DP

1. Memorization
2. Tabulation : 1. Storage & Meaning
   2. Direction of probblem -> Chote problem kis taraf hain or bade problem kis taraf hein
   3. Travel and solve: traverse from chote -> bade problem

Approaches

1. TSS and its variation
2. Include/Exclude problem— Any question that says NO CONSECUTIVE ELEMENTS can be solved using this method
3. No of ways Problems -> THESE ARE LIKE RECURSION PROBLEMS WHERE WE USE **EXPECTATION FAITH STRATEGY**.
4. Buy and Sell stock
5. Path printing problem -> DP followed by BFS
6. Longest Increasing Subseq variations
7. Catalan Numbers
8. Cutting Strategy
9. Subsequence Variations & variation of Longest Common Subsequence

1. Target Sum ubset Variation Problems

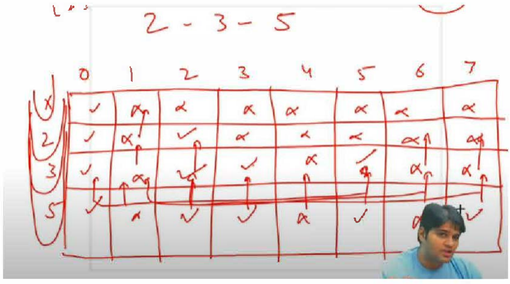
TSS
- Coin changhe Permutation
- Coin change Combination
- 01 Knapsack
- Unbounded Knapsaack

2D Arrray method: **(Target sum subset)**

Storage-> 2D array of n+1 rows and target+1 cols
Meaning-> dp[i][j] -> Is there any subset with elements fron 0th idx -> ith idx which can sum up to value j

If we are deciding if 4th player can bat or not (and desired score is x)
1. If the team of first 3 players are already able to make a score of x, we need not make the 4th player bat
2. If the team of first 3 player donot score x runs, we can further decide if the 4th player should bat or not
   -> If the first 3 player hav their scores sum up = x-score[4th player] then we can make the 4th player bat coz x-score[4th player]+score[4th player] = x

Agar ye banda battting karta hain to baki team ko bache hue run banana padta he
Agar ye banda batting nah karta to baki ki team ko saari run banane padta he



Permutation and Combination: 1D array

Coin change combination

Meaning-> dp[i] -> No of combinations of coins that can be used to give change 'i'
Direction-> 0-> Amt

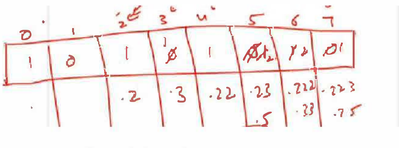
```
vector<int> dp(tarAmt + 1, 0);
dp[0] = 1; // No of ways 0rs can be payed is 1(Give nothging)
//* We are going coinwise because this is combination Eg if coins are {2,3,5,7} then when we go coinwise, 3 can come only
after 2 and hence 32 is newer possible, ieif 23 is there 32 will never be there hence avoiding permutation
for(int &coin : coins){
    for(int amt = coin; amt < dp.size(); amt++){
        dp[amt] += dp[amt - coin];
    }
}
return dp[tarAmt];
```

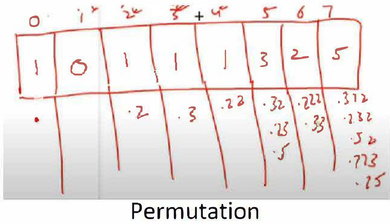
Coin change permutation:

Meaning-> dp[i] -> No of permutations of coins that can be used to give change 'i'
Direction-> 0-> Amt

```
dp[0] = 1; // No of ways 0 rs can be payed is 1(Give nothging)
for(int amt = 1; amt <= tarAmt; amt++){
    for(int &coin : coins){
        if(amt - coin >= 0){
            int remAmt = amt - coin;
            dp[amt] += dp[remAmt];
        }
    }
}
return dp[tarAmt];
```

VVIP point we need to observe is that
In combination we are iterating **coinwise** so fi coins = {2,3,5}, 3 will always be analysed after 2 ie for target =5, if 23 is an answer, 32 will never be an answer hence avoiding permutation
In permutation we are iterating amount wise, ie each target mount can be payed by any coin ie if target = 5 and we are iterating over coins for 5, we can get both 23 and 32


Combination


Permutation

Here all the 3 problems above are solving the same problem is
TARGET SUM SUBSET(in coins problem, we have to find subset of coins summing up to given target) but why are we using 1D array for coins problem and 2d array for Target sum subset?

Ans is that in coins problem, duplicacy is allowed and in TSS duplicacy is not allowed Hence

IF DUPLICACY-> 2D array
No DUPLICACY-> 1D array

To solve any variation of TSS

1. Check if duplicacy is allowed or not
   - Allowed => 1D array
   - Not allowed => 2D array
2. For 1D array further check if order matters or not
   - Matters -> Permutation
   - Doesnot matter-> Combination

2. Include Exclude Problems

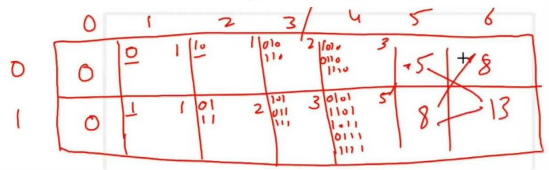
Any question that says NO CONSECUTIVE ELEMENTS can be solved using this method

1. Count binary substrings with No cosecutive 0's

```
Now we start a new type of problems -> Include and not include
Storage-> 2 1D arrays
Meaning-> arr1[i]-> Stores the no of binary strings of  length i that have no repetetive 0's and //* end with 0
          arr2[i]-> Stores the no of binary strings of  length i that have no repetetive 0's and //* end with 1
Direction->0->n
Traverse and solve-> To find arr1[i] -> we append 0 to all the strings ending with 1
                                        we get all the strings with no repetitive 0's
                                        and end with 0
                    To find arr2[i] -> we append 1 to all the strings ending with both 0
                                        and 1 we get all the strings with no repetitive
                                        0's and end with 1
```

```
int noofBinaryStrings(int n){
    int *dp0 = new int[n+1];
    int *dp1 = new int[n+1];
    dp0[0] = dp1[0] = 0;
    dp0[1] = dp1[1] = 1;
    for(int i = 2; i <= n; i++){
        dp0[i] = dp1[i-1];
        dp1[i] = dp0[i-1] + dp1[i-1];
    }
    return dp0[n] + dp1[n];
}
```

```
int noofBinaryStringsoptimal(int n){
    int ocZeros = 1, ocOnes = 1; //Old count zeros and ones
    for(int i = 2; i <= n; i++){
        int nczeros = ocOnes;
        int ncOnes = ocZeros + ocOnes;
        ocOnes = ncOnes;
        ocZeros = nczeros;
    }
    return ocOnes + ocZeros;
}
```



2. Arrange Buildings:

The road has n plots on it's each side.The road is to be so planned that there should **not be any consecutive buildings** on either side of the road.

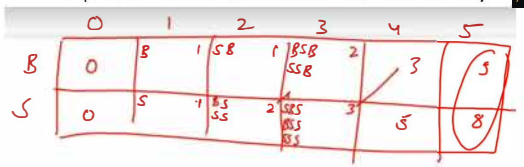
This problem is similar to Binary strings problem

if 1- Building is present

0- Plot is left empty

The question is that no two 1's should occur consecutively

```
void arrangeBuilding(int n){
    long int ob = 1;
    long int os = 1;
    for(int i = 2; i <= n; i++){
        int nb = os;
        int ns = ob + os;
        ob = nb;
        os = ns;
    }
    cout << (ob + os)*(ob + os);
}
```

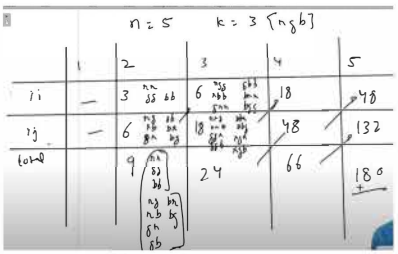


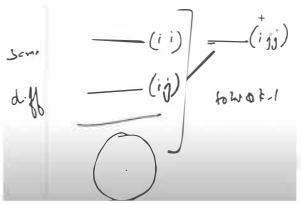
5. Paint Fence

You are given n fences and k colors, You have to print the no of ways in which n fences can be painted using k colors such that **not more than two consecutive** fences are painted with same color

```
Storage-> 2 1D arrays SAME and DIFFERENT
Meaning ->
    same[i] -> No of ways in which the first i fences can be painted with k colors with last two fenses having the same color
               ie //* i and i-1 th fence painted with SAME color
    diff[i] -> No of ways in which the first i fences can be painted with k colors with last two fenses having the diff color
               ie //* i and i-1 th fence painted with DIFFERENT color
Direction -> 1->n
Traverse and solve->
    same[2] = K*1 ie We can color the first fense with any color ie we have k options and we have to color the second fence wi
                  with the same color which was used in the prev fense ie only one option
    diff[2] = K * (K-1) ie we can color the first fence with any color ie we have K options, and we can color the second
                  fence with any other color excluding the prev color ie K-1 options
    for 3rd to nth fence
      same[i] = diff[i-1] because we can color the fence with same color as prev fence only when prev two fences aare painted
                  with different color ie diff[i-1]
    diff[i] = ( same[i-1]+diff[i-1] ) * (K-1)  -> We can choose any of the previous options and color it with a color
                  different than previous color i
```

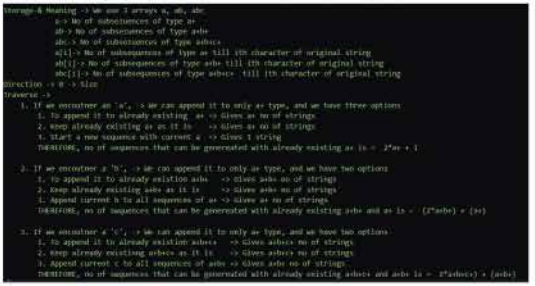
```
int paintFence(int n, int k){
    if(n == 1) return k;
    int same = k;
    int diff = k * (k-1);
    int total = same+diff;
    for(int i = 3; i <= n; i++){
        same = diff * 1;
        diff = total * (k-1);
        total = same + diff;
    }
    return total;
}
```

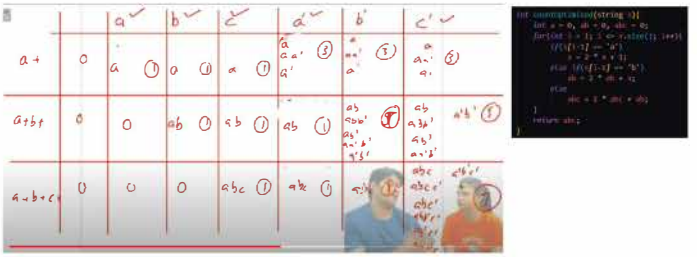




3. Count a+b+c+ sub sequences

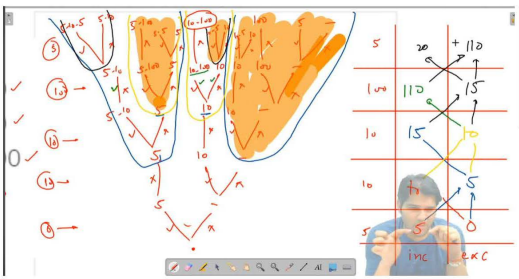
Given a string of abc, count no of a+b+c+ subsequences





4. Max sum not adjascent elements

```
Storage-> 2 1D arrays
Meaning-> inc[i]-> Stores max sum non adjascent elements ending with ith element INCLUDED
          exc[i]-> Stores max sum non adjascent elements ending with ith element NOT INCLUDED
Direction->0->n
Traverse and solve:
      Initialise inc[1] = arr[0] and exc[1] = 0
      inc[i] has only option ie inc[i] = exc[i] + arr[i] because if current element has to be included, prev element has
      to be excluded
      exc[i], since curr element cannot be included, we can choose either the sum till prev element where prev element was
      not included or prev element included WHICHEVER IS GREATEST ie max(inc[i-1], exc[i-1]) ie if curr element is excluded,
      prev element can be excluded or included
```



```
int maxSum(vector<int> arr){
    int inc = arr[0];
    int exc = 0;
    for(int i = 1; i < arr.size(); i++){
        int ninc = exc + arr[i];
        int nexc = max(inc, exc);
        inc = ninc;
        exc = nexc;
    }
    return max(inc, exc);
}
```

3. No of ways problems

These are types of pronblems in which the current **no of ways** depends on prev no of ways depending on conndition and the problem

THESE ARE LIKE RECURSION PROBLEMS WHERE WE USE **EXPECTATION FAITH STRATEGY.**We just need to see on which prev value is the current value depending

1. COUNT ENCODINGS

You are given a string str of digits. In which 1 represents a, and 2 represents and so on. You are required to calculate and print the count of encodings for the string str.

For 123 -> there are 3 encodings. abc, aw, lc

For 993 -> there is 1 encoding. iic

For 013 -> This is an invalid input. A string starting with 0 will not be passed.

For 103 -> there is 1 encoding. jc

For 303 -> there are 0 encodings. But such a string maybe passed. In this case print 0.

```
Meaning-> dp[i] No of possible encodings for first i digits
Direction -> 1->n
Traverse and solve-> For each digit, we have 2 options
          1. To treat it as a single digit and append it to all the encodings of i-1
          2. If the prevdight and curr digit is <= 26, we can append the encoding to all encodings of i-2
```

```cpp
int countEncodings(string s){
    int n = s.size();
    vector<int> dp(n+1, 0);
    dp[0] = dp[1] = 1; //* Even though wkt dp[0]=0, we are initializing it to 1 because it is a HELPER value for further indices Eg for i = 1 and str = 103, and option
    2 -> dp[idx] += dp[idx-2] ie dp[2] += dp[0] so if we initialize dp[0] = 0, dp[2] will also be 0 which is not the case , so we initialize it to 1
    for(int i = 1; i < s.size(); i++){
        int idx = i+1;
        if(s[i] != '0')
            dp[idx] += dp[idx-1];

        if(s[i-1] != '0'){
            int val = stoi(s.substr(i-1, 2));
            if(val >= 1 && val <= 26){
                dp[idx] += dp[idx-2];
            }
        }
    }
    return dp[n];
}
```

2. Tiling with 2x1 tiles

You are given a number n representing the length of a floor space which is 2m wide. It's a 2 * n board. Also you are given infinite supply of 2x1 tile. Print the number of ways floor can be tiled using 2x1 tiles.

```
h[i] -> No of ways in which tiles can be placed such that ith tile is a horizontal tile
v[i] -> No of ways in which tiles can be placed such that ith tile is a Vertical tile

Traverse and Solve->
    We can place tiles horizontally beside tiles of length i-2
    we can place tiles vertically beside ties of length i-1
```

```cpp
int countWays(int n){
    vector<int> h(n+1, 0);
    vector<int> v(n+1, 0);
    h[1] = 0;
    v[1] = 1;
    h[2] = v[2] = 1;
    for(int i = 3; i <= n; i++){
        h[i] = h[i-2] + v[i-2];
        v[i] = h[i-1] + v[i-1];
    }
    return h[n] + v[n];
}
```

```cpp
int countWays(int n){
    vector<int> dp(n+1, 0);
    dp[1] = 1;
    dp[2] = 2;
    for(int i = 3; i <= n; i++){
        int h = dp[i-2];
        int v = dp[i-1];
        dp[i] = h+v;
    }
    return dp[n];
}
```

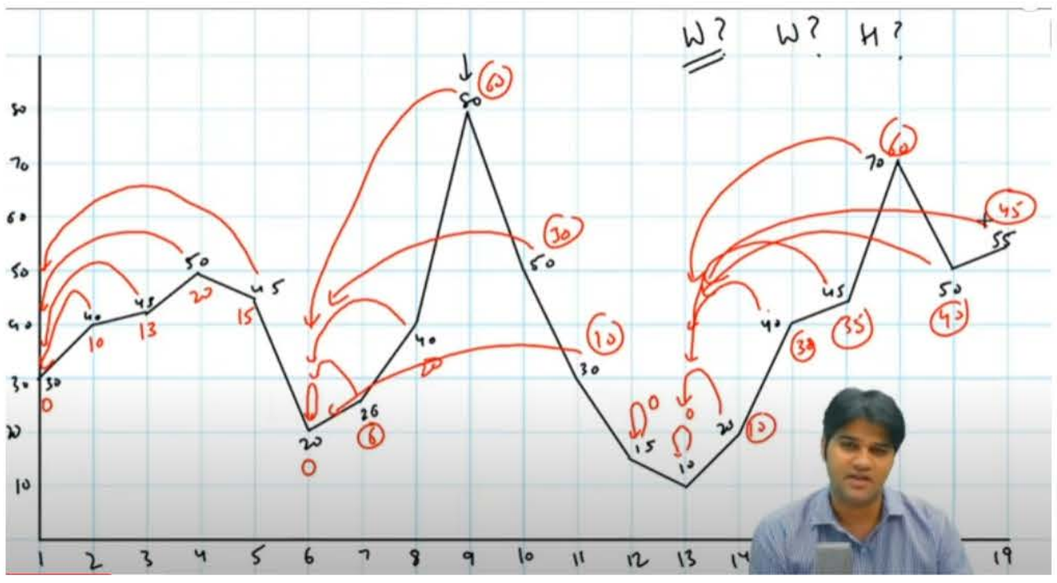
3. Tiling with Mx1 tiles

Here you are given an Mxn board and infinite Mx1 tiles

```cpp
//This is just a variation of prev problem
int countWays(int n, int m){
    vector<int> dp(n+1, 0);
    dp[0] = 1; //*This is just a helper value for horizontal placing , say n = m = 3, dp[i[ = dp[i-1] ie dp[3]
    = dp[2] for vertical and for horizontal i-m ie 3-3 = 0, so in this case, we can place a horizontal tile, if
    dp[0]=0 , the horizontal wont be considered
    dp[1] = 1;
    for(int i = 2; i <= n; i++){
        dp[i] = dp[i-1]; //Placing vertical
        if(i - m >= 0)   //To place horizontal
            dp[i] += dp[i-m];
    }
    return dp[n];
}
```

Buy and Sell Stock : 1Transaction

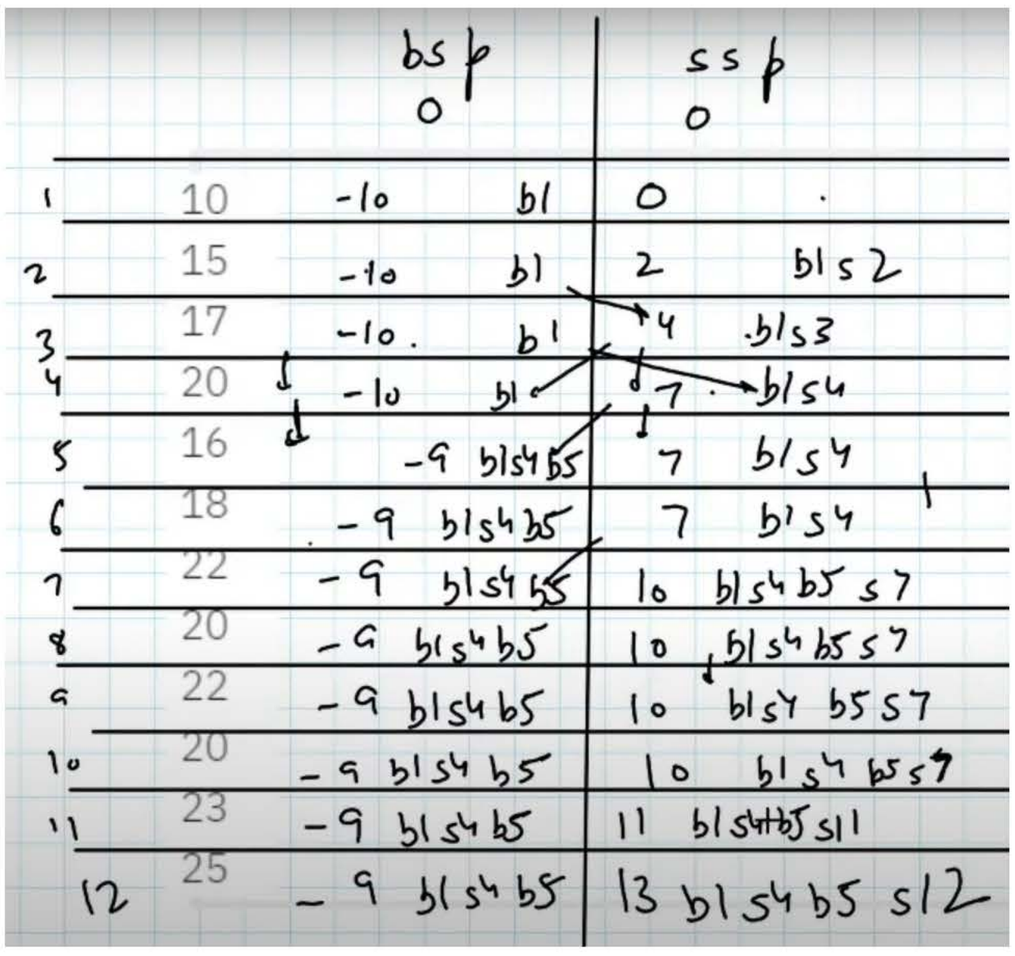
```cpp
//This is just a normal iteration problem and not a DP problem
void Transaction(vector<int> price){
    int lsf = INT_MAX; //Least so far
    int op = 0;        //Overall max profit
    int pist = 0;      //profit of sold today
    for(int i = 0; i < price.size(); i++){
        if(price[i] < lsf)
            lsf = price[i];

        pist = price[i] - lsf;
        if(pist > op)
            op = pist;
    }
    cout << op;
}
```



