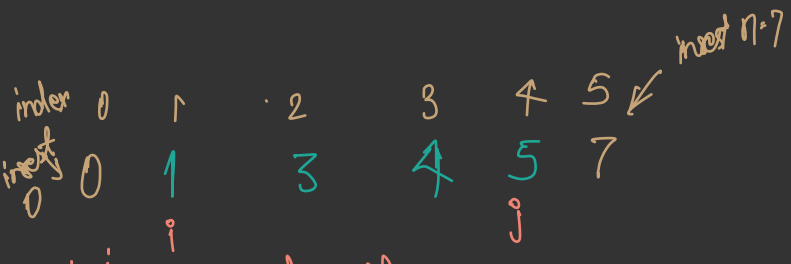
Problem Statement Suggest Edit

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.
```



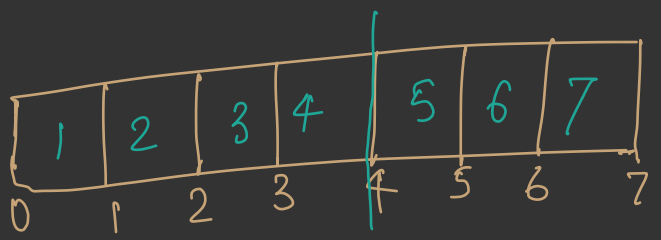
i, j

f(1,4) ← initial call

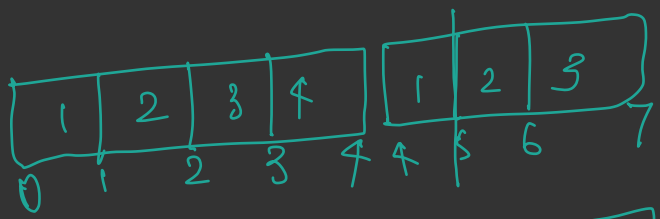
TC → exponential
SC → O(N)

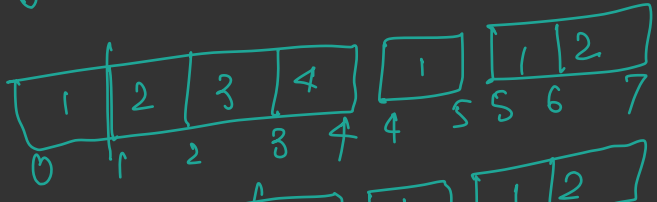
```
f(i,j){
  if(i>j) return 0;        → min = INT_MAX
  for(ind = i to j)
  { cost = cuts[j+1] - cuts[i-1] + f(i, ind-1) + f(ind+1, j)
    min = min(min, cost)
  }
  return min;
}
```

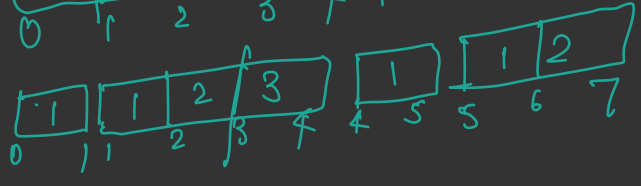
1,3,4,5



7+3+4+3 = 17







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

    let min = Number.MAX_SAFE_INTEGER;
    for(let ind=i; ind<=j; 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

```
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; 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[i+1][c+1]

c → size of cuts

TC → $O(c \times c \times c) \approx O(c^3)$ (i, j, loop)

SC → $O(c \times c) + O(c)$

Tabulation

```
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 | position DP

return max coins?

[3, 1, 5, 8] → 1x3x1 = 3

[x, 5, 8] → 1x1x5 = 5

[x, 8] → 1x5x8 = 40

[8] → 1x8x1 = 8

total coins = 56

[3, x, 5, 8] → 3x1x5 = 15

[3, x, 8] → 3x5x8 = 120

[3, 8] → 1x3x8 = 24

[8] → 1x8x1 = 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 → 1x8x1 = 8

3 → 1x3x8 = 24

5 → 3x5x8 = 120

1 → 3x1x5 = 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;

for(ind = i to j) { → max = INT min

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(N^3)

SC → O(N x N) + O(N)

ASS