Buy and Sell Stock -> Infinite transaction

```cpp
int Transaction(vector<int> price){
    int buy = price[0];
    int sell = price[0];
    int profit = 0;
    for(int i = 1; i < price.size(); i++){
        if(price[i] < sell){ //This condition is the first condition and it is a must
            profit += sell-buy;
            sell = buy = price[i];
        }
        else if(price[i] < buy)
            buy = price[i];
        else if(price[i] > sell)
            sell = price[i];

    }
    profit += sell-buy;
    return profit;
}
```

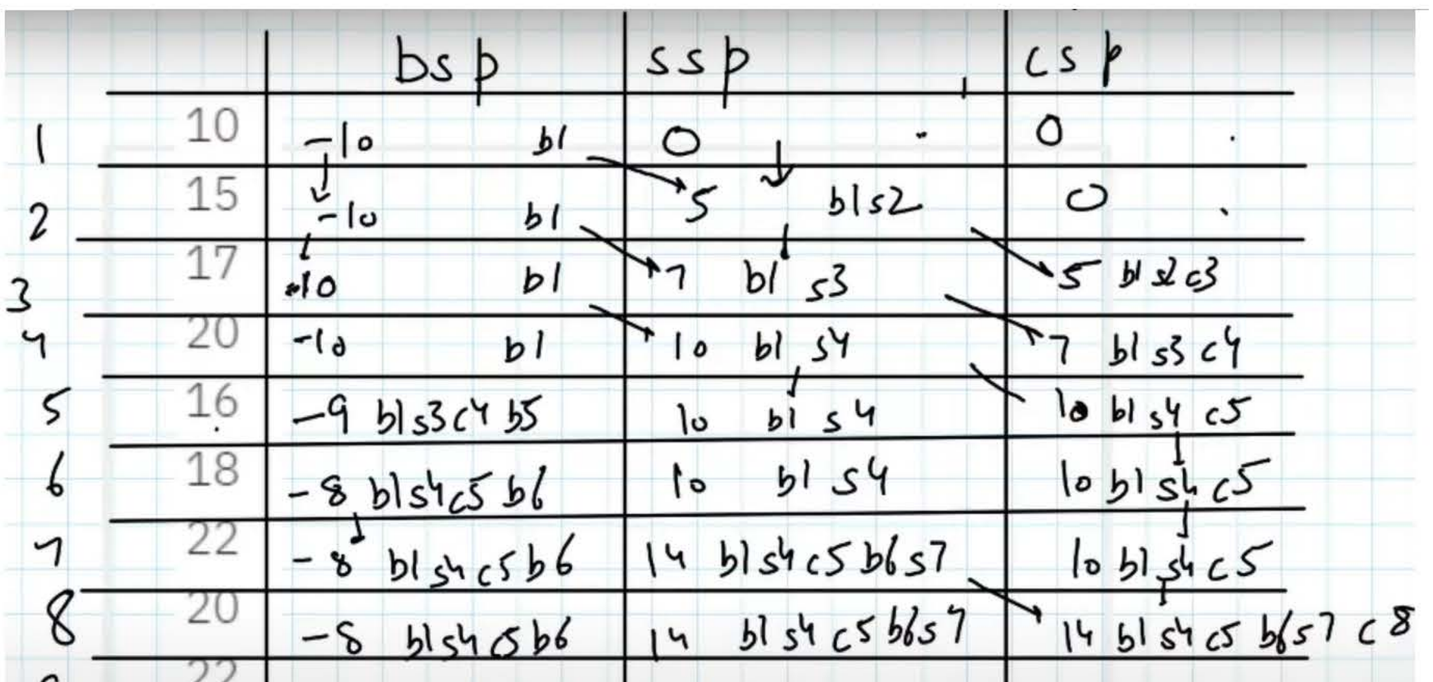
Buy and Sell Stock -> Infinite transaction with Transaction fee




```
Storage -> We require two arrays buy(profit if baught today) & sell(profit if sold today)
Meaning -> buy[i] - net profit if share was baught today
           sell[i] - net profit if share was sold today
traverse & solve ->
          buy[i] :
                We have two options out of which max is chosen
                  1. Buy a share today on prev sold share : buy[i] = sell[i-1] - todays price
                  2. Hold the share baught earlier : buy[i] = buy[i-1]
          sell[i] :
                We have two options out of which max is to be chosen
                  1. Sell a share which was baught earlier : sell[i] = todays price - buy[i-1] - fee
                  2. Do nothing : sell[i] = sell[i-1]
```

```cpp
int transactions(vector<int> arr,int fee){
    int n = arr.size();
    int buy = -arr[0];
    int sell = 0;

    for(int i = 1; i < n; i++){
        int buyToday = sell - arr[i];
        int sellToday = buy + arr[i] - fee;

        buy = max(buy, buyToday);
        sell = max(sell, sellToday);
    }
    return max(buy, sell);
}
```

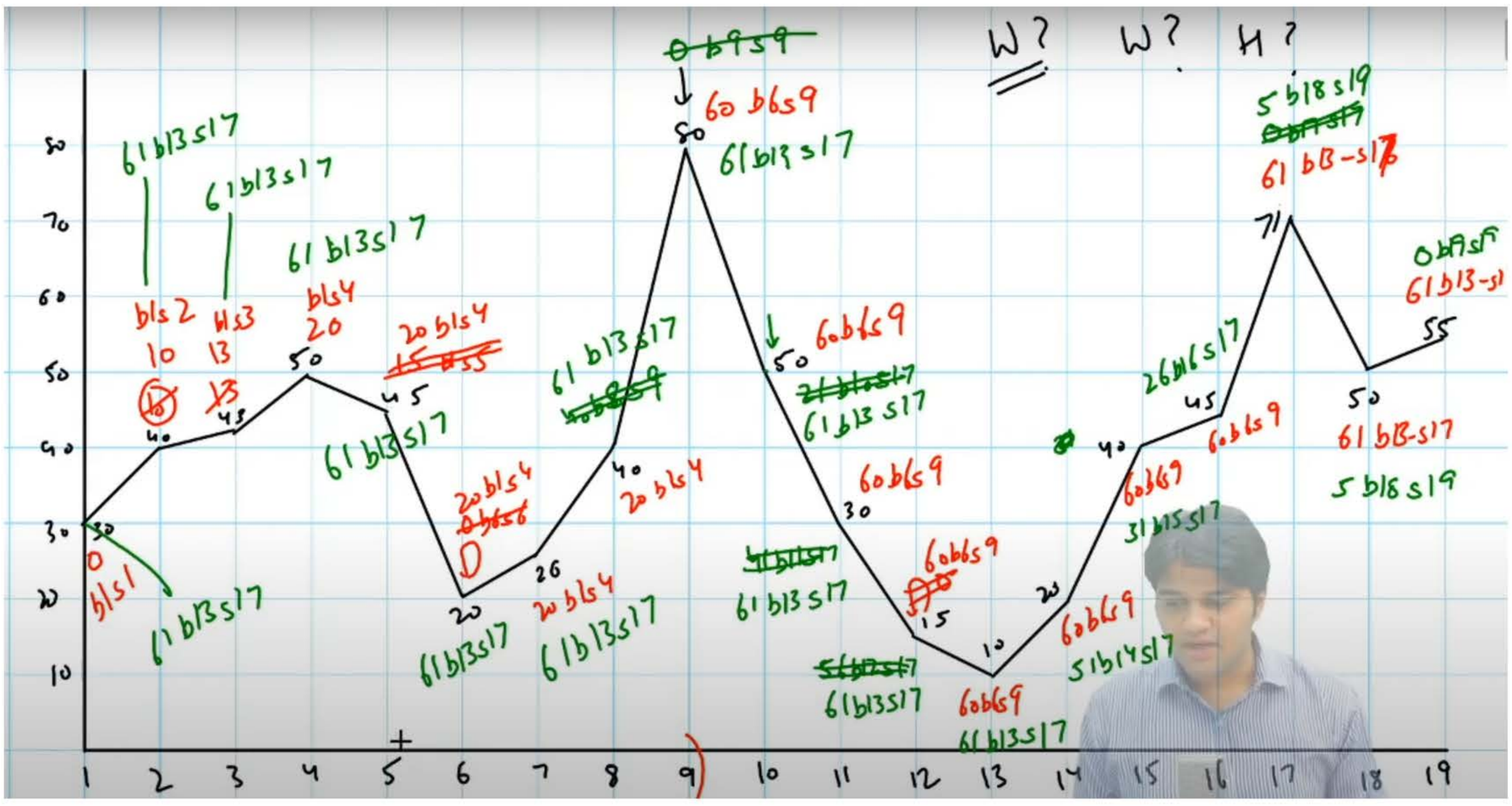
Buy and Sell Stock -> Infinite transaction with Cooldown




```
This is very simular to the previous question, but here we introduse a new state called as cooldown,
Previously we used to buy a stock on previous sell, but here we buy a stock from previous cooldown

Storage -> Three arrays buy(profit if baught today) , sell(profit if sold today) , cooldown(profit if today was cooldown)
Meaning -> buy[i] - net profit if share was baught today
           sell[i] - net profit if share was sold today
           cooldown[i] - net profit if today was a cooldown
traverse & solve ->
          buy[i] :
                We have two options out of which max is chosen
                  1. Buy a share today on prev sold share ie from cooldown : buy[i] = cooldown[i-1] - todays price
                  2. Hold the share baught earlier : buy[i] = buy[i-1]
          sell[i] :
                We have two options out of which max is to be chosen
                  1. Sell a share which was baught earlier : sell[i] = todays price - buy[i-1]
                  2. Do nothing : sell[i] = sell[i-1]
          cool[i] :
                We have two options out of which max is to be chosen
                  1. cool[i] = sell[i-1] coz if today is a cooldown, the share has to be sold yesterday
                  2. Do nothing and carry the prev cooldown as it is cool[i] = cool[i-1]
```

```cpp
int transaction(vector<int> arr){
    int buy = -arr[0];
    int sell = 0;
    int cool = 0;

    for(int i = 1; i < arr.size(); i++){
        buy = max(buy, cool - arr[i]);
        cool = max(cool, sell);
        sell = max(sell, buy + arr[i]);
    }
    return sell;
}
```

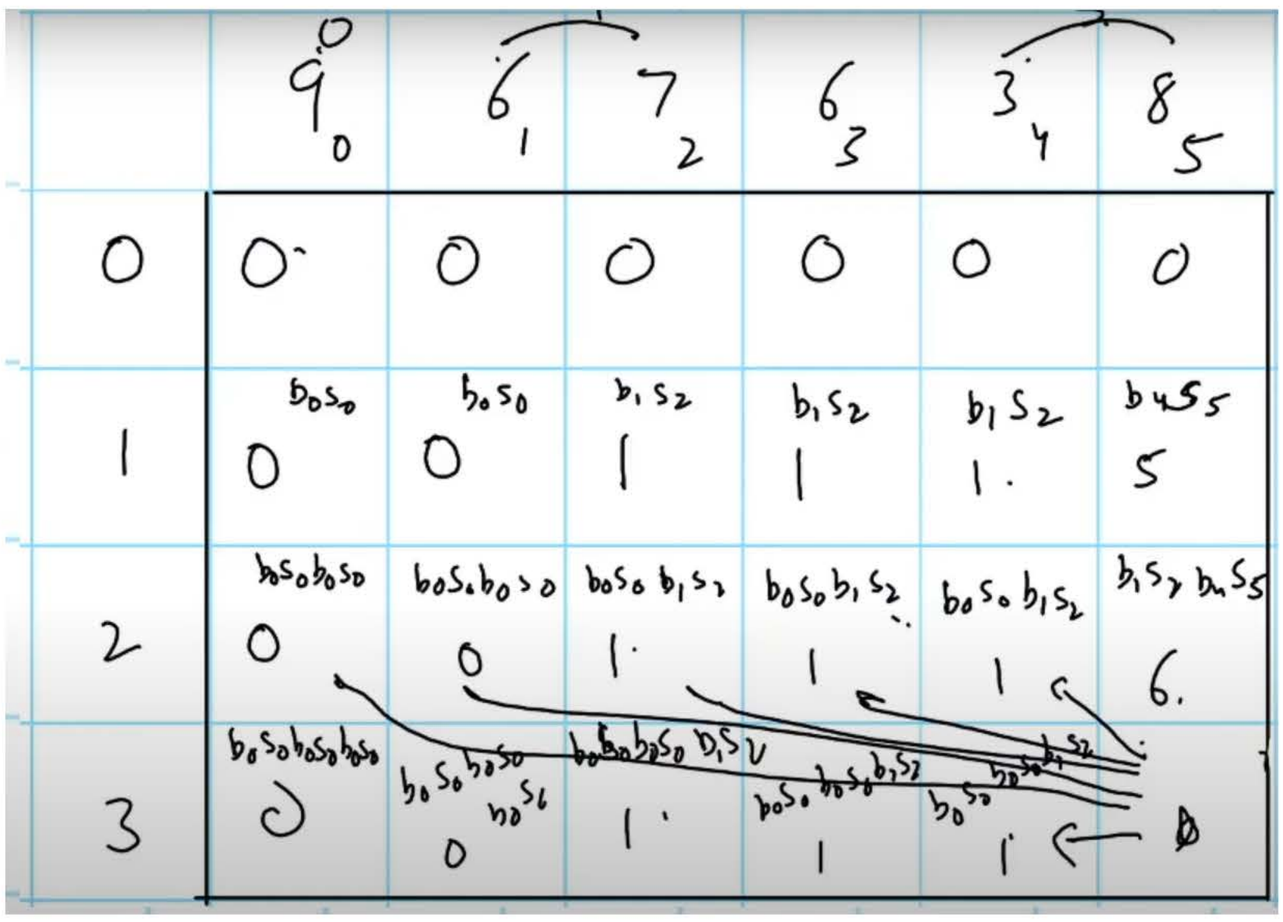
Buy and Sell Stock : 2 Transaction

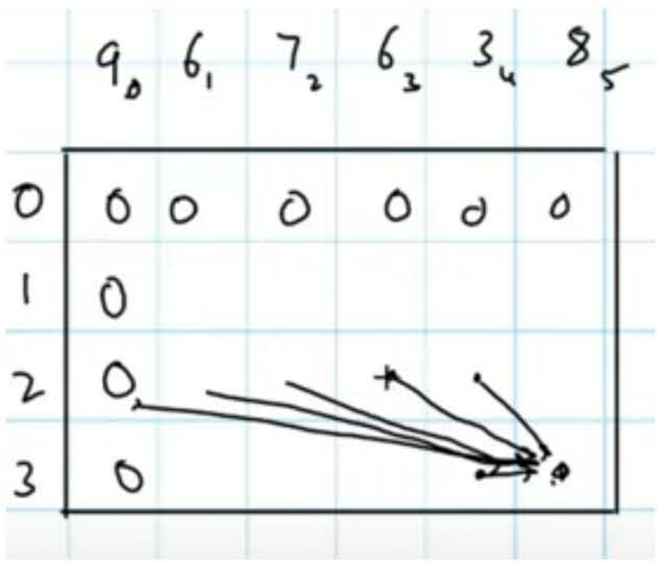



```
This is a variation of 1 Transaction allowed
1st iteration is the same ie Aaj sell karna fixed hain, to hum past me jate the aur least buying price pr buy karte the
2nd iteration is slightly different, we iterate in reverse derection and we do the following
    Aaj buy karna fix he aur hum ab future me jayenge aur maximum price pe sell karenge

Initially, this was my approach. BUT THIS DOESNOT WORK
    int lsf = INT_MAX;  //Least so far
    int op1 = 0, op2=0; //Overall max profit
    int pist = 0;       //Profit if sold today
    for(int i = 0; i < price.size(); i++){
        if(price[i] < lsf)
            lsf = price[i];

        pist = price[i] - lsf;
        if(pist >= op1){
            op2 = op1;
            op1 = pist;
        }
    }
For example : 5 0 3 1 4
        5
         \          4
          \     3   /
           \   / \ /
            \ /   1
             0
   From 0, lsf = 0
   When we reach 3, pist=3 and op1 will become 3
   But after that, lsf is not updated and when we reach 4, pist=4 and op1=4 and op2=3 WHICH IS NOT POSSIBLE BECAUSE FOR BOTH T
   SELLING 0 IS THE BUYING DAY.
   This approach is not taking care about buying day must come after selling day

The following approach works because it is taking care of buying and selling days
How is it taking care?
Say at index i, the first iterations indicates IS DIN SE PEHLE BUY OR SELL KHATAM HOJAYE TO MAX PROFIT KITNE BANENGE
               the second iteration indicates IS DIN KE BAAD  BUY OR SELL KHATAM HOJAYE TO MAX PROFIT KITNE BANENGE
So the second buy is happening only after selling the first stock
```

```
int Transaction(vector<int> arr){
    int lsf = arr[0];               //Least so far
    int mpist = 0;                  // Max profit if sold today
    vector<int> dp(arr.size(), 0); //Max profit in sold until toda
    for(int i = 1; i < arr.size(); i++){
        if(arr[i] < lsf)
        lsf = arr[i];
        mpist = arr[i] - lsf;
        dp[i] = max(dp[i-1], mpist);
    }

    int msf = arr[arr.size()-1];    //Max so far
    int mpibt = 0;                  //Max profit if baught today
    vector<int> dp2(arr.size(), 0); //Max profit in baught until t
    for(int i = arr.size()-2; i >= 0; i--){
        if(arr[i] > msf)
        msf = arr[i];
        mpibt = msf - arr[i];
        dp2[i] = max(dp2[i+1], mpibt);
    }

    int mp = 0; //Max prfit
    for(int i = 0; i < dp.size(); i++){
        if(dp[i] + dp2[i] > mp)
        mp = dp[i] + dp2[i];
    }
    return mp;
}
```

Buy and Sell Stock : K Transaction






```
Storage - 2D array
Meaning: dp[i][j] -> Max profit that can be made in first i days in j transactions
Traverse and solve:
    For example for n = 8 and k = 3, we have 2 options
        1. kth transaction that already occured on n-1 th day ie dp[k][n-1]
        2. We perfomrm the kth transaction on nth day, ie k-1th transaction can be chosen on any day
           from 1st-n-1th day but we
           need to max out the profit
           So we need to choose max between dp[k-1][1]+price[n]-price[1] -> dp[k-1][n-1]+price[n]-price[n-1]
*/
```

```
int transactions(vector<int> price,int K){
    int N = price.size();
    vector<vector<int>> dp(K, vector<int>(N+1, 0));
    for(int k = 1; k <= K; k++){
        for(int n = 1; n <= N; n++){
            int kth = INT_MIN; //kth transaction
            for(int x = n-1; x >= 0; x--){
                kth = max(kth, dp[n-1][x] + price[n] - price[x]);
            }
            dp[k][n] = max(dp[k][n-1], kth);
        }
    }
    return dp[K][N];
}
```

```
//* This is simple, instead of looping to calculate the max of prev row,
//* we are calculating the max as we traverse. Its just like MAX TILL NOW
//* for prev row
int transactionsOptimised(vector<int> price,int K){
    int N = price.size();
    vector<vector<int>> dp(K, vector<int>(N+1, 0));
    for(int k = 1; k <= K; k++){
        int mx = INT_MIN;
        for(int n = 1; n <= N; n++){
            mx = max(mx, dp[k-1][n-1]-price[n-1]);
            dp[k][n] = max(dp[k][n-1], mx+price[n]);
        }
    }
    return dp[K][N];
}
```

Path printing Problems

1. Minimum path cost

```
* Meaning-> dp[i][j] -> Min cost of path from current point to dest point
* Direction-> [n][m] -> [0][0]
* Traverse and solve -> dp[i][j] = cost[i][j] + min(dp[i][j+1], dp[i+1][j]);
*                     We take a move which is minimum of horizontal and vertival move
```

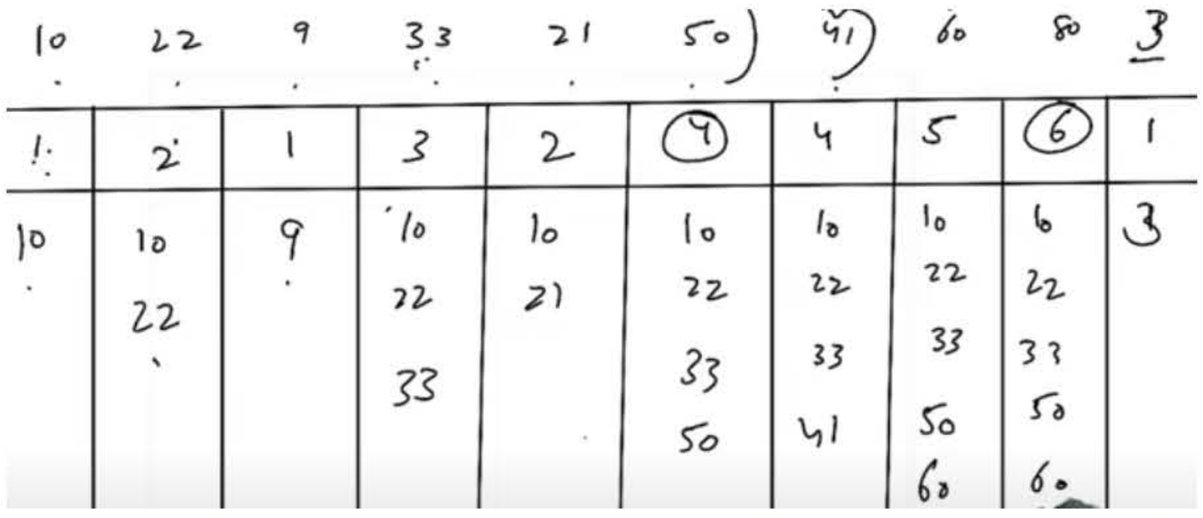
```cpp
void minCostPath(int n, int m, vector<vector<int>> &maze){
    vector<vector<int>> dp(n, vector<int>(m, 0));
    //Minimum cost matrix
    for(int i = n-1; i >= 0; i--){
        for(int j = m-1; j >= 0; j--){
            if(i == n-1 && j == m-1)
                dp[i][j] = maze[i][j];
            else if(i == n-1)
                dp[i][j] = maze[i][j] + dp[i][j+1];
            else if(j == m-1)
                dp[i][j] = maze[i][j] + dp[i+1][j];
            else
                dp[i][j] = maze[i][j] + min(dp[i][j+1], dp[i+1][j]);
        }
    }
    //BFS on cost matrix
    queue<myPair> q;
    q.push(myPair(0, 0, ""));
    while(!q.empty()){
        myPair p = q.front();
        q.pop();

        int i = p.i;
        int j = p.j;
        string s = p.s;

        if(i == n-1 && j == m-1)
            cout << s << endl;
        else if(i == n-1)
            q.push(myPair(i, j+1, s + "H"));
        else if(j == m-1)
            q.push(myPair(i+1, j, s + "V"));
        else{
            if(dp[i][j+1] == dp[i+1][j]){
                q.push(myPair(i, j+1, s + "H"));
                q.push(myPair(i+1, j, s + "V"));
            }
            else if(dp[i][j+1] < dp[i+1][j])
                q.push(myPair(i, j+1, s + "H"));
            else
                q.push(myPair(i+1, j, s + "V"));
        }
    }
}
```

2. 01 KNAPSACK

```cpp
void path_01napsack(int n, vector<int> &weight, vector<int> &value, int cap){
    vector<vector<int>> dp(n+1, vector<int>(cap+1, 0));
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= cap; j++){
            int wt = weight[i-1];
            int val = value[i-1];

            int exc = dp[i-1][j];
            int inc = val + (j >= wt ? dp[i-1][j-wt] : 0);

            dp[i][j] = max(inc, exc);
        }
    }

    cout << dp[n][cap] << endl;

    queue<myPair> q;
    q.push(myPair(n, cap, ""));
    while(!q.empty()){
        myPair p = q.front();
        q.pop();

        int i = p.i;
        int j = p.j;
        string res = p.res;

        if(i == 0 || j == 0)
            cout << res << endl;
        else{
            if(dp[i][j] == dp[i-1][j])
                q.push(myPair(i-1, j, res+ " "));
            else
                q.push(myPair(i-1, j-weight[i-1], to_string(value[i-1]) + " " + res + " "));
        }
    }
}
```

Subsequence Problems

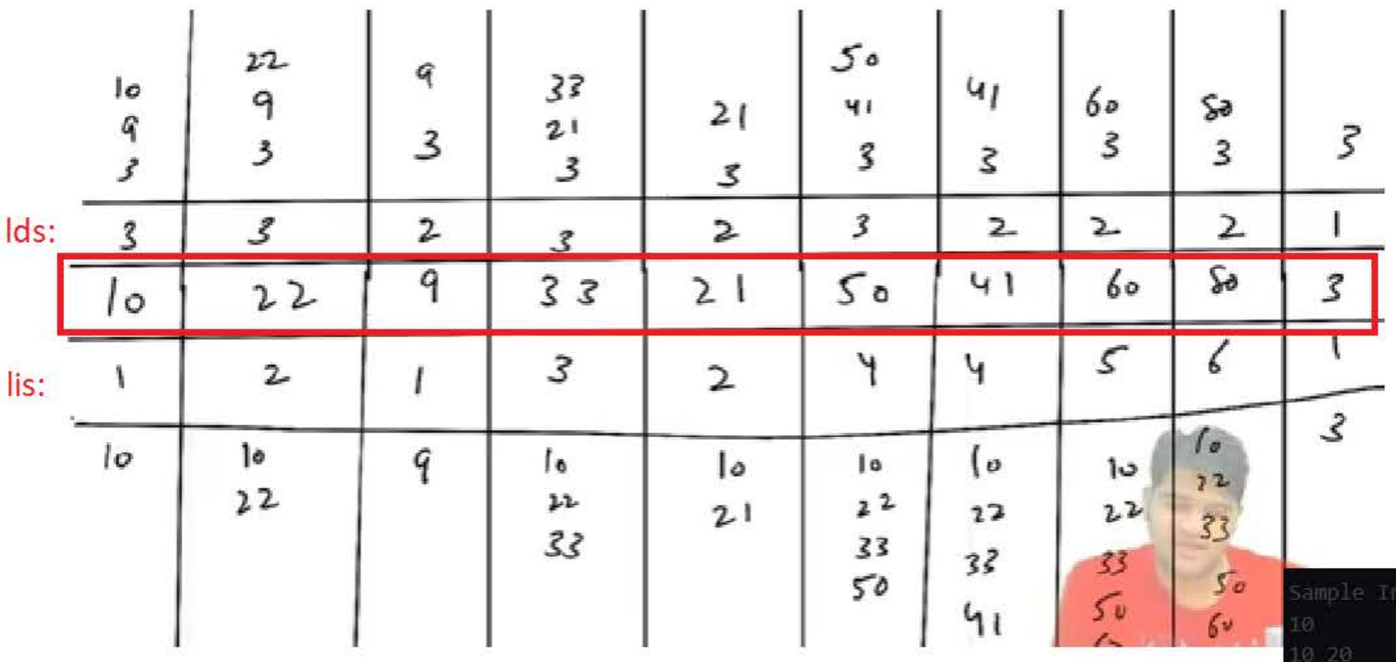
1. Longest Increasing Subsequence




```
Meaning-> dp[i] -> Longest increasing subsequence till ith number that ends with arr[i]
direction-> 0->n

Why did we choose ending with arr[i] -> https://youtu.be/odrfUCS9sQk?t=704
```

```cpp
int lis(vector<int> arr, int n){
    vector<int> dp(n, 0);
    dp[0] = 1;
    int ans = 1;

    for(int i = 1; i < n; i++){
        int mx = 0;
        for(int j = 0; j < i; j++){
            if(arr[j] < arr[i]){
                mx = max(mx, dp[j]);
            }
        }
        dp[i] = mx + 1;
        ans = max(ans, dp[i]);
    }
    return ans;
}
```

2. Longest Bitonic Subsequence

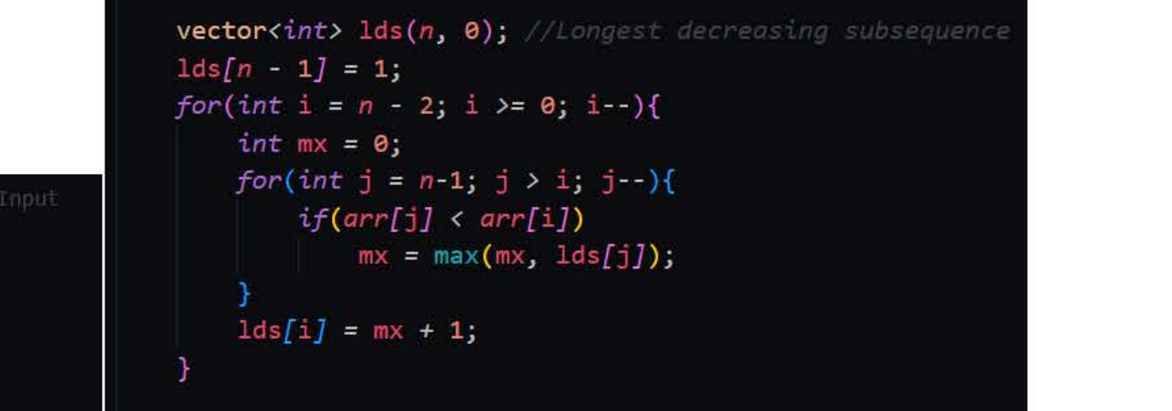



```
Sample Input
10
10 20
2 7
8 15
17 3
21 40
50 4
41 57
60 80
80 90
1 30

Sample Output
7
```

```cpp
int longestBitnicSubseq(vector<int> arr, int n){
    vector<int> lis(n, 0); //Longest increasing subsequence
    lis[0] = 1;
    for(int i = 1; i < n; i++){
        int mx = 0;
        for(int j = 0; j < i; j++){
            if(arr[j] < arr[i])
                mx = max(mx, lis[j]);
        }
        lis[i] = mx + 1;
    }

    vector<int> lds(n, 0); //Longest decreasing subsequence
    lds[n - 1] = 1;
    for(int i = n - 2; i >= 0; i--){
        int mx = 0;
        for(int j = n-1; j > i; j--){
            if(arr[j] < arr[i])
                mx = max(mx, lds[j]);
        }
        lds[i] = mx + 1;
    }

    int ans = 0;
    for(int i = 0; i < n; i++){
        ans = max(ans, lis[i]+lds[i]-1);
    }
    return ans;
}
```

3. Maximum non-overlaping bridges

```
This is just an application of Longest increasing subsequence
Two bridges are non-overlaping if
    1. north of first bridge > noth of second beidge and south of first > south of second OR
    2. north of first bridge < noth of second beidge and south of first < south of second
If we sort the brdges based on north coordinates and apply LIS, we get the answer
```

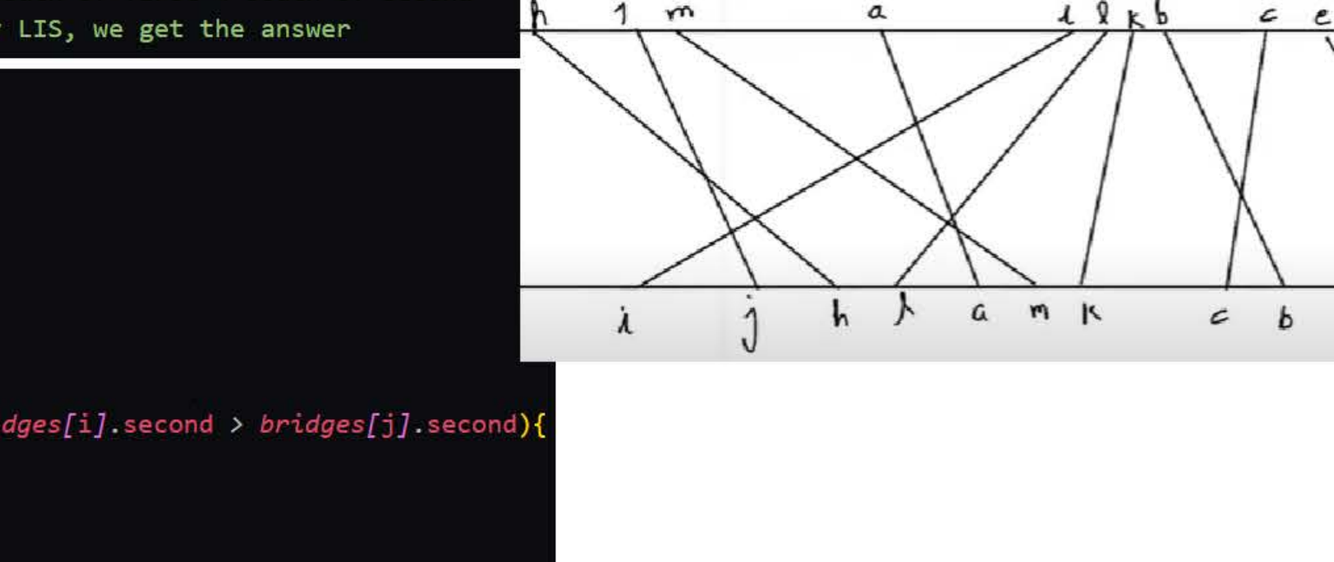



```cpp
int solution(vector<pair<int, int>> &bridges){
    sort(bridges.begin(), bridges.end());

    int n = bridges.size(), ans = 1;
    vector<int> dp(n, 0);
    dp[0] = 1;
    for(int i = 1; i < n; i++){
        int mx = 0;
        for(int j = 0; j < i; j++){
            if(bridges[i].first > bridges[j].first && bridges[i].second > bridges[j].second){
                mx = max(mx, dp[j]);
            }
        }
        dp[i] = mx+1;
        ans = max(ans, dp[i]);
    }
    return ans;
}
```

4. Min Squares

You are given a number N.You have to find the minimum number of squares that sum to N.

```cpp
int solution(int n){
    vector<int> dp(n+1, 0);
    for(int i = 1; i <= n; i++){
        int mn = n;
        for(int j = 1; (i - j*j) >= 0; j++){
            mn = min(mn, dp[i-j*j]);
        }
        dp[i] = mn+1;
    }
    return dp[n];
}
```

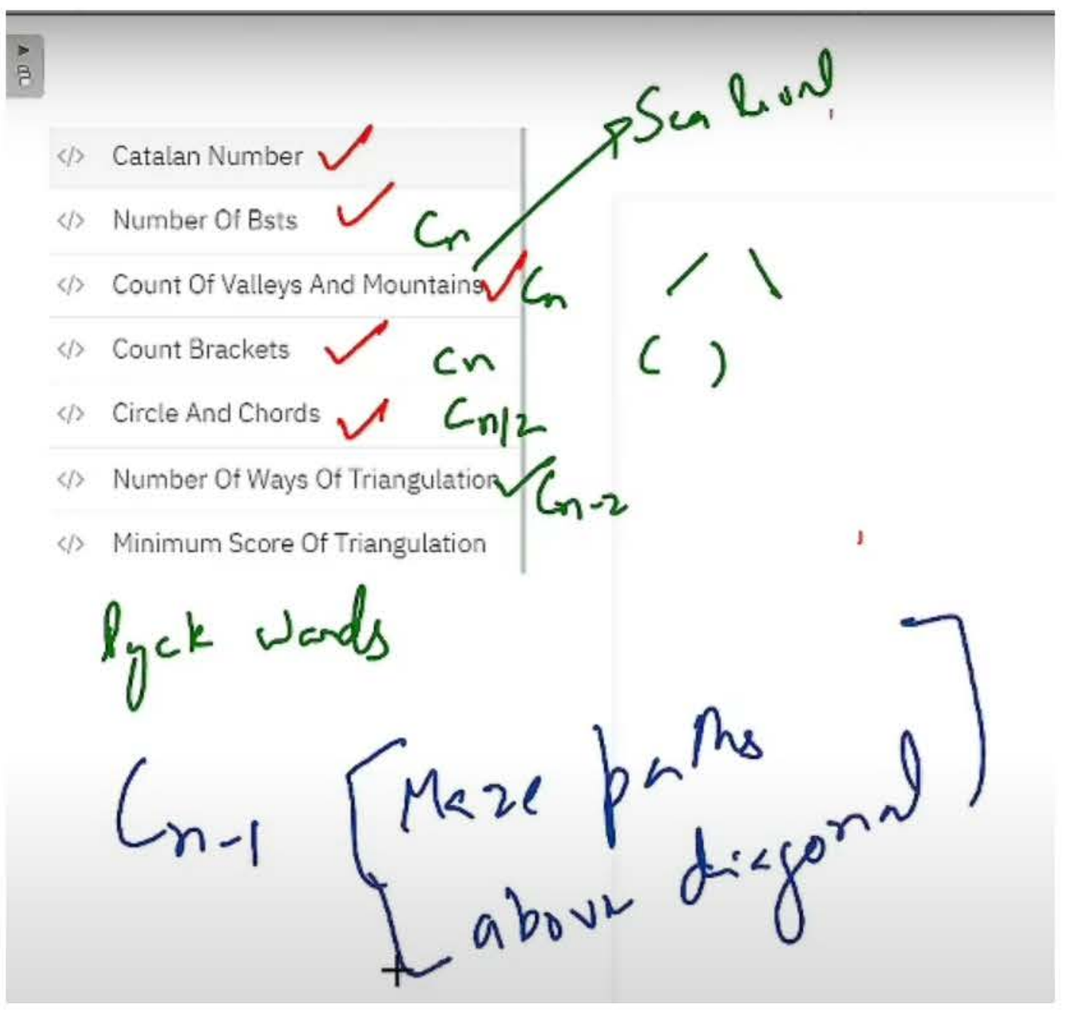


Catalan Numbers

Series : 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, …
Cn -> C0.Cn-1 + C1.Cn-2 + .. + Cn-2.C1 + Cn-1.C0

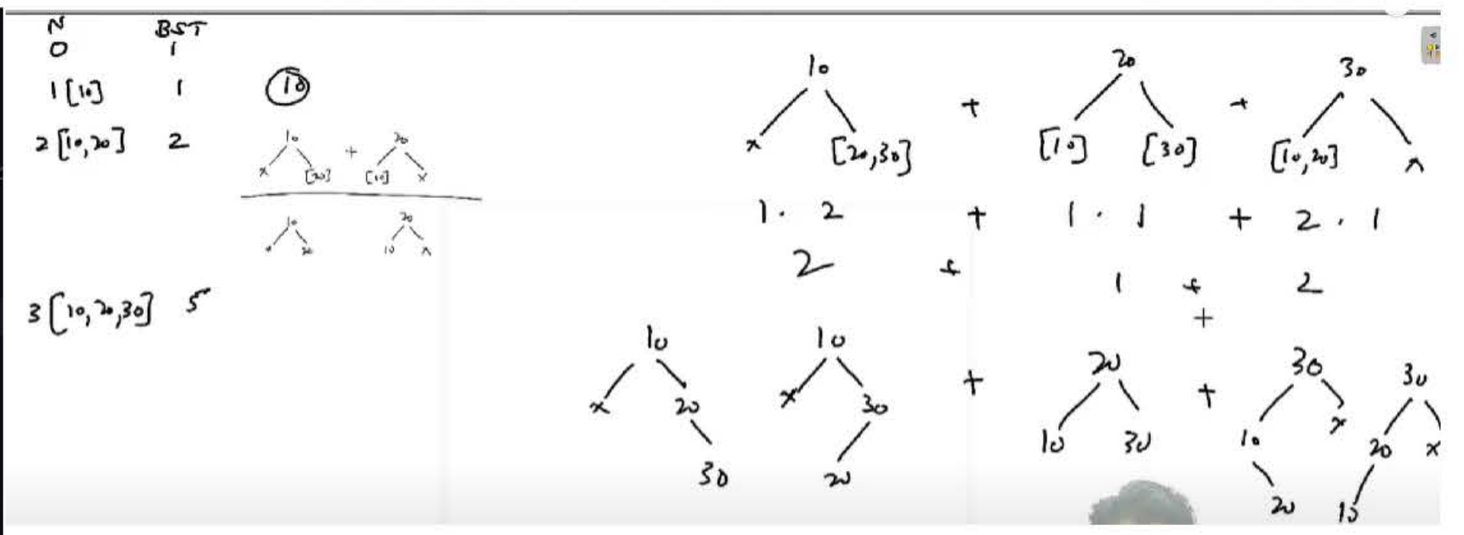
```
C2 = C0C1 + C1C0                        = 2C0C1
C3 = C0C2 + C1C1 + C2C0                 = 2C0C2 + C1C1
C4 = C0C3 + C1C2 + C2C1 + C3C0          = 2C0C3 + 2C1C2
C5 = C0C4 + C1C3 + C2C2 + C3C1 + C4C0   = 2C0C4 + 2C1C3 + C2C2
```