```
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

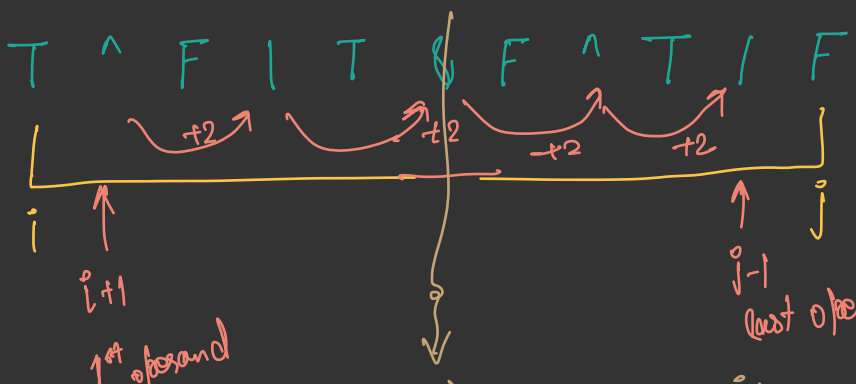
You are given an expression 'EXP' in the form of a string where operands will be : (TRUE and FALSE) and operators will be : (AND, OR, XOR). Now you have to find the number of ways we can parenthesize the expression such that it will evaluate to TRUE.
Note :

```
'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

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

left ^ right
↓ ↓
T = x1 T = x2
F = x3 F = x4

x1 × x4
x3 × x2

left | right → when it will be true
↓ ↓
T = x1 T = x2 ⇒ x1 × x2
F = x4 F = x3 ⇒ x1 × x3
⇒ x2 × x4

total ways =

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)

memoization

i j 1/0

dp[N][N][2]

TC → (N x N x 2) x N ≈ N^3

SC → (N x N x 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

loc 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

babababcbadcede (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 x N] + O(N) — exponential

SC → O(N)

memoization

i: 1 to N

dp[N]

TC → O[N x N] (state, loop)

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
```

afpf! 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

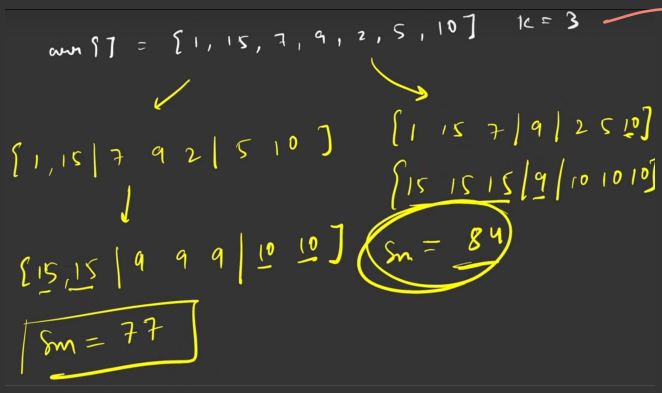
③ i = n-1 to 0

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 / Hard partition



Rules

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

[15 15 15 | 9 2 5 10]