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

    for(int i= 2; i <= n; i++){
        int ith = 0;
        for(int j = 0; j < i/2; j++){
            ith += dp[j] * dp[i-j-1];
        }
        dp[i] = 2 * ith + (i%2 != 0 ? dp[i/2]*dp[i/2] : 0);
    }

    return dp[n];
}
```

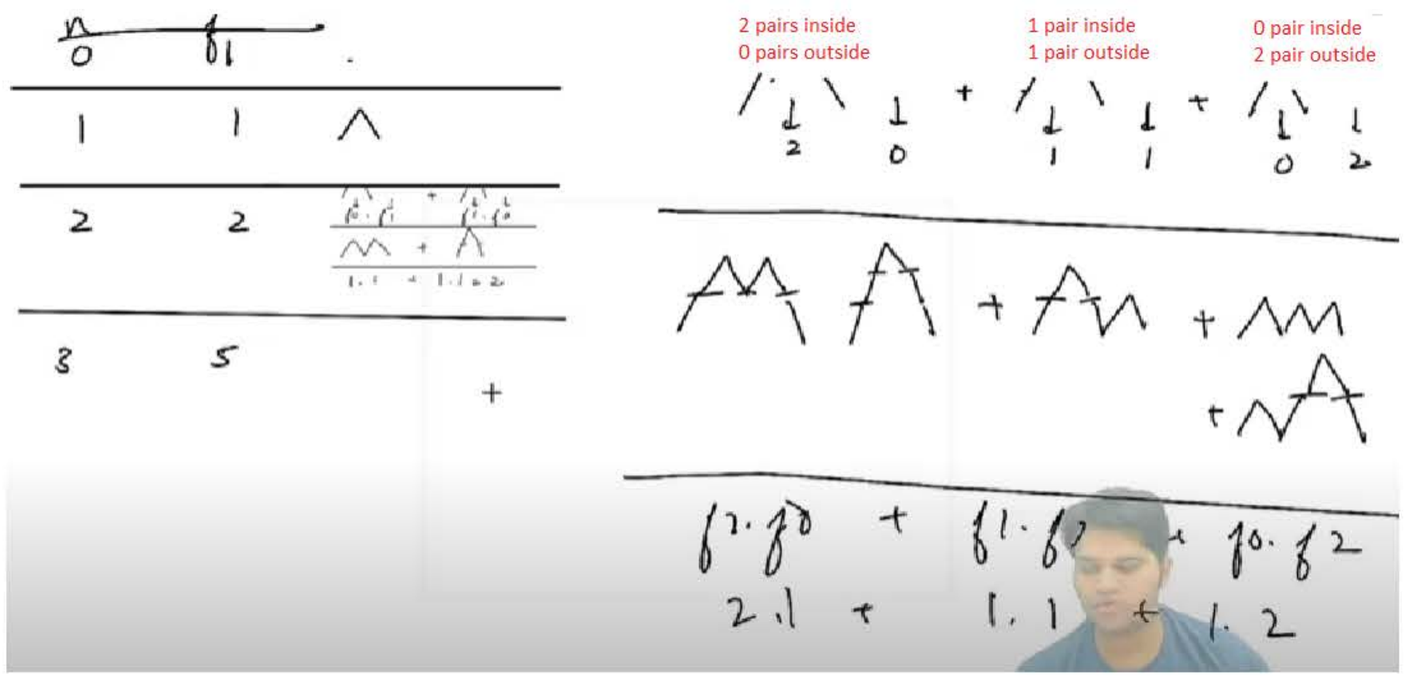


1. No of bst

```
int countbst(int n){
    vector<int> dp(n+1, 0);
    dp[0] = dp[1] = 1;
    for(int i = 2; i <= n; i++){
        for(int j = 1; j <= i; j++){ //If jth element is the ro
            int left = dp[j-1];
            int right = dp[i-j];

            dp[i] += left * right;
        }
    }
    return dp[n];
}

// if we observe the problem it is just catalan numbers
int catalan(int n){
    vector<int> dp(n+1, 0);
    dp[0] = dp[1] = 1;

    for(int i= 2; i <= n; i++){
        int ith = 0;
        for(int j = 0; j < i/2; j++){
            ith += dp[j] * dp[i-j-1];
        }
        dp[i] = 2 * ith + (i%2 != 0 ? dp[i/2]*dp[i/2] : 0);
    }

    return dp[n];
}
```

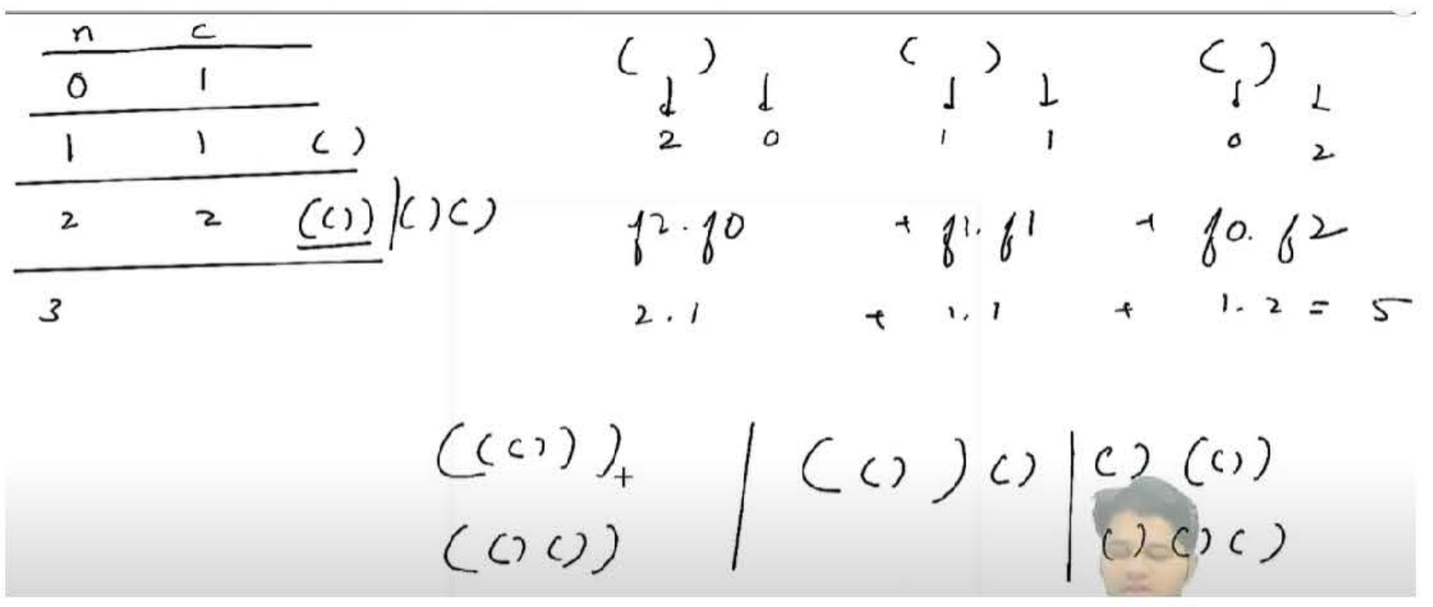


2. Count no of valleys and Moutains



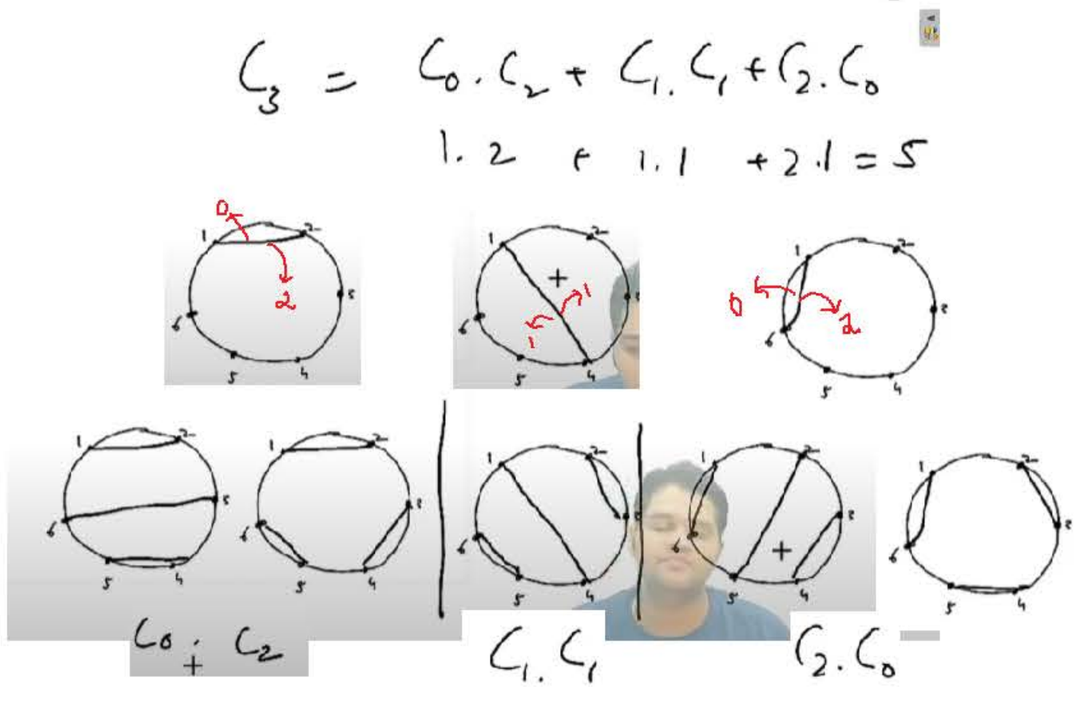
You are given a number n, representing the number of upstrokes / and number of downstrokes .
.You are required to find the number of valleys and mountains you can create using strokes.

3. Count Brackets



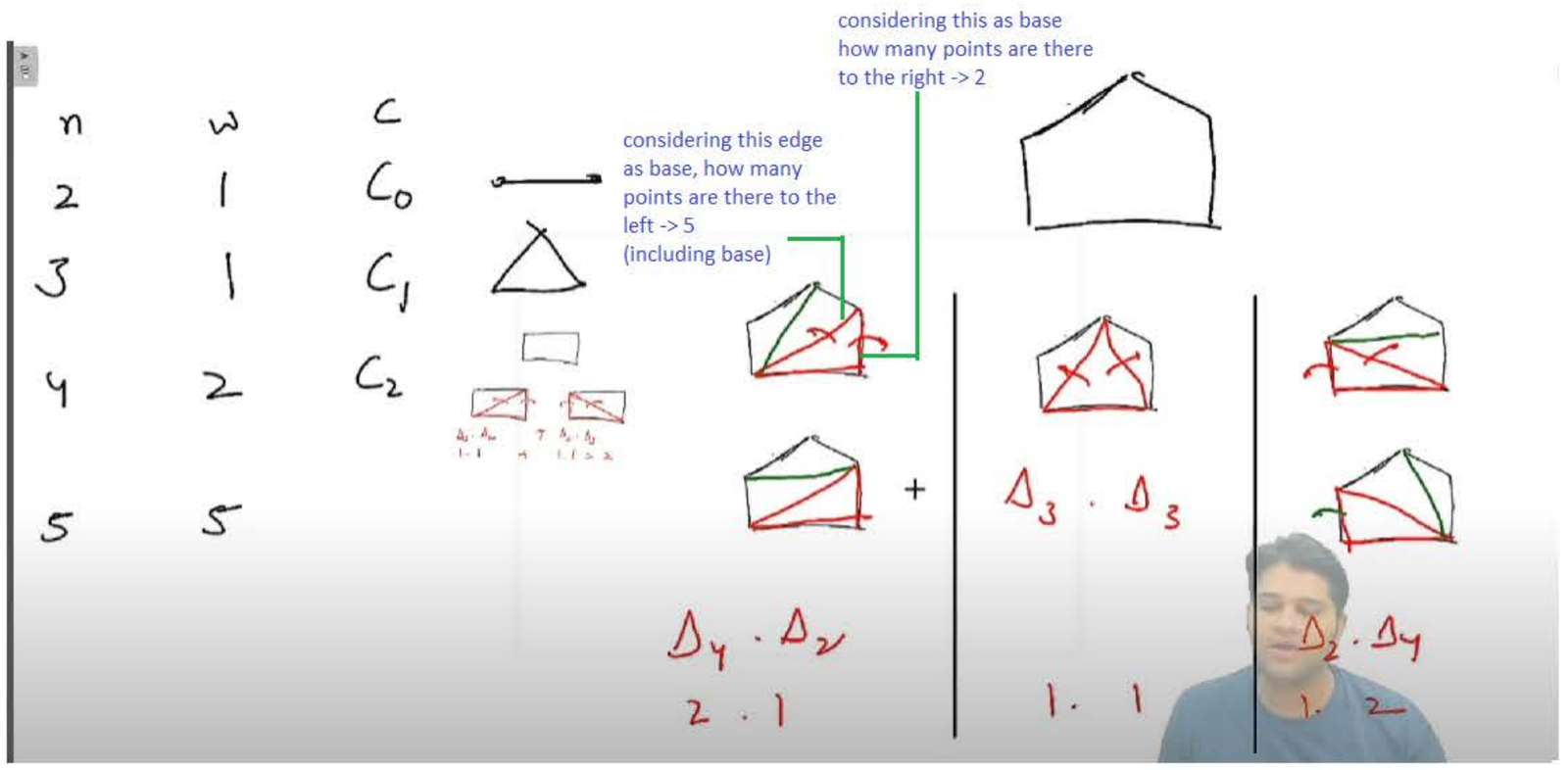
You are given a number n, representing the number of opening brackets (and closing brackets)
You are required to find the number of ways in which you can arrange the brackets if the closing brackets should never exceed opening brackets

4. Non intersecting chords in a circle



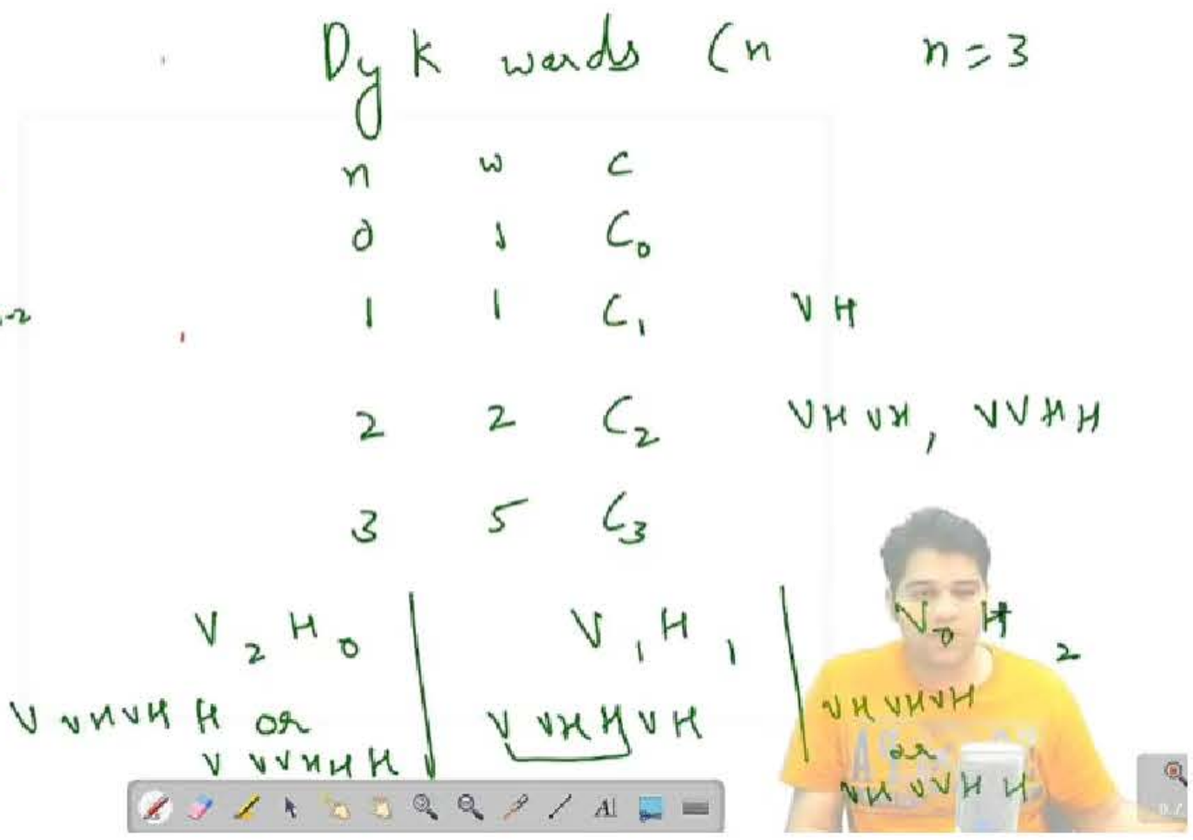
You are given a number N. There are 2*N points on a circle. You have to draw N non-intersecting chords on a circle.
You have to find the number of ways in which these chords can be drawn.

5. No of ways of triangulation



1. You are given a number N, which represents the number of sides in a polygon.
2. You have to find the total number of ways in which the given polygon can be triangulated.

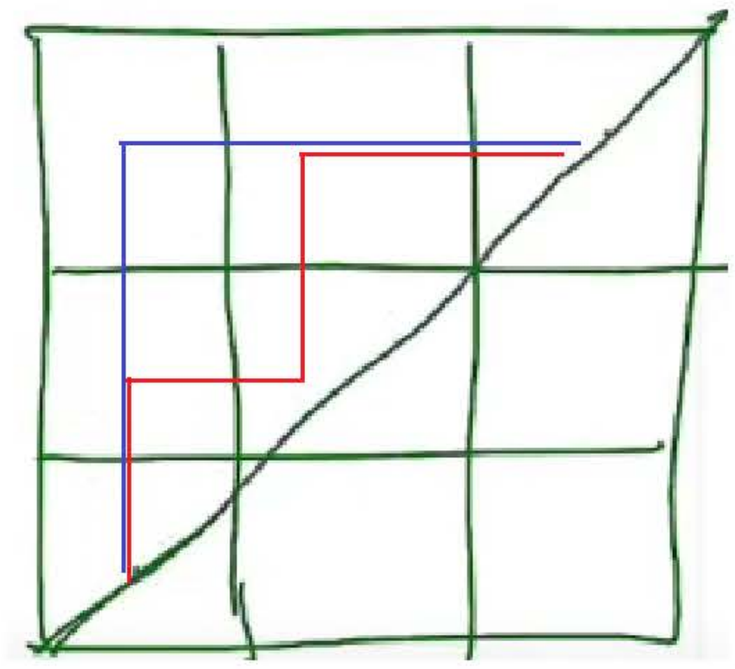
6. Dyck words



Given an integer n, the task is to count Dyck words possible of length n.
A DYCK word is a word containing only characters 'H' and 'V' such that At current idx,
No if V >= No of H

Since V >= H, we can never start out string with H for n = 1 and 2. V (for n=1) and VH for(n=2)
It is almost same as the brackets question we did earlier

7. Maze path -> Catalan(n-1)



Given an nxn matrix, you have to find the no of ways from (n,0) to (0,n). You are allowed to take 1H and 1V step.
You should not cross the diagonal and should always be above the diagonal

* This is nothing but mountain and valley question.
* This can also be said that it is same as the 6th question because, No of V should always be greater than or equal to No of H

Cut Strategy and Gap strategy

These question are ones related level options approach. The first step it to draw the recursion tree and the curring strategy table. Using these two, we can easily map the table and tree to get a formula

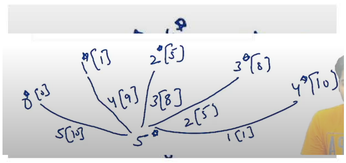
Also poblems which have starting pioint and ending point can be solved using this approach. Eg -> Longest Palindrommic Subseq OR substring

1. Rod cutting

Given a rod of length N inches and an array of prices, price[]. price[i] denotes the value of a piece of length i. Determine the maximum value obtainable by cutting up the rod and selling the pieces.

```
length   | 1   2   3   4   5   6   7   8
--------------------------------------------
price    | 1   5   8   9  10  17  17  20
```

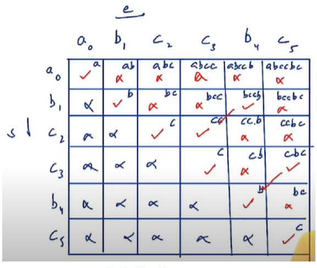
Cutting strategy in 1D array



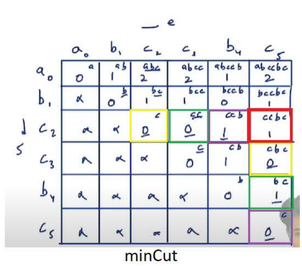
```
int solution2(vector<int> &prices, int n){
    vector<int> dp (n+1, 0);
    for(int len = 1; len <= n; len++){
        dp[len] = prices[len];
        int l = 1;
        int r = len-1;
        while(l < r){
            dp[len] = max(dp[len], dp[l]+dp[r]);
            l++;
            r--;
        }
    }
    return dp[n];
}
```

2. Minimum palindromic cut

Given a string s, partition s such that every substring of the partition is a palindrome



isPalindrome

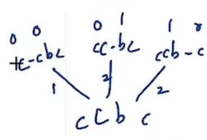


minCut

```
// Min cut calculation
vector<vector<int>> dp(n, vector<int>(n, n));
for (int g = 0; g < n; g++) {
    for (int i = 0, j = g; i < n && j < n; i++, j++) {
        if (isPalindrome[i][j])
            dp[i][j] = 0;
        else {
            for (int k = i; k < j; k++) {
                int left = dp[i][k];
                int right = dp[k + 1][j];
                dp[i][j] = min(dp[i][j], left + right + 1);
            }
        }
    }
}
return dp[0][n - 1];
```

O(N^3)

```
A string is palindrom if
1. Len = 1 => it is palindrome
2. Len = 2 => if first and last char is same
3. Len > 2 => first and last character is same ie
              s[0]=s[n] and s[1:n-1] is pallindrome
```

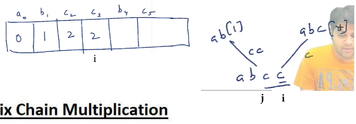


Here we have option to cut after every character

Optimization : We have to make a cut only when we encounter a palindrome and not after every character

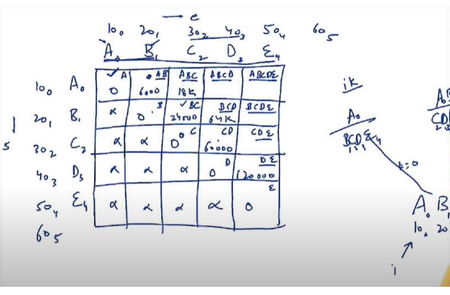
```
// Is palindrome
vector<vector<bool>> isPalindrome(n, vector<bool>(n, false));
for (int g = 0; g < n; g++) {
    for (int i = 0, j = g; i < n && j < n; i++, j++) {
        if (g == 0)
            isPalindrome[i][j] = true;
        else if (g == 1)
            isPalindrome[i][j] = (s[i] == s[j]);
        else
            isPalindrome[i][j] = (s[i] == s[j] && isPalindrome[i + 1][j - 1]);
    }
}
```

```
vector<int> dp(n, n);
dp[0] = 0;
for(int i = 1; i < n; i++){
    if(isPalindrome[0][i]){ // If substring[0->i] is palindrome we need 0 cuts
        dp[i] = 0;
        continue;
    }
    //Cutting only when substr[j,i] is a palindrome
    for(int j = i; j >= 0; j--){
        if(isPalindrome[j][i])
            dp[i] = min(dp[i], dp[j-1]+1);
    }
}
return dp[n-1];
```



3. Matrix Chain Multiplication

Given the dimension of a sequence of matrices in an array arr[], where the dimension of the ith matrix is (arr[i-1] * arr[i]), the task is to find the most efficient way to multiply these matrices together such that the total number of element multiplications is minimum.



```
int solution(vector<int> arr){
    int n = arr.size() - 1;
    vector<vector<int>> dp(n, vector<int>(n, 1e9));
    for(int g = 0; g < n; g++){
        for(int i = 0, j = g; j < n; i++, j++){
            if(g == 0)
                dp[i][j] = 0;
            else if(g == 1)
                dp[i][j] = arr[i] * arr[i+1] * arr[i+2];
            else{
                for(int k = i; k < j; k++){
                    //dp -> left half: i, k    right half: k+1  j
                    //arr-> left half: i, k+1  right half: k+1  j+1
                    int left = dp[i][k];
                    int right = dp[k+1][j];
                    int currCost = arr[i] * arr[k+1] * arr[j+1];
                    dp[i][j] = min(dp[i][j], left+right+currCost);
                }
            }
        }
    }
    return dp[0][n-1];
}
```

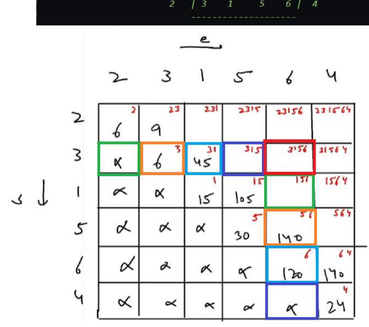
4. Burst Baloon

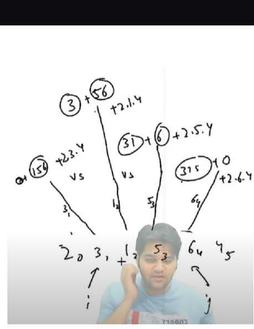
You are given n balloons, indexed from 0 to n - 1. Each balloon is painted with a number on it represented by an array nums. You are asked to burst all the balloons.

If you burst the ith balloon, you will get nums[i - 1] * nums[i] * nums[i + 1] coins. If i - 1 or i + 1 goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it.

Return the maximum coins you can collect by bursting the balloons wisely.

```
We make use of cutting strategy itself, but here the storage and meaning is slightly different
storage & Meaning -> dp[i][j] = maximum valus that can be obtainied by bursting baloons i->j
          Eg -> 2 3 1 5 6 4
          dp[1][4] -> This would store max value obtained by bursting  3 1 5 6, BUT 3 1 5 6 ARE NOT IN COMPLETE ISOLATION
          BY THAT WE MEAN THAT MAX MONEY THAT CAN BE COLLECTED BY BURSTING 3 1 5 6 IN COMPLETE ARRAY

       Simple example -> dp[1][1] -> Max money obtained by bursting baloon 3
          If we considered 3 is in isolation, the ans would be only 3
          but here we are not considering 3 in complete isolation, we are telling that dp[1][1] is money obtained by
sting 3 in complete array ie dp[1][1] = 2*3*1

Bursting options -> For each dp[i][j] ie subarray arr[i:j], we have options to burst any baloon
          say in subarr : 3 1 5 6 we have options to burst 3 OR 1 OR 5 OR 6
          WHILE BURSTING WE WILL CONSIDER IT TO BE BURST LAST AMONG ALL IN SUB ARRAY

                    0+       dp(3)+  dp(3,1)+  dp(3,1,5)+
LEFT+RIGHT+CURR     dp(1,5,6)+ dp(5,6)+  dp(6)+      0+
                    3*2*4     2*1*4   2*5*4      2*6*4
                      \         |       |        /
LAST TO BE BURST :    3 IS \  1 IS |    |5 IS  / 6 IS
                      LAST \ LAST |    |LAST /  LAST
                           \      |    |   /
                            -------------------
                        2   | 3    1     5    6 | 4
                            -------------------
```





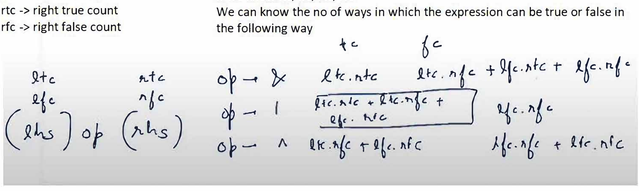
```
int solution(vector<int> &arr){
    int n = arr.size();
    vector<vector<int>> dp(n, vector<int> (n, 0));
    for(int g = 0; g < n; g++){
        for(int i = 0, j = g; j < n; i++, j++){

            //Elements outside subarray (2 & 4 in the above example)
            int l = i == 0   ? 1 : arr[i-1];
            int r = j == n-1 ? 1 : arr[j+1];

            //Processing subarray
            for(int k = i; k <= j; k++){
                int left  = k == i ? 0 : dp[i][k-1];
                int right = k == j ? 0 : dp[k+1][j];
                int curr  = l * arr[k] * r;
                dp[i][j] = max(dp[i][j], left + right +curr);
            }
        }
    }
    return dp[0][n-1];
}
```

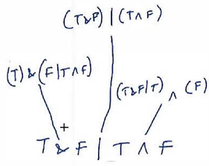
5. Boolean Paranthisation

Given a boolean expression, Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.

ltc -> left true count
lfc -> left false count
rtc -> right true count
rfc -> right false count

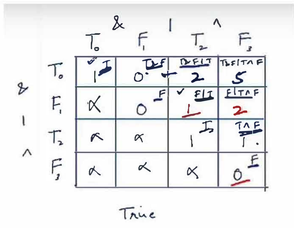
For an expression LHS op RHS
If we know ltc, lfc, rtc, rfc
We can know the no of ways in which the expression can be true or false in the following way

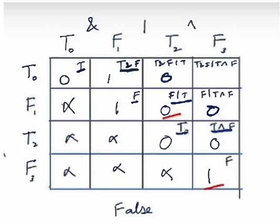





```
int solution(string str){
    string oprnd, oper;
    for(char ch : str){
        if(ch == 'T' || ch == 'F')
            oprnd.push_back(ch);
        else
            oper.push_back(ch);
    }

    int n = oprnd.size();
    vector<vector<int>> dpt(n, vector<int> (n, 0));
    vector<vector<int>> dpf(n, vector<int> (n, 0));

    for(int g = 0; g < n; g++){
        for(int i = 0, j = g; j < n; i++, j++){
            if(g == 0){
                if(oprnd[i] == 'T')
                    dpt[i][j] = 1;
                else
                    dpf[i][j] = 1;
            }
            else{
                for(int k = i; k < j; k++){
                    char op = oper[k];
                    int ltc = dpt[i][k];
                    int rtc = dpt[k+1][j];
                    int lfc = dpf[i][k];
                    int rfc = dpf[k+1][j];

                    if (op == '&') {
                        dpt[i][j] += ltc * rtc;
                        dpf[i][j] += ltc * rfc + lfc * rtc + lfc * rfc;
                    }
                    else if (op == '|') {
                        dpt[i][j] += ltc * rtc + ltc * rfc + lfc * rtc;
                        dpf[i][j] += lfc * rfc;
                    }
                    else if (op == '^') {
                        dpt[i][j] += ltc * rfc + lfc * rtc;
                        dpf[i][j] += ltc * rtc + lfc * rfc;
                    }
                }
            }
        }
    }
    return dpt[0][n-1];
}
```

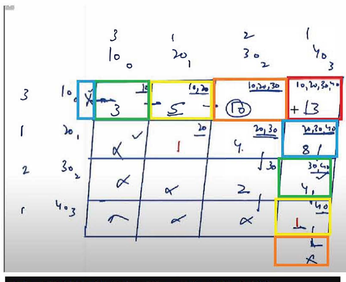
So we have to create 2 tables
1. Which has count of true
2. Which has count of false

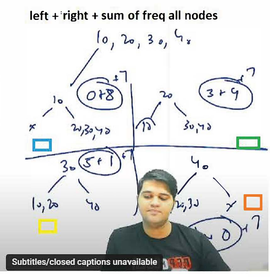




6. Optimal BST

You are given a two arrays :- Nodes, No of times a perticular node is searched
Your task is to construct an optimal BST in which the no of operations to seacrh is minimized. Return the minimum operations






```
int solution(vector<int> nodes, vector<int> freq){
    int n = nodes.size();
    vector<vector<int>> dp(n, vector<int> (n, 1e9));
    for(int g = 0; g < n; g++){
        for(int i = 0, j = g; j < n; i++, j++){
            if(g == 0){
                dp[i][j] = freq[i];
            }
            else{
                int freqSum = 0;
                for(int x = i; x <= j; x++){
                    freqSum += freq[x];
                }
                for(int k = i; k <= j; k++){
                    int left = k == i ? 0 : dp[i][k-1];
                    int right = k == j ? 0 : dp[k+1][j];
                    int curr = left + right + freqSum;

                    dp[i][j] = min(dp[i][j], curr);
                }
            }
        }
    }
    return dp[0][n-1];
}
```

Here the cut in cutting strategy is to make a particular node root
Why -> left + right + freq of all node

Here
left = 0
right = dp(bc)

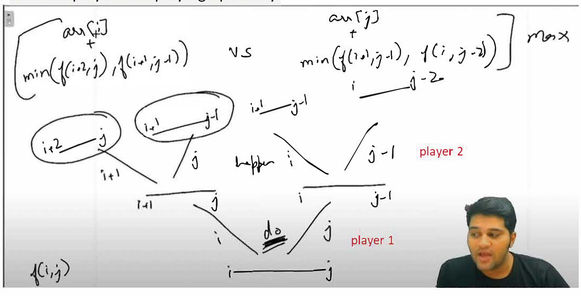
Say that this is the optimal subtree bc
No of operations to search
c -> 1
b -> 2
if c' is frequency of c and b' is frequency of b
then no of operations = c' +2b'

When we consider tree abc, no of search operations for c and b will inrease by **1** for every search because of edge a ->c
So no of search operations of c -> 2c'
b -> 3b'
So now total no of search operations for a,b,c comnbined = a' +2c' +3b'
= a' + c' + b' + (c' +2b')
= a' +b' +c' + OPERATION(bc)

Therefore NO OF OPERATIONS = left + eight + freq of all elements

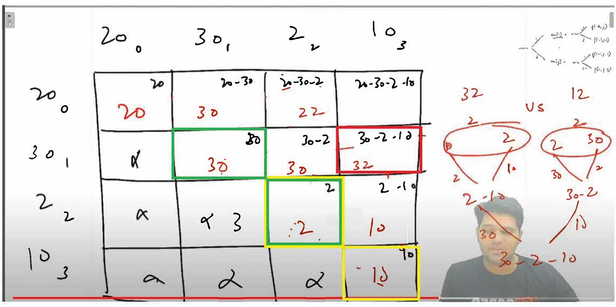
Optimal Strategy to win a game

You are given an array A of size N. The array contains integers and is of even length. The elements of the array represent N coin of values V1, V2, ...Vn. You play against an opponent in an alternating way. In each turn, a player selects either the first or last coin from the row, removes it from the row permanently, and receives the value of the coin. You need to determine the maximum possible amount of money you can win if you go first.
Note: Both the players are playing optimally.



When we are choosing, we have to choose the best, when the other is choosing, we are getting the worst

by that we mean that in arr[i -> j]
1. We can choose best of i and j
so upon our choice, the opponent gets either i+1 -> j or i->j-1
2. Opponent also chooses the best => We get the worst when opponent chooses

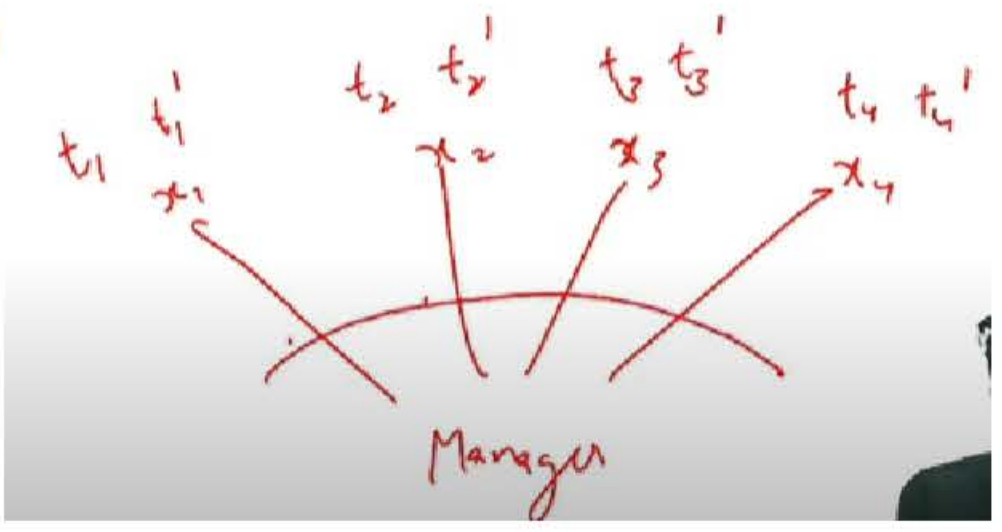


```
int solution (vector<int> coins){
    int n = coins.size();
    vector<vector<int>> dp(n, vector<int> (n, 0));

    for(int g = 0; g < n; g++){
        for(int i = 0, j = g; j < n; i++, j++){
            if(g == 0)
                dp[i][j] = coins[i];
            else if(g == 1)
                dp[i][j] = max(coins[i], coins[j]);
            else{

                int val1 = coins[i] + min(dp[i+2][j], dp[i+1][j-1]);
                int val2 = coins[j] + min(dp[i+1][j-1], dp[i][j-2]);