i j K

len1 len2 len3

```
f(ind)
{
  if(ind == n) return 0;
  maxAns = INT_MIN;
  len = 0; max = INT_MIN;
  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) x 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

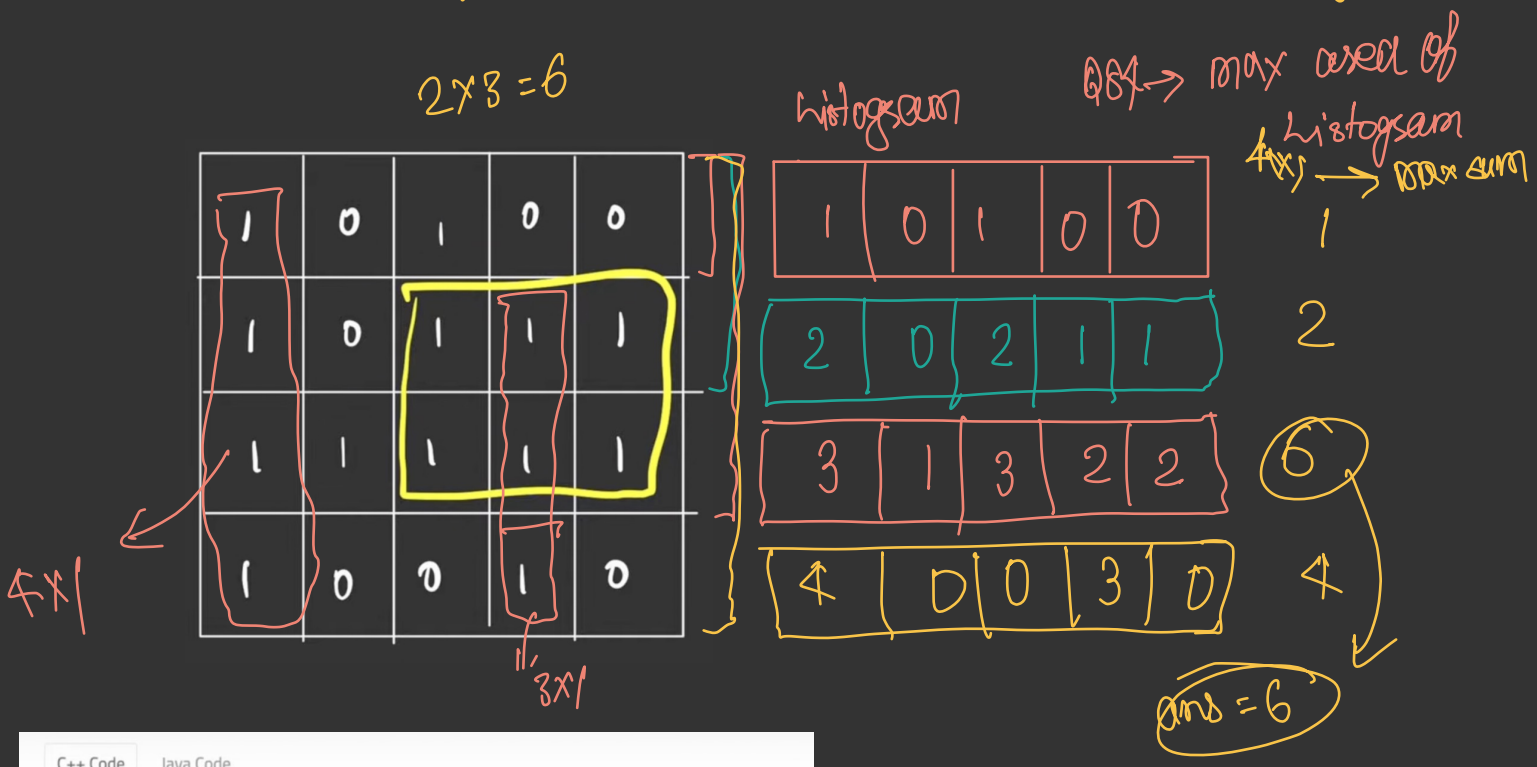
```javascript
function f(i, arr, k, n, dp) {
    if(i == n) return 0;
    if(dp[i] != -1) return dp[i];

    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.



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

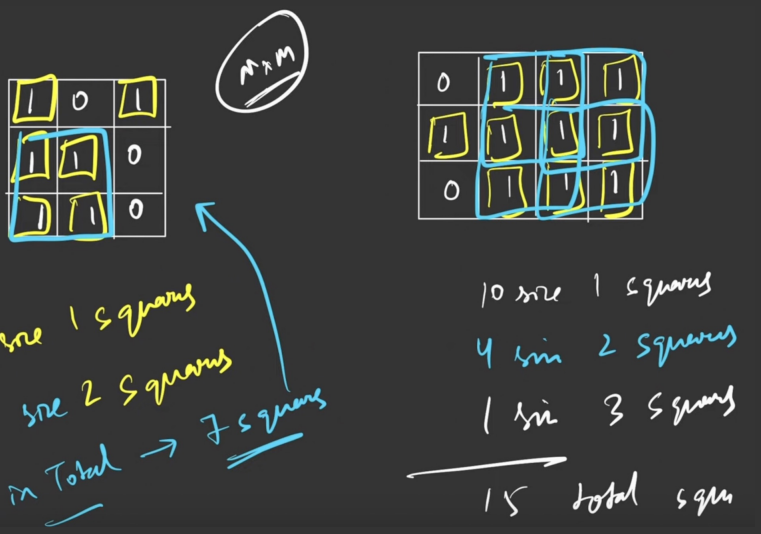
Time Complexity: O(N) + O (N)