                dp[i][j] = max(val1, val2);
            }
        }
    }
    return dp[0][n-1];
}
```

Prombels that are dependent on luck

Suppose there is a manager who has to complete a project P and he has 4 employess from which he can choose one to do this job. He has to give the client a guarenteed timeline in which the project can be delevered.
Each employee works differently on different days, some days optimally and some days not-optimal. But the manager has to give the clinet a guarenteed end date ie guarenteed best time

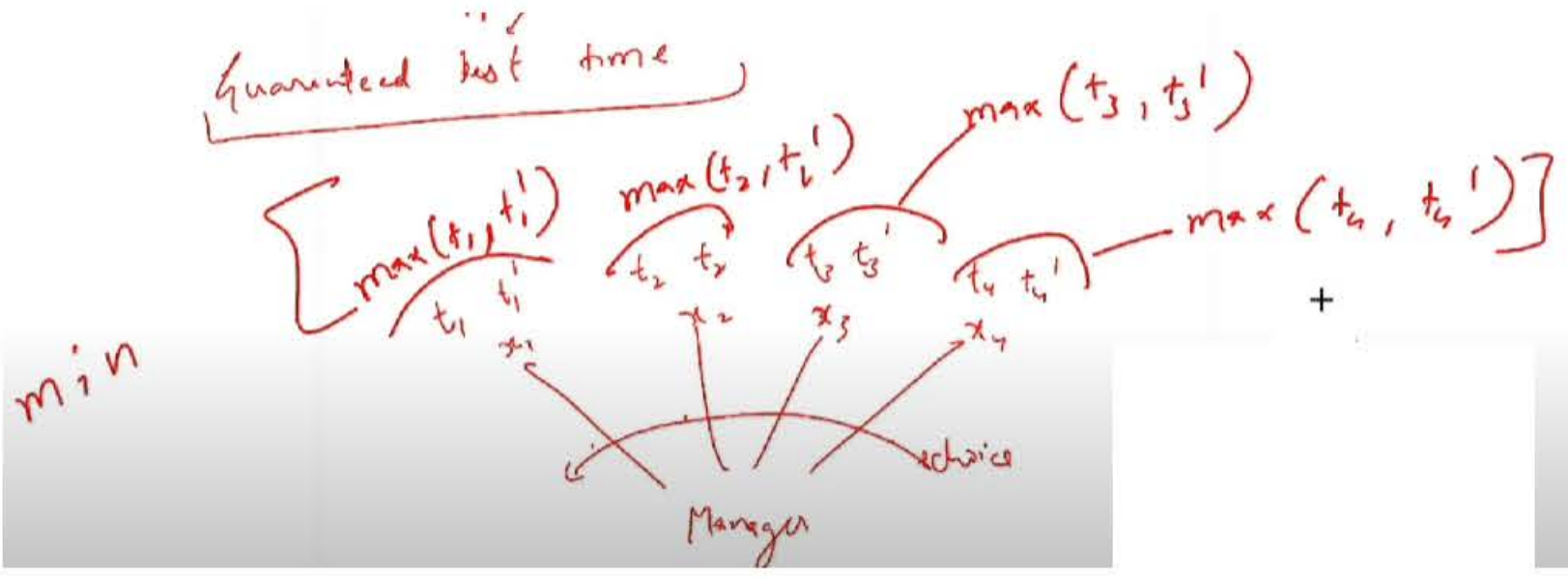
Here we are asked two things :
1. best time
2. guarenteed



Choosing an employee among the 4 is managers **choice**, but how optimaly the employee works is **luck**. Som eday he can work optimaally in t time and some day not optimally in t' time.
So agar manager ko guarenteed time dena hai na to jaha bhi luck invoved hain, wo **luck ko against manega**

GOLDE RULE IN ESTIMATION/LUCK BASED PROBLEMS:
Jo tumhare control me hain, use best kardo
Jo tumhare control me nahi hain, use worst manlo
Guarenteed best time = BEST OF THE WORST

So in this problem, we consider the worst time of each employee -> Maximising Worst
Manager chooses best of the worst -> Smallest of the worst



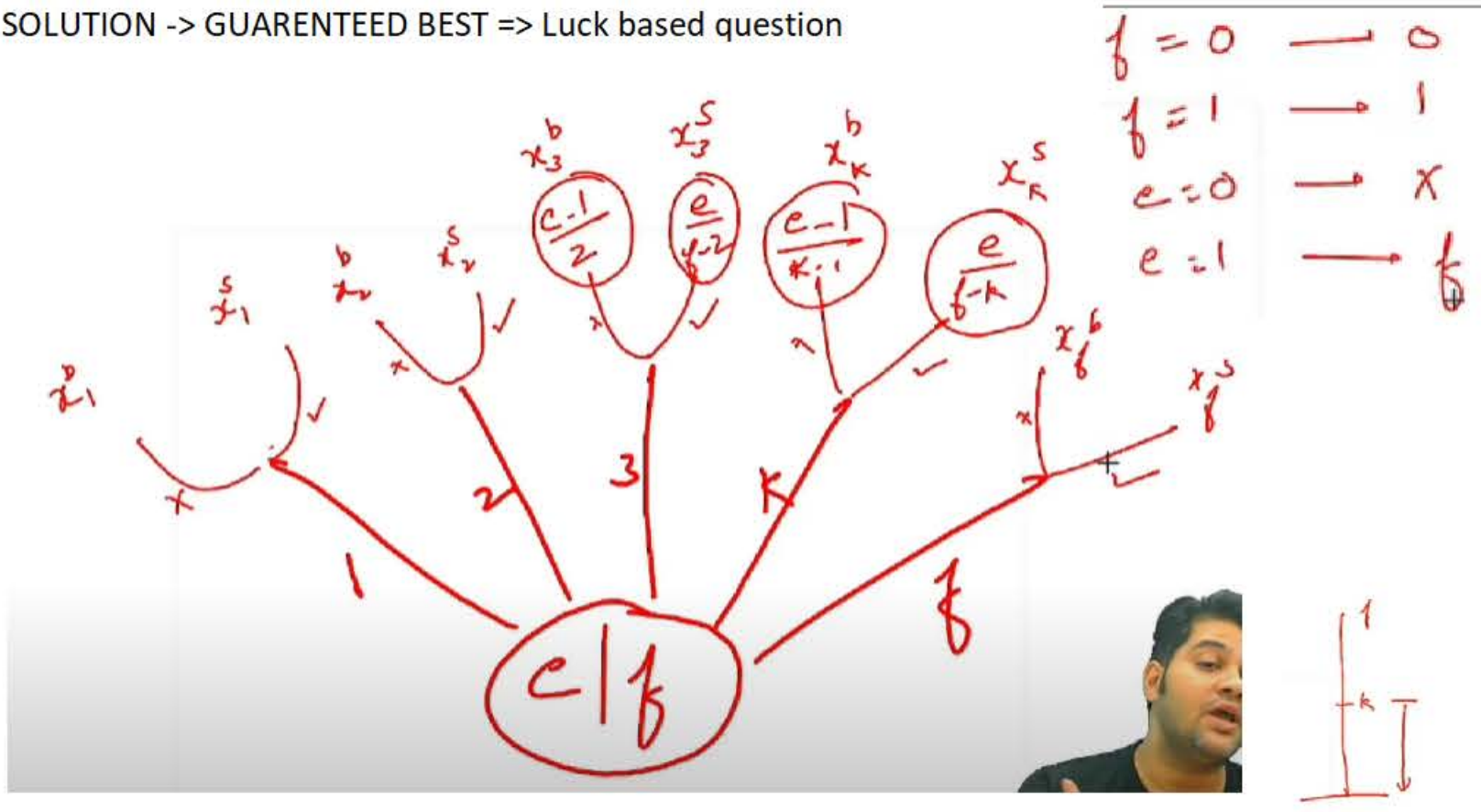
solution -> min (max(t1, t1'), max(t2, t2'), max(t3, t3'), max(t4, t4'))

Egg Drop problem

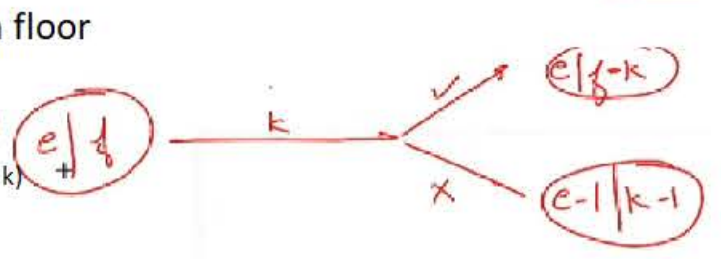
Given **e** eggs and **f** floors. An egg thrown from a floor k can either break or not break. There exists a critical floor **c** where 0 <= c <= f such that any egg dropped at a floor higher than c will break, and any egg dropped at or below floor c will not break.
* An egg that survives a fall can be used again.
* A broken egg must be discarded.
Your task is to find the **minimum** no of attempts required to find the critical floor.

Base cases

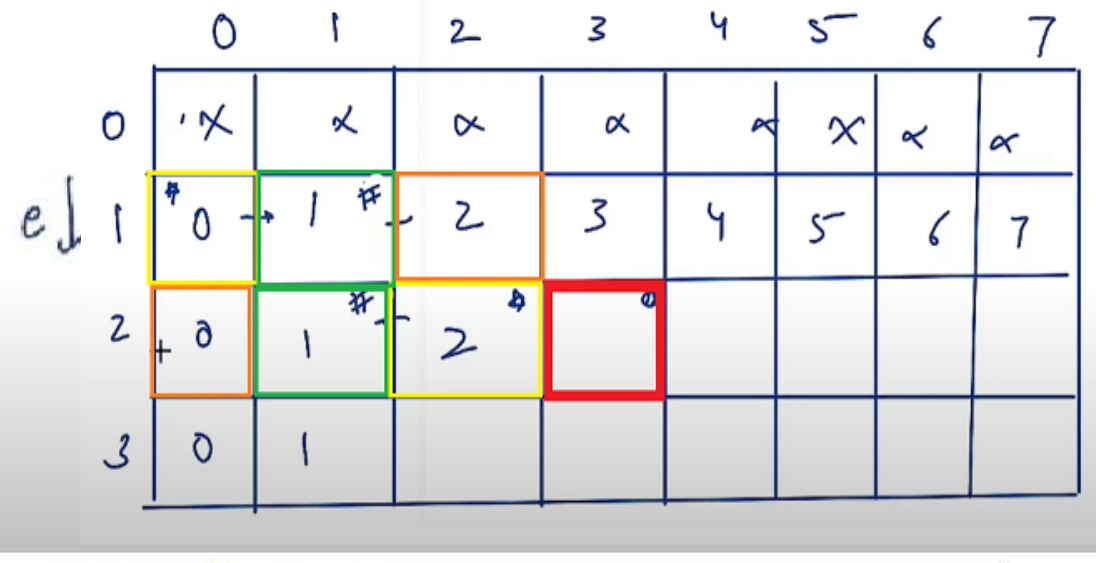
SOLUTION -> GUARENTEED BEST => Luck based question

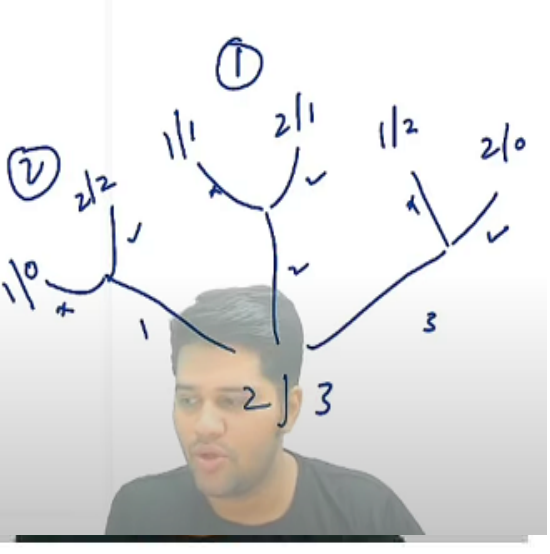


We can throw an egg from 1st, 2nd, 3rd, ..., fth floor
At each floor, the egg can break OR survive
If egg survuves -> we check further : e, k-1 (below k)
If egg breaks ->nwe check further: e-1, f-k (above k)



LUCK -> egg breaks or survives => Maximise the WORST
Solution = min (max(x1b, x1s), max(x2b, x2s), ...)

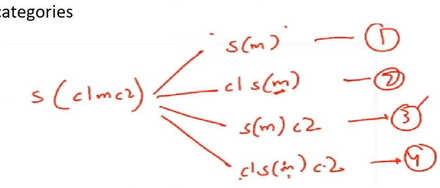


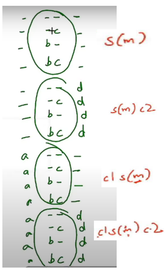


```
int solution (int e, int f){
    vector<vector<int>> dp(e+1, vector<int> (f+1, 0));
    for(int i = 1; i <= e; i++){
        for(int j = 1; j <= f; j++){
            if(i == 1)
                dp[i][j] = j;
            else if(j == 1)
                dp[i][j] = 1;
            else{
                int mn = j;
                for(int mj = j-1, pj = 0; mj >= 0; mj--, pj++){
                    int xb = dp[i-1][pj];
                    int xs = dp[i][mj];
                    int luck = max(xb, xs);
                    mn = min(mn, luck);
                }
                dp[i][j] = mn + 1;
            }
        }
    }
    return dp[e][f];
}
```

Subsequence Questions

Wenever you get question related to substring or subsequence, breaek the string into c1 mid c2 and make 4 categories

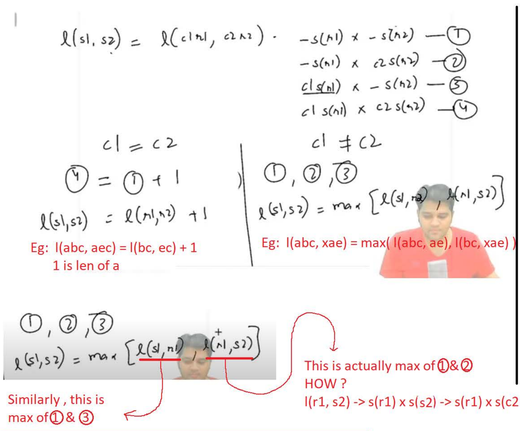




In all the problems, we have to find the formula to two cases
1. When c1 =c2
2. When c1 != c2

s -> set of subsequences

Longest Common Subsequence



Similarly , this is max of ① & ③

This is actually max of ① & ②
HOW ?
l(r1, s2) -> s(r1) x s(s2) -> s(r1) x s(c2 r2) ->
s(r1) x -s(r2) --->①
s(r1) x c2 s(r2) --->②

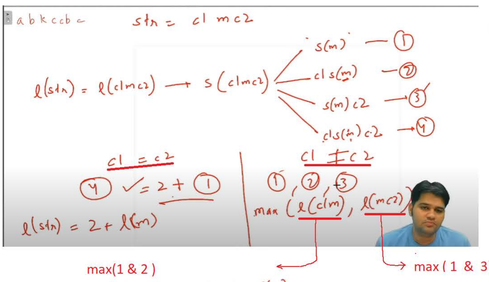
Therefore
max(l (s1, r2), max(r1, s2) = max(① , ② , ③)

FORMULA DERIVED

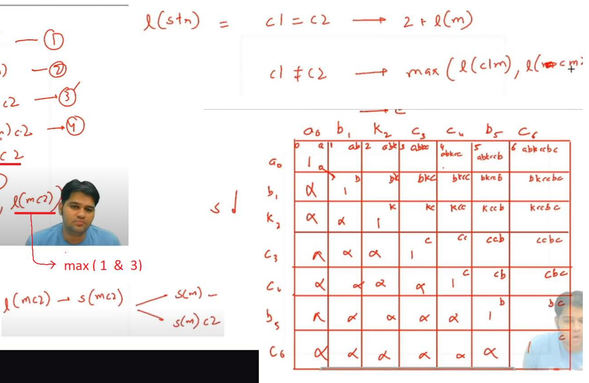
$$l(s1, s2) \rightarrow \begin{cases} 1 + l(n1, n2) & c1 = c2 \\ \max\left(l(n1, s2), l(s1, n2)\right) & c1 \neq c2 \end{cases}$$

```
int lcs(string text1, string text2){
    int n = text1.size();
    int m = text2.size();

    vector<vector<int>> dp(n+1, vector<int> (m+1, 0));
    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            if(text1[i-1] == text2[j-1]) // c1 == c2
                dp[i][j] = 1 + dp[i-1][j-1];
            else                        // c1 != c2
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[n][m];
}
```

Longest Palindromic Subsequence



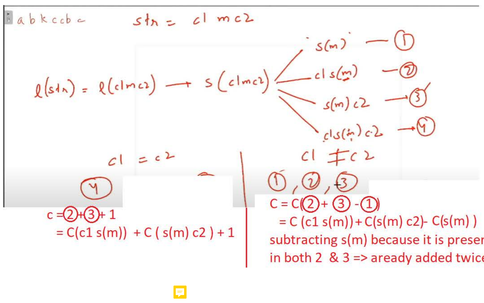
max(1 & 2)
max (1 & 3)




```
int lps(string str){
    int n = str.length();

    vector<vector<int>> dp(n, vector<int>(n, 0));
    for(int g = 0; g < n; g++){
        for(int i = 0, j = g; j < n; i++, j++){
            if(g == 0)
                dp[i][j] = 1;
            else{
                if(str[i] == str[j])
                    dp[i][j] = 2 + dp[i+1][j-1];
                else
                    dp[i][j] = max(dp[i][j-1], dp[i+1][j]);
            }
        }
    }
    return dp[0][n-1];
}

// Second approach is to use Longest common subsequence solution , passing second string as
reverse fo the string and finding the longest common subsequence will give longest palindromic
subseq
```

Alternative Approach -> LCS(s, reverse(s))

Count of Palindromic Subsequence




```
int count_of_lps(string str){
    int n = str.length();

    vector<vector<int>> dp(n, vector<int>(n, 0));
    for(int g = 0; g < n; g++){
        for(int i = 0, j = g; j < n; i++, j++){
            if(g == 0)
                dp[i][j] = 1;
            else{
                if(str[i] == str[j])
                    dp[i][j] = dp[i][j-1] + dp[i+1][j] + 1;
                else
                    dp[i][j] = dp[i][j-1] + dp[i+1][j] - dp[i+1][j-1];
            }
        }
    }
    return dp[0][n-1];
}
```

Longest Repeating Subsequence

Given a string, find the length of the longest repeating subsequence, such that the two subsequences don't have same string character at the same position.

for sting: a b a c b c
0 1 2 3 4 5

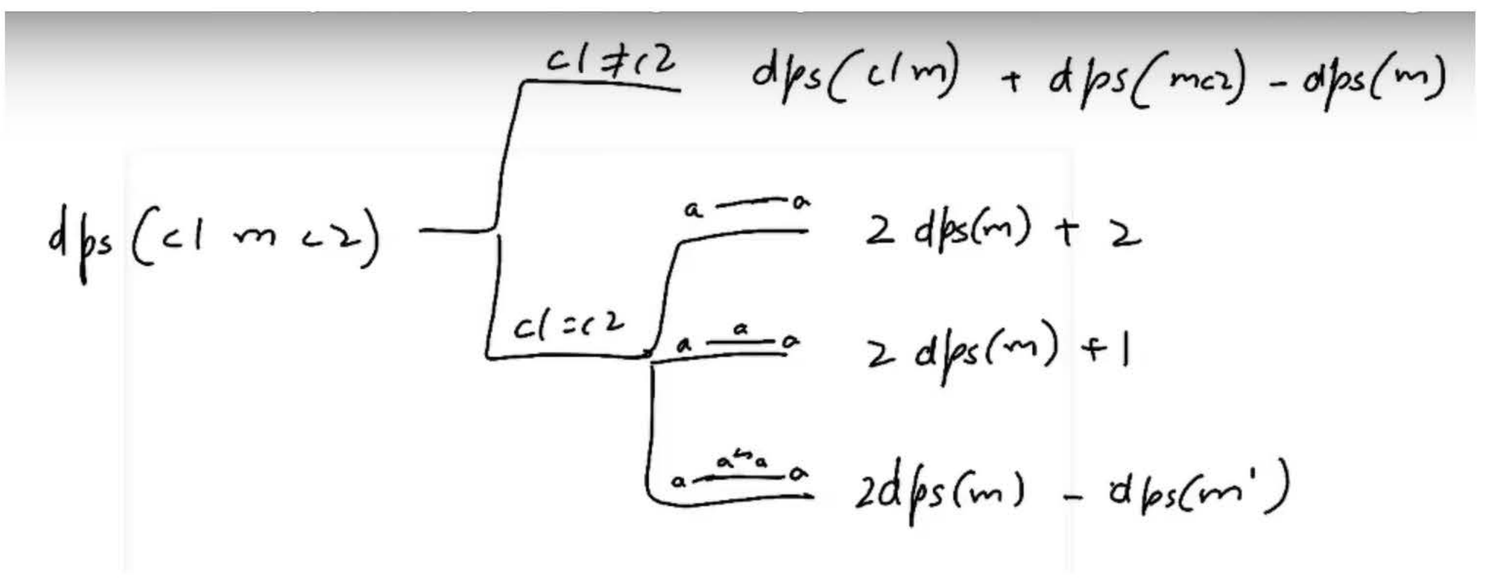
a b c a b c -> Valid
0 1 3 2 4 5

a b c a b c -> Inalid coz same a(index 0) appeared in both substrings
0 1 3 0 4 5

Solution -> LCS with itself but we need to alter c1 == c1 condition. While checking c1==c2, we have to make sure that index[c1] != index[c2] when c1 == c1

```
int lcs(string s1, string s2){
    int n = s1.size();
    int m = s2.size();
    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));

    for(int i = 1; i <= n; i++){
        for(int j = 1; j <= m; j++){
            if(s1[i-1] == s2[j-1] && i != j)
                dp[i][j] = dp[i-1][j-1] + 1;
            else
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
        }
    }
    return dp[n][m];
}

int lrs(string s){
    return lcs(s, s);
}
```

Count Distinct Palindromic Subsequences



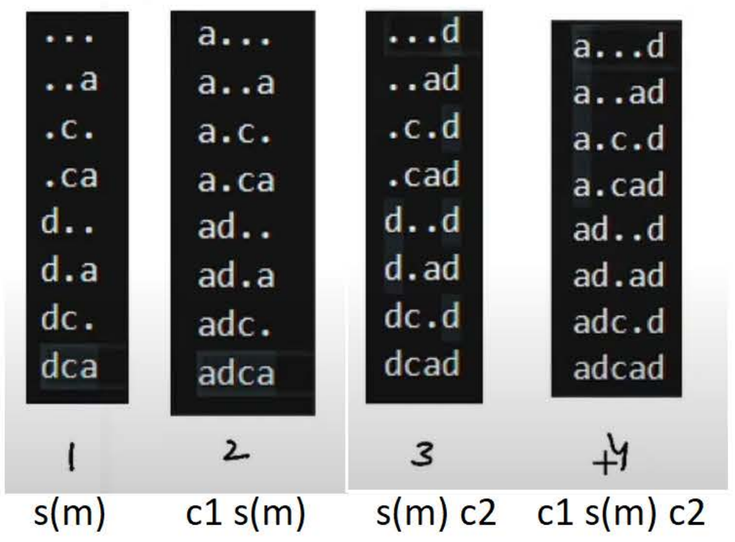
Pre Requisite

Given 4 sets S1 S2 S3 S4

Unique elements in

1. S1 & S2 = S1 + S2 - S1∩S2
2. S1, S2, S3 = S1+S2+S3- S1∩S2 - S2∩S3 - S1∩S3+ S1∩S2∩S3
3. S1 S2 S3 S4 = S1+S2+S3- S1∩S2- S2∩S3- S1∩S3+ S1∩S2∩S3 + S2∩S3∩S4 + S1∩S2∩S4 + S1∩S3∩S4 - S1∩S2∩S3∩S4

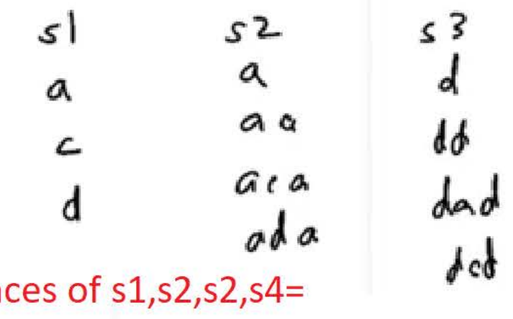
a d c a d



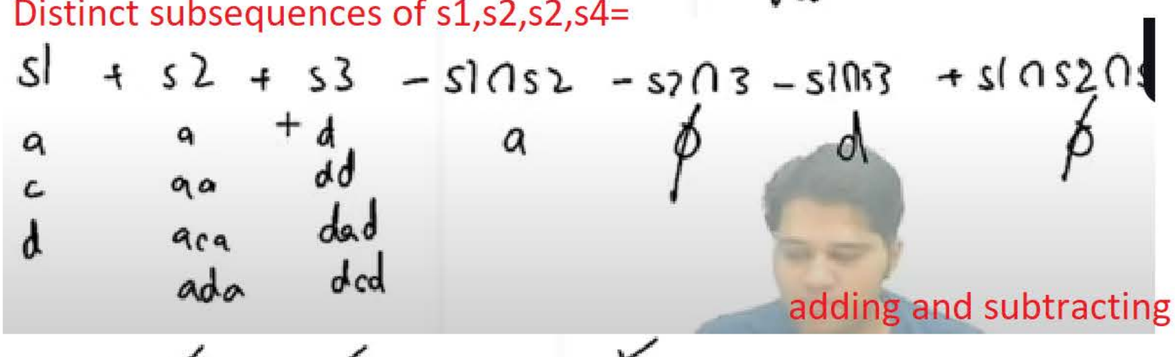
CASE 1 : C1 != C2

let s1 s2 s3 be the sets which contains distinct palindromic subsequence of m, c1m, mc2

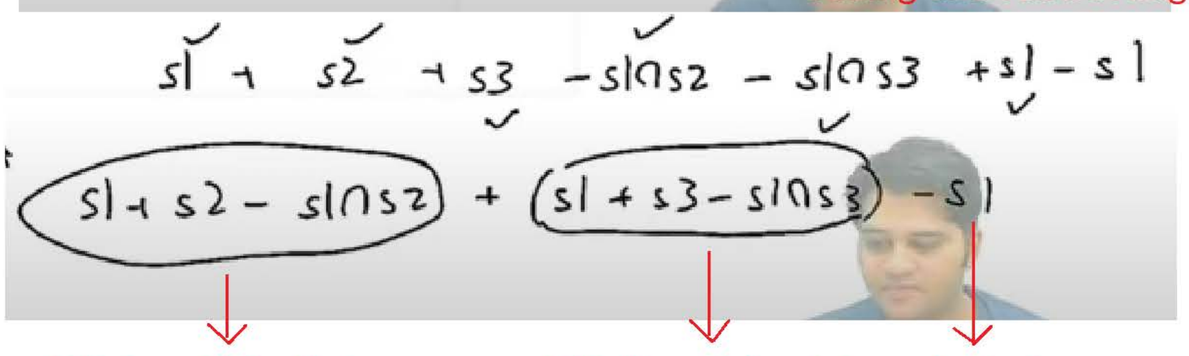
s4 is empty set



Distinct subsequences of s1,s2,s2,s4=



adding and subtracting s1



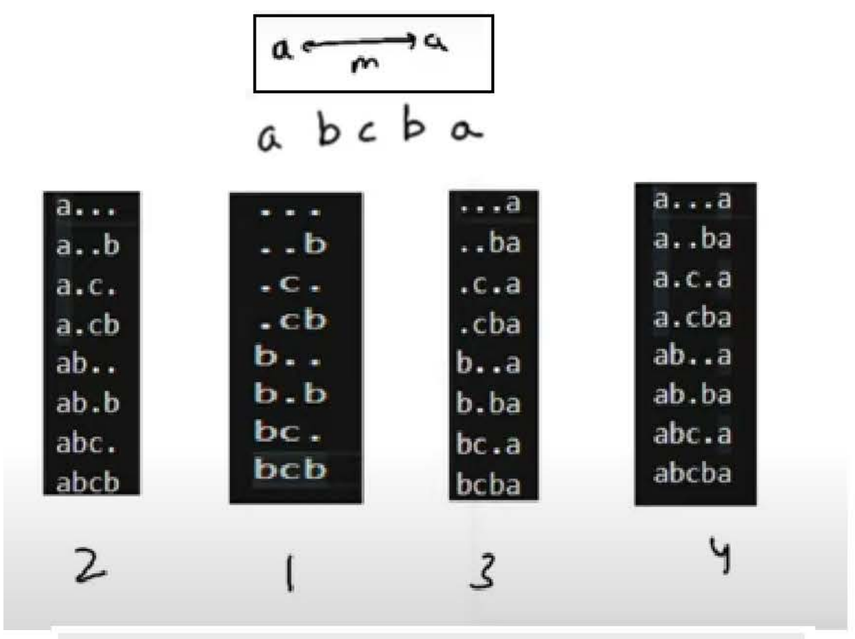
This is nothing but dps(c1 m)

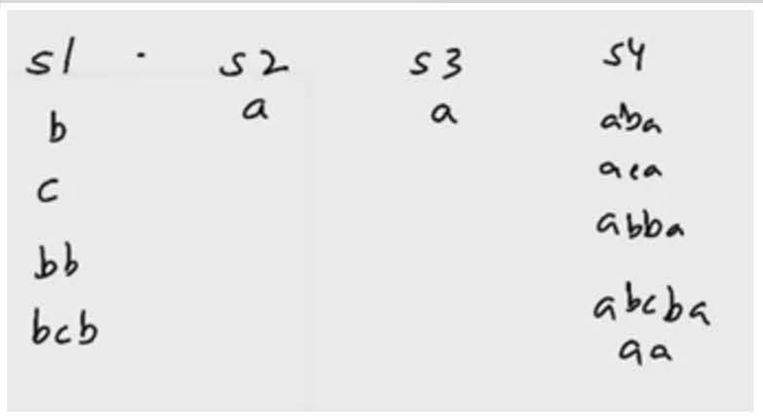
This is nothing but dps(mc2)

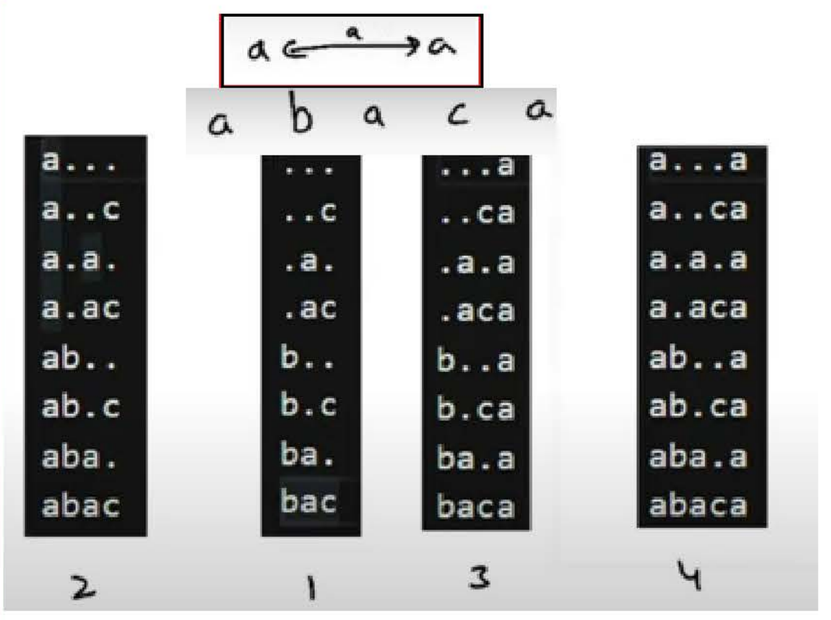
dps(m)

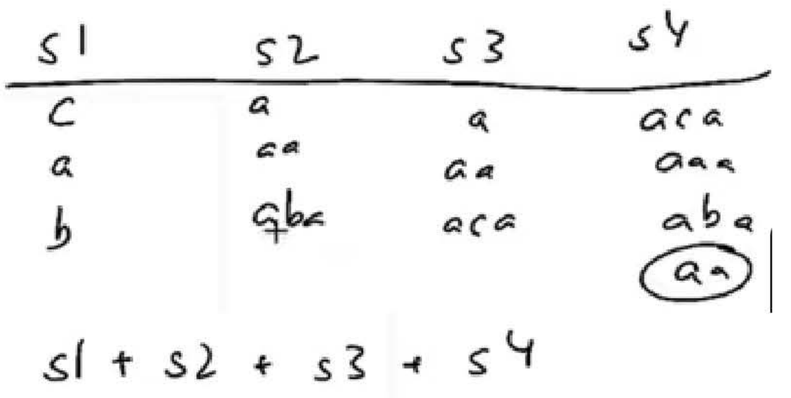
$$c1 \neq c2 \quad dps(c1m) + dps(mc2) - dps(m)$$

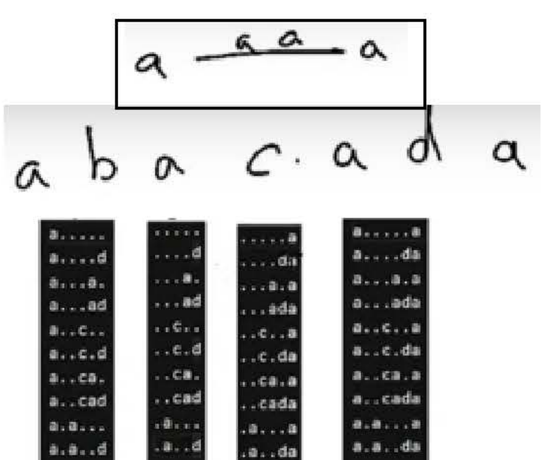
CASE 2 : C1 = C2

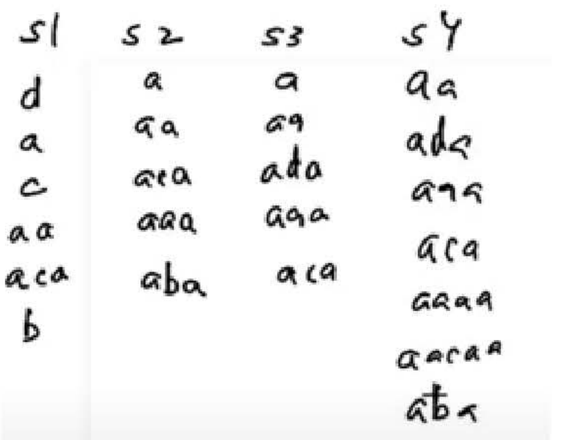










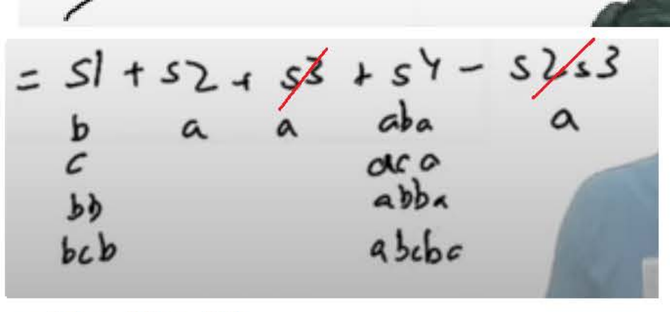


$s1 + s2 + s3 + s4$

$- s1s2 - s1s3 - s1s4 - s2s3 - s2s4 - s3s4$

$+ s1s2s3 + s1s2s4 + s1s3s4 + s2s3s4$

$- s1s2s3s4$ all cancelled elements -> null set



= S1 + S2 + S4

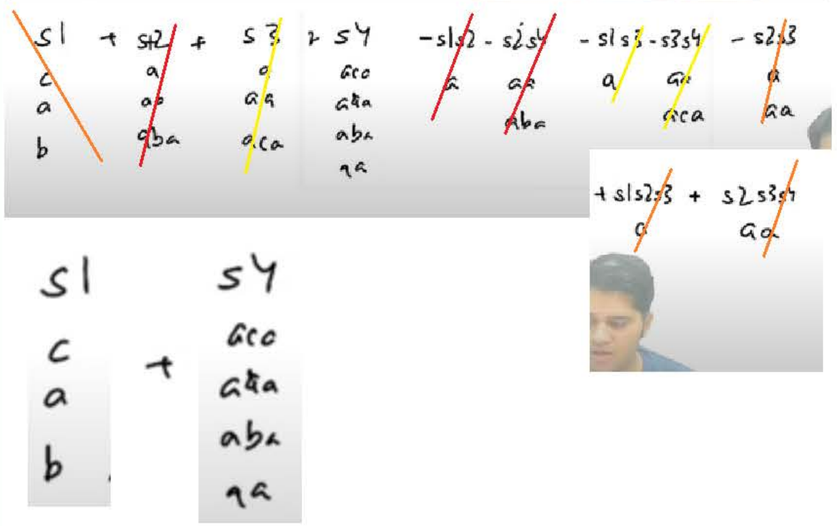
NO OF SUBSEQUENCES OF S4 = S1+1 COZ S4(C1 M C2) IS FORMED BY ADDING C1 AT START AND C2 AT END. EXTRA 1 IS C1C2

NO OF SUBSEQ IN S2 WILL ALWAYS BE EQUAL TO 1

THEREFORE THE EQN BECOMES

= 2S1 + 2

2*dsp(m) + 2



S4 = S1+1

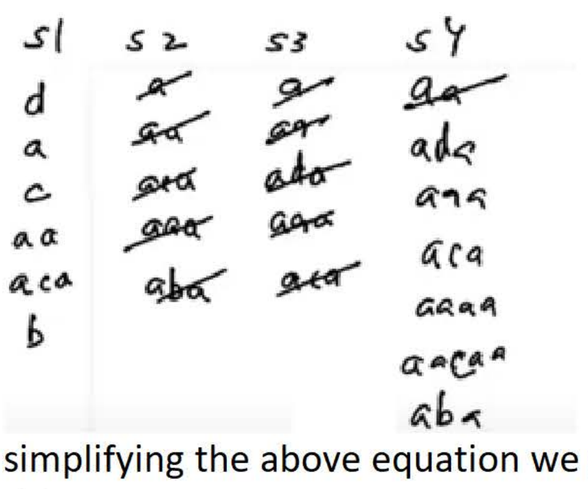
THE REASON IS DISCUSSED IN a <---> a

Therefore the equation becomes

= 2S1 + 1

2*dsp(m) + 1

S1 S2 S3 S4 = S1+S2+S3- S1∩S2- S2∩S3- S1∩S3+ S1∩S2∩S3 + S2∩S3∩S4 + S1∩S2∩S4 + S1∩S3∩S4 - S1∩S2∩S3∩S4



By simplifying the above equation we get S1+S4- aca

here S4=S1 because aa is already cancelled by some other elemnt and **aca** is nothing but palindromic subsequence of inner a - a

= 2S1 - aca

2*dps(m) - dps(m')

```
vector<vector<int>> dp(n, vector<int>(n, 0));
for(int g = 0; g < n; g++){
    for(int i = 0, j = g; j < n; i++, j++){
        if(g == 0)
            dp[i][j] = 1;
        else if(g == 1)
            dp[i][j] = 2;
        else{
            char c1 = str[i];
            char c2 = str[j];
            if(c1 != c2){ // dp[str] = dp[c1m] + dp[mc2] - dp[m]
                dp[i][j] = dp[i][j-1] + dp[i+1][j] - dp[i+1][j-1];
            }
            else{
                int n = next[i];
                int p = prev[j];
                int mid = dp[i+1][j-1];
                if(n > p)
                    // a <--- a ---> a  -> dp[str] = 2*dp[mid]+2
                    dp[i][j] = 2*mid + 2;
                else if(n == p)
                    // a <--- a ---> a  -> dp[str] = 2*dp[mid]+1
                    dp[i][j] = 2*mid + 1;
                else
                    //          m'
                    // a <---a---> a ---> a  -> dp[str] = 2*dp[mid] - dp[m']
                    dp[i][j] = 2*mid - dp[n+1][p-1];
            }
        }
    }
}
return dp[0][n-1];
```

Pattern matching

Given an input string (`s`) and a pattern (`p`), implement wildcard pattern matching with support for `'?'` and `'*'` where:

- `'?'` Matches any single character.
- `'*'` Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

```cpp
bool isMatch2(string s, string p){
    int n = p.size();
    int m = s.size();
    vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
    dp[n][m] = true; //empty pattern matches empty string

    for(int i = n-1; i >= 0; i--){
        for(int j = m; j >= 0; j--){
            if(j == m){
                if(p[i] == '*')
                    dp[i][j] = dp[i+1][j];
            }
            else{
                if(p[i] == '?')
                    dp[i][j] = dp[i+1][j+1];
                else if(p[i] == '*')
                    dp[i][j] = dp[i+1][j] || dp[i][j+1];
                else
                    dp[i][j] = dp[i+1][j+1] && p[i] == s[j];
            }

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

Given an input string `s` and a pattern `p`, implement regular expression matching with support for `'.'` and `'*'` where:

- `'.'` Matches any single character.
- `'*'` Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial).

```cpp
bool isMatch(string s, string p) {
    int n = p.size();
    int m = s.size();
    vector<vector<bool>> dp(n+1, vector<bool> (m+1, false));
    dp[n][m] = true;

    for(int i = n-1; i >= 0; i--){
        for(int j = m; j >= 0; j--){
            if(p[i+1] == '*'){
                dp[i][j] = dp[i+1][j];
            }
            else if(j == m){
                if(p[i] == '*')
                    dp[i][j] = dp[i+1][j];
            }
            else{
                if(p[i] == '.')
                    dp[i][j] = dp[i+1][j+1];
                else if(p[i] == '*'){
                    bool zeroApp = dp[i+1][j];                                          //zero appearance
                    bool oneApp = dp[i+1][j+1] && (p[i-1] == '.' ? true : p[i-1] == s[j]); //one appearance
                    bool twoOrMore = dp[i][j+1]&& (p[i-1] == '.' ? true : p[i-1] == s[j]); //two or more appearance
                    dp[i][j] = zeroApp || oneApp || twoOrMore;
                }
                else
                    dp[i][j] = dp[i+1][j+1] && p[i] == s[j];
            }
        }
    }
    return dp[0][0];
```

Regular exp matching

Scrambe String

We can scramble a string s to get a string t using the following algorithm:

If the length of the string is 1, stop.

If the length of the string is > 1, do the following:

- Split the string into two non-empty substrings at a random index, i.e., if the string is `s`, divide it to `x` and `y` where `s = x + y`.
- **Randomly** decide to swap the two substrings or to keep them in the same order. i.e., after this step, `s` may become `s = x + y` or `s = y + x`.
- Apply step 1 recursively on each of the two substrings `x` and `y`.

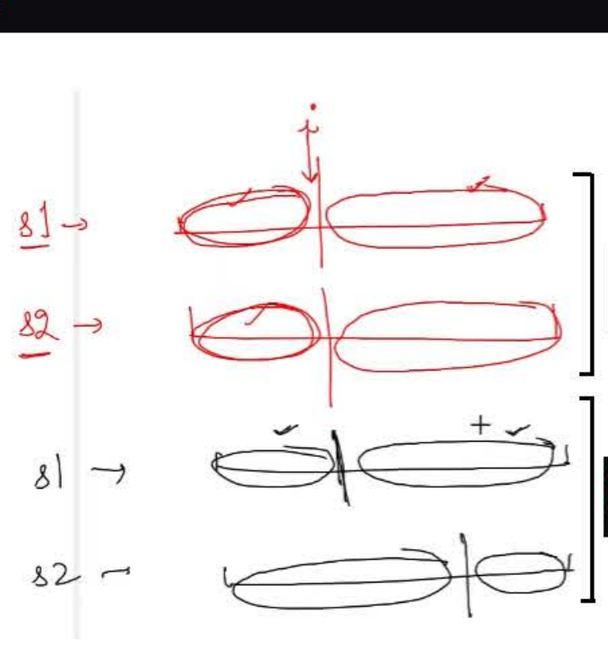
Given two strings `s1` and `s2` of **the same length**, return `true` if `s2` is a scrambled string of `s1`, otherwise, return `false`.

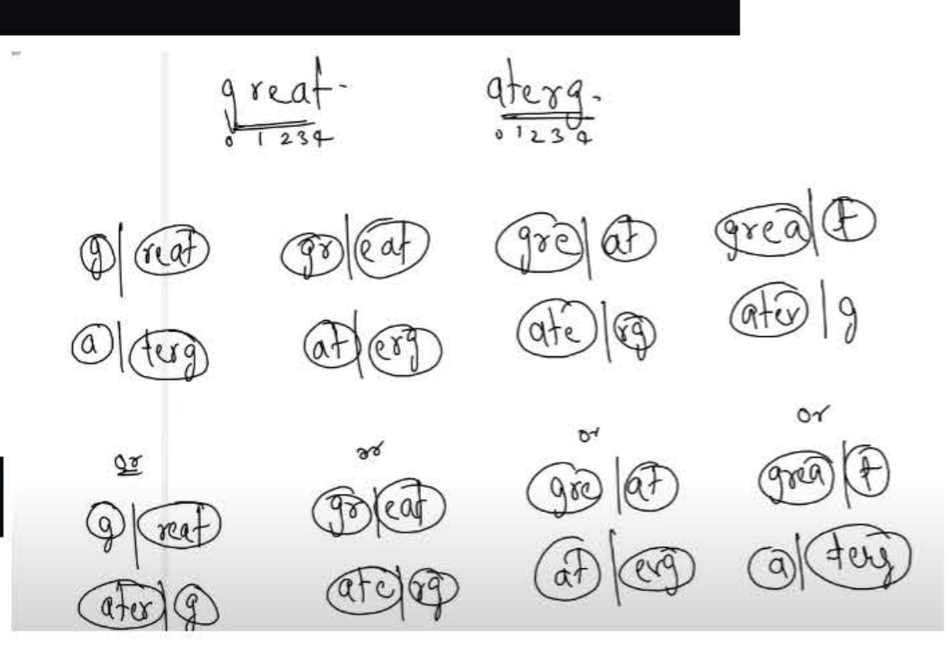
Recursive Approach

```cpp
unordered_set<string> s;
bool isScramble(string s1, string s2){
    if(s1 == s2) return true;
    if(s1.size() != s2.size()) return false;

    //Memorization
    if(s.find(s1+s2) != s.end()) return false;

    for(int i = 1; i < s1.size(); i++){ //We can split anywhere
        string x1 = s1.substr(0, i);
        string y1 = s1.substr(i);

        string x2 = s2.substr(i);
        string y2 = s2.substr(i);

        //No swap
        bool check1 = isScramble(s1.substr(0, i), s2.substr(i));
        bool check2 = isScramble(s2.substr(i), s2.substr(i));

        //Swap
        if(!(check1 && check2)){
            check1 = isScramble(s1.substr(0, i), s2.substr(s2.size()-i));
            check2 = isScramble(s1.substr(i), s2.substr(0, s2.size()-i));
        }

        if(check1 && check2) return true;
    }
    s.insert(s1+s2);
    return false;
}
```