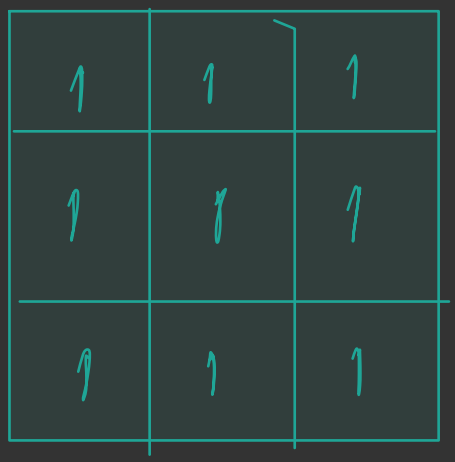
```cpp
int maximalAreaOfSubMatrixOfAll1(vector<vector<int>> &mat,
                                 int n, int m){
    int maxArea = 0;
    vector<int> height(m, 0);
    for(int i = 0; i< n; i++) {
        for(int j = 0;j<m;j++) {
            if(mat[i][j] == 1) height[j]++;
            else height[j]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

TC → O(N * (M+N))

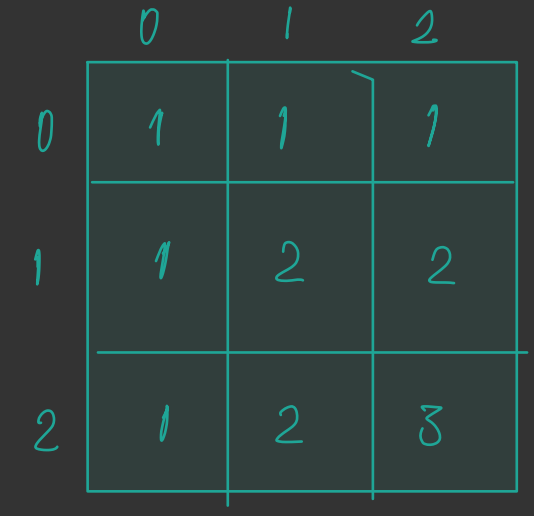
SC → O(N)

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



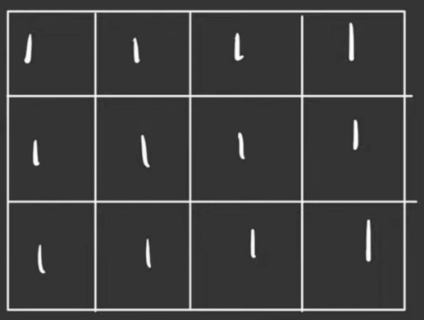


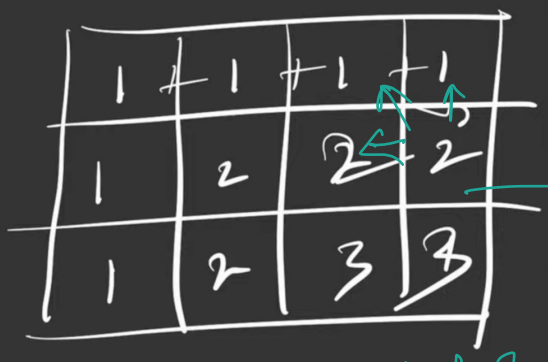
right bottom is at (i,j)



dp[i][j]

how many squares end at (i,j)





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 x M)
SC -> O(N x M)