avoiding use of substr function

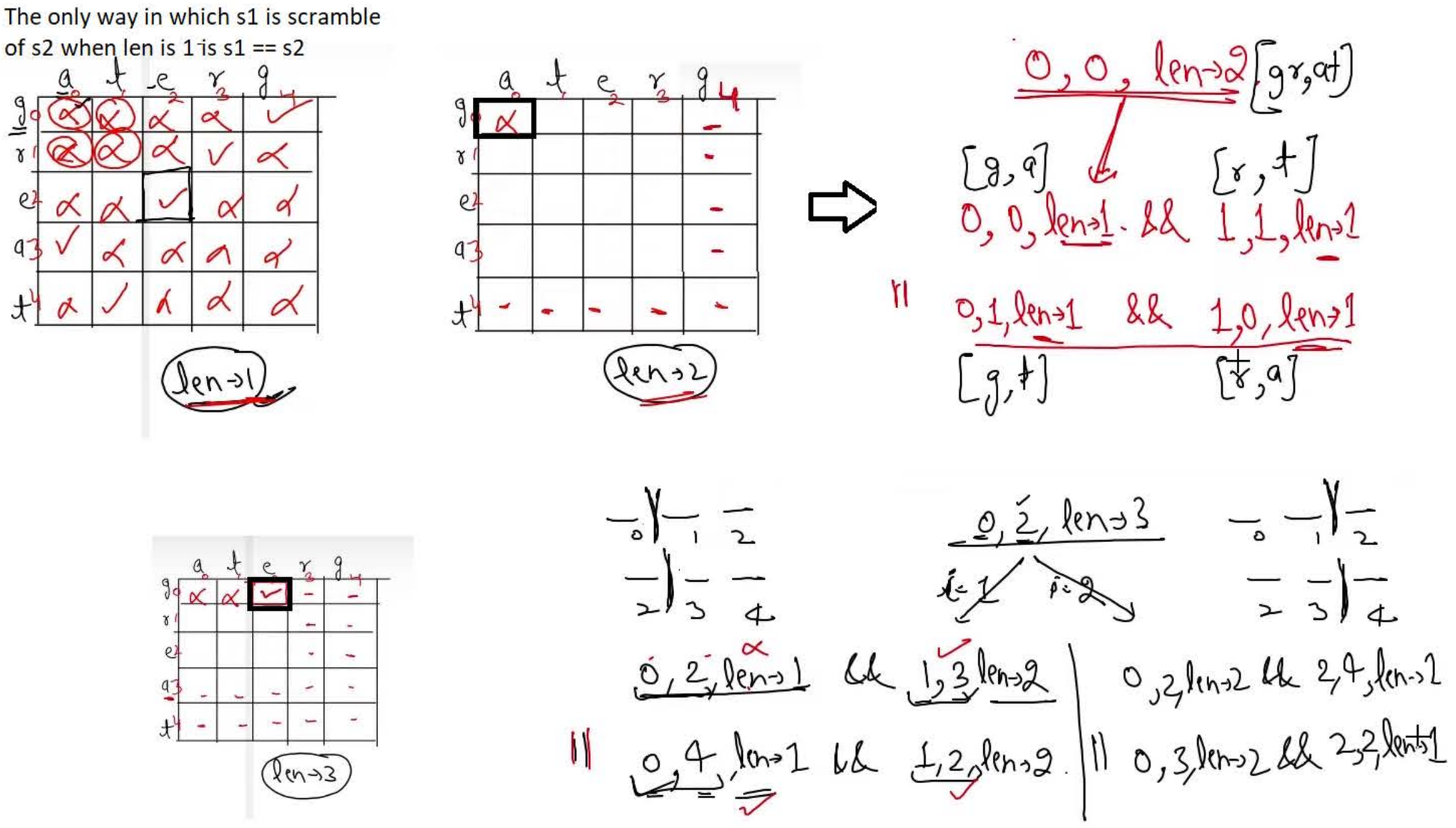
```cpp
unordered_map<string, bool> mp;
bool isScramble2(string word1, string word2, int si1, int si2, int len) {
    string s1 = word1.substr(si1, len);
    string s2 = word2.substr(si2, len);

    string key = s1 + s2;
    if (mp.find(key) != mp.end()) return mp[key];
    if (s1 == s2){
        mp[key] = true;
        return true;
    }

    for (int i = si1; i < len; i++) { // We can split anywhere
        // No swap
        bool check1 = isScramble2(word1, word2, si1, si2, i);
        bool check2 = isScramble2(word1, word2, si1 + i, si2 + i, len-i);
        // Swap
        if (!(check1 && check2)) {
            check1 = isScramble2(word1, word2, si1, si2+len-i, i);
            check2 = isScramble2(word1, word2, si1+i, si2, len-i);
        }
        if (check1 && check2) {
            mp[key] = true;
            return true;
        }
    }

    mp[key] = false;
    return false;
}
```

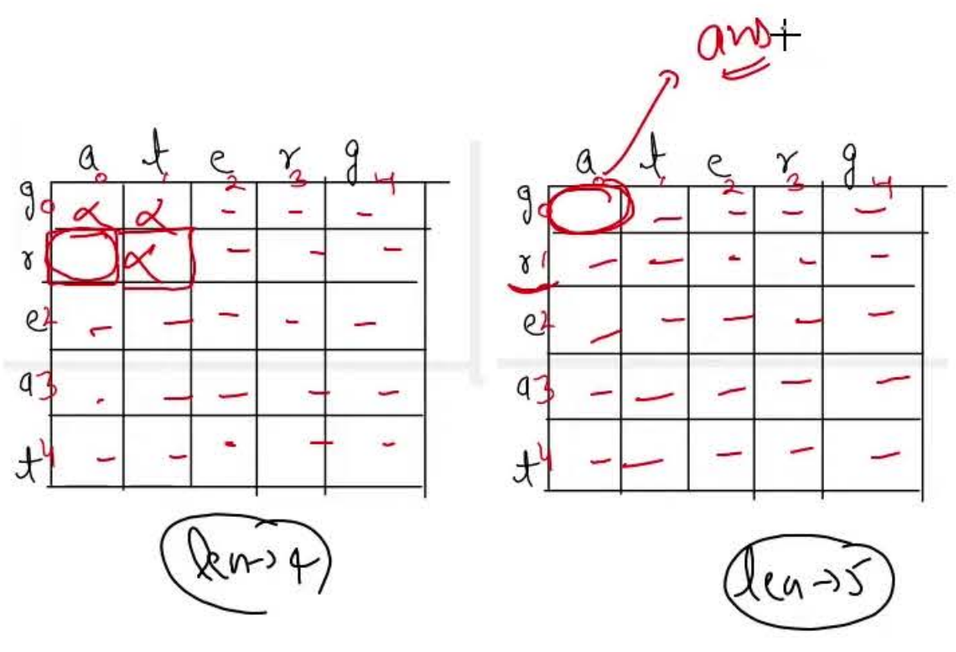




DP Approach

dp[len][i][j] -> is s1 of length 'len' starting from ith index scramble of s2 of length 'len' starting at jth index






```cpp
//dp[len][i][j] -> s1.substr(i, i+len) is samble of s2.substr(j, j+len)
bool isScramble(string s1, string s2) {
    int n = s1.size();
    vector<vector<vector<int>>> dp(n + 1, vector<vector<int>>(n, vector<int>(n)));
    for (int i = 0; i < n; i++) { // len 1
        for (int j = 0; j < n; j++) {

        }
    }
    for (int len = 1; len <= n; len++) {
        for (int i = 0; i <= n-len; i++) {
            for (int j = 0; j <= n-len; j++) {
                if(len == 1){
                    dp[len][i][j] = s1[i] == s2[j];
                    continue;
                }
                for(int k = 1; k < len && !dp[len][i][j]; k++){
                    //Indexing is same as in recursion  i->si1  j->si2  k->len
                    dp[len][i][j] = (dp[k][i][j] && dp[len-k][i+k][j+k]) || (dp[k][i][j+len-k] && dp[len-k][i+k][j]);
                }

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

Max Sum Subarray Variation

Kadanes Algorithm

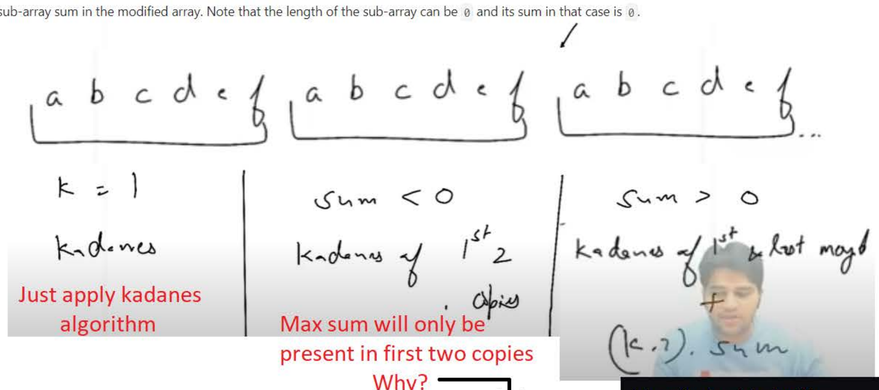
```cpp
int kadanes(vector<int> &arr){
    int currSum = arr[0];
    int mxSum = arr[0];
    for(int i = 1; i < arr.size(); i++){
        if(currSum >= 0)
            currSum += arr[i];
        else
            currSum = arr[i];
        mxSum = max(mxSum, currSum);
    }
    return mxSum;
}
```

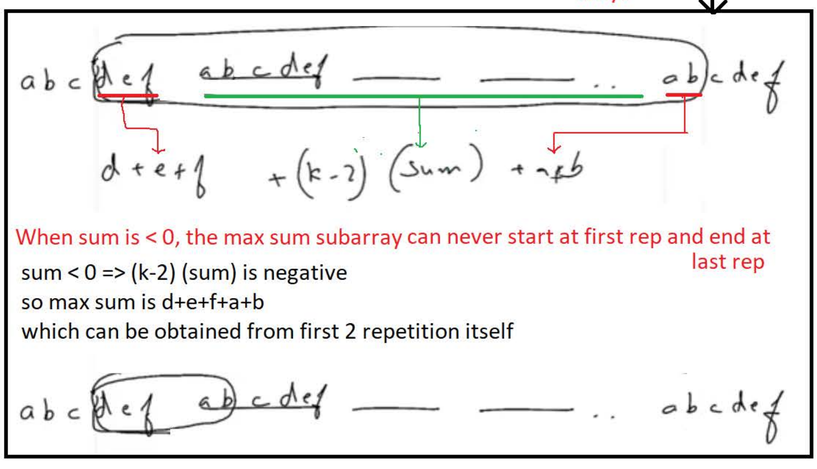
Each element has an option to start a new subarray and to join with prev subarray

By observation we find that an element will join the prev subarray if it has a positive sum
Else it will start a new subarray

K-Concatenation Maximum Sum

Given an integer array `arr` and an integer `k`, modify the array by repeating it `k` times.
For example, if `arr = [1, 2]` and `k = 3` then the modified array will be `[1, 2, 1, 2, 1, 2]`.
Return the maximum sub-array sum in the modified array. Note that the length of the sub-array can be `0` and its sum in that case is `0`.





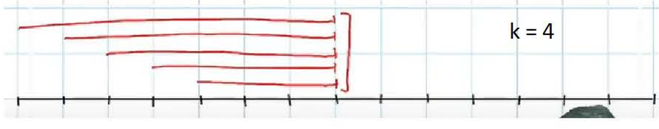
When sum is < 0, the max sum subarray can never start at first rep and end at last rep

sum < 0 => (k-2) (sum) is negative
so max sum is d+e+f+a+b
which can be obtained from first 2 repetition itself


```cpp
int sum(vector<int> &arr){
    int s = 0;
    for(int i : arr) s += i;
    return s;
}

int kadanes(vector<int> &arr){
    int currSum = arr[0];
    int maxSum = arr[0];
    for(int i = 1; i < arr.size(); i++){
        if(currSum >= 0)
            currSum += arr[i];
        else
            currSum = arr[i];
        maxSum = max(maxSum, currSum);
    }
    return maxSum;
}

int kConcatenationMaxSum(vector<int> arr, int k) {
    int s = sum(arr);
    if(k == 1)
        return kadanes(arr);
    else{
        arr.insert(arr.end(), arr.begin(), arr.end()); //Creating 2 c
        if(s >= 0)
            return kadanes(arr) + (k-2)*s;
        else
            return kadanes(arr);
    }
}
```

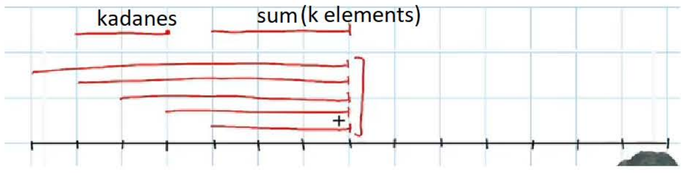
Maximum Sum Subarray with at least Size K

Given an array, find the subarray (containing at least k numbers) which has the largest sum.



For every subarray ending at index **i** , the starting point can be anywhere from -> i-k.
So maximum of sum of all those subarrays will be the answer

Instead an optimized approach will be to get the max sum subarray tilll index **i-k** (irrespective of length) ie **kadanes[i-k]** and add sum of i-kth element to ith element (k elements) ie **sum(arr[i-k] -> arr[i])**
This will give the answer coz kananes(i-k) will give the max sum subarray till i-kth index and by adding sum(arr[i-k] -> arr[i]), the length constraint is managed




```cpp
int kadanes(vector<int> &arr, vector<int> &dp){
    int currSum = 0;
    int mxSum = 0;
    dp[0] = 0;
    for(int i = 0; i < arr.size(); i++){
        if(currSum >= 0)
            currSum += arr[i];
        else
            currSum = arr[i];
        mxSum = max(mxSum, currSum);
        dp[i] = mxSum;
    }
    return mxSum;
}
```

```cpp
vector<int> Ksum(vector<int> &arr, int k){
    vector<int> sum(arr.size());
    int s = 0;
    for(int i = 0; i < k-1; i++){
        s += arr[i];
    }

    int si = 0;
    for(int i = k-1; i < arr.size(); i++, si++){
        s += arr[i];
        sum[i] = s;
        s -= arr[si];
    }
    return sum;
}
```

```cpp
int ans(vector<int> &arr, int k){
    vector<int> dp(arr.size());
    kadanes(arr, dp);

    vector<int> ksum = Ksum(arr, k); //Sum of arr[i-k] -> arr[i]
    int maxKsum = ksum[k-1];
    for(int i = k; i < arr.size(); i++){
        int currSum = dp[i-k] + ksum[i];
        maxKsum = max(maxKsum, currSum);
    }
    return maxKsum;
}
```

Cherry Pickup

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

- `0` means the cell is empty, so you can pass through,
- `1` means the cell contains a cherry that you can pick up and pass through, or
- `-1` means the cell contains a thorn that blocks your way.

Return *the maximum number of cherries you can collect by following the rules below*:

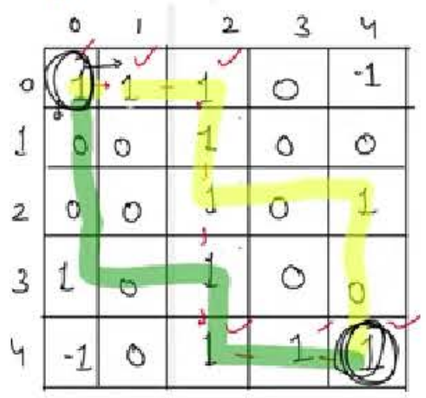
- Starting at the position `(0, 0)` and reaching `(n - 1, n - 1)` by moving right or down through valid path cells (cells with value `0` or `1`).
- After reaching `(n - 1, n - 1)`, returning to `(0, 0)` by moving left or up through valid path cells.
- When passing through a path cell containing a cherry, you pick it up, and the cell becomes an empty cell `0`.
- If there is no valid path between `(0, 0)` and `(n - 1, n - 1)`, then no cherries can be collected.

Recursive Approach

Start from (0,0) and explore all paths to (n-1, n-1)
For every path, explore all paths from (n-1, n-1) to (0,0)
Upon Reaching 0, update max path

Recursion Optimised

Instead of first going down and then coming back, We can go down as two different persons. One path can be considered as godown and another as goUp.



Ek bandhe ki jana aur vapas aana
is equivalent to do bandhe ka jana

Memorization

Since there are 4 variables varying (r1, c1, r2, c2)
Dimension = 4d

Also we can observe that always r1+c1 = r2+c2

=> If we know r1, r2 & c1 we can find c2

c2 = r1+c1-r2

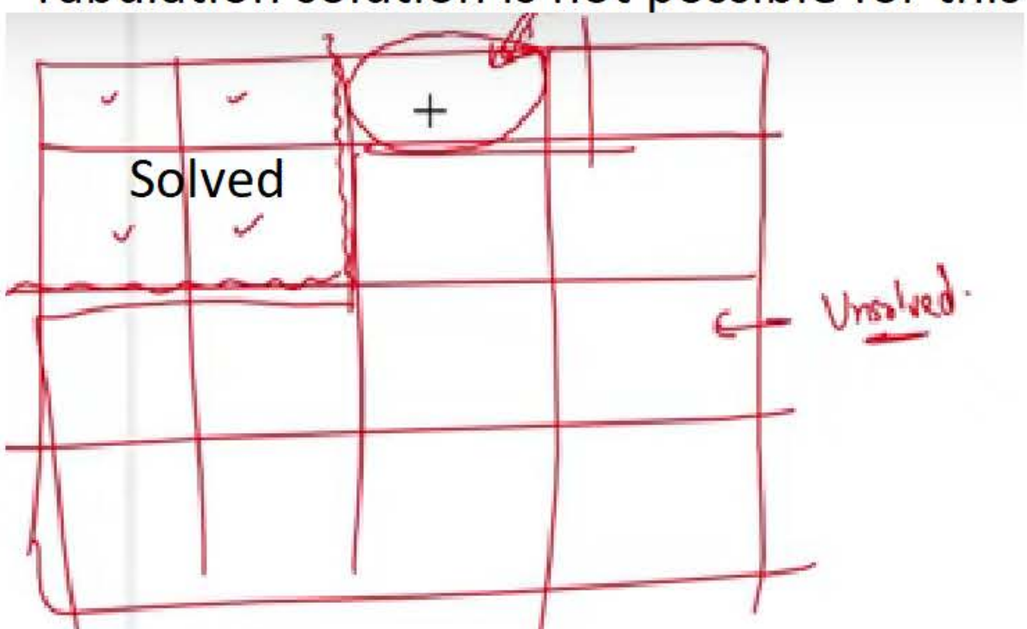
New Dimension = 3D

Recursive Approach

```cpp
int maxCherry = 0;
void goUp(vector<vector<int>> &grid, int i, int j, int cherry){
    if(i < 0 || j < 0 || grid[i][j] == -1)
        return;

    if(i == 0 && j == 0){
        maxCherry = max(maxCherry, cherry);
        return;
    }

    int cherries = grid[i][j];
    grid[i][j] = 0;
    goUp(grid, i-1, j, cherry + cherries); //Up
    goUp(grid, i, j-1, cherry + cherries); //Left
    grid[i][j] = cherries;
}

void goDown(vector<vector<int>> &grid, int i, int j, int cherry){
    if(i >= grid.size() || j >= grid.size() || grid[i][j] == -1)
        return;

    if(i == grid.size()-1 && j == grid.size()-1){
        goUp(grid, i, j, cherry);
        return;
    }

    int cherries = grid[i][j];
    grid[i][j] = 0;
    goDown(grid, i+1, j, cherry + cherries); //Down
    goDown(grid, i, j+1, cherry + cherries); //Right
    grid[i][j] = cherries;
}
int cherryPickup(vector<vector<int>>& grid) {
    goDown(grid, 0, 0, 0);
    return maxCherry;
```

```cpp
//Recursion optimisec approach
int goDown2(vector<vector<int>> &grid, int r1, int c1, int r2, int &n,vector<vector<vector<int>>> &dp){
    int c2 = r1+c1-r2;
    if(r1 >= n || r2 >= n || c1 >= n || c2 >= n || grid[r1][c1] == -1 || grid[r2][c2] == -1) return INT_MIN;

    //If both p1 and p2 reach destination
    if(r1 == n-1 && c1 == n-1)
        return grid[r1][c1];
    if(dp[r1][c1][r2] != -1)
        return dp[r1][c1][r2];

    int cherries = 0;
    if(r1 == r2 && c1 == c2) //If both are on same cell => The cherry can be picked only once
        cherries += grid[r1][c1];
    else
        cherries += grid[r1][c1] + grid[r2][c2];

    int m1 = goDown2(grid, r1, c1+1, r2, n, dp); //1-> H & 2->H
    int m2 = goDown2(grid, r1+1, c1, r2+1, n, dp); //1-> V & 2->V
    int m3 = goDown2(grid, r1, c1+1, r2+1, n, dp); //1 -> H & 2-> V
    int m4 = goDown2(grid, r1+1, c1, r2, n, dp); //1 -> V & 2-> H

    cherries += max(max(m1, m2), max(m3, m4));
    return dp[r1][c1][r2] = cherries;
}

int cherryPickup2(vector<vector<int>>& grid) {
    int n = grid.size();
    vector<vector<vector<int>>> dp(n, vector<vector<int>> (n, vector<int>(n, -1)));
    return max(0, goDown2(grid, 0, 0, 0, n, dp));
}
```

Tabulation solution is not possible for this solution



This is because, in every dp solution, the current solution dependes on previous solved part. But here that is not the case.