Dynamic Programming

~ by Striver Bhaiya.

Lecture - 1 (17/01/22)

① Fibonacci No

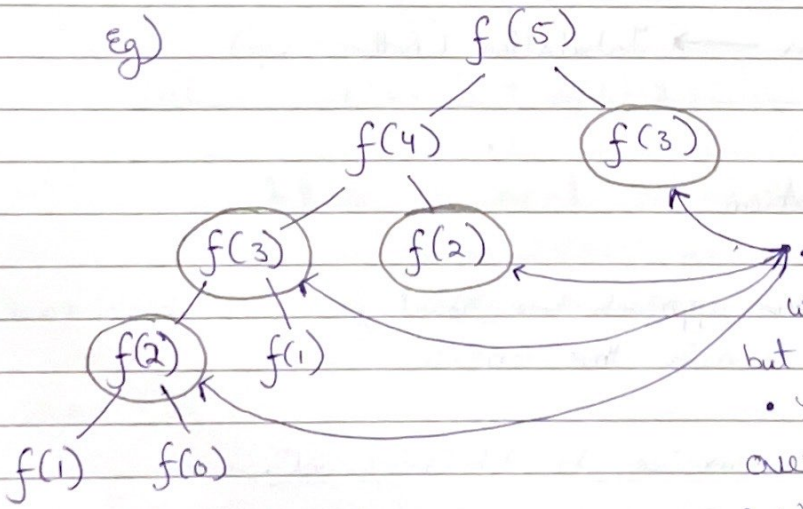
0 1 1 2 3 5 8 13 21 ....

$$f(n) = f(n-1) + f(n-2)$$

Part - 1 :- Memoization

Overlapping Subproblems.

Eg)



- These problems were computed once but these getting repeat
- These are called overlapping Subproblems.
- We can save these recurrence and that process is called memoization.

Code

```
int f(int n, vector<int>& dp)
{
    if(n<=1) return n;
    if(dp[n]!=-1) return dp[n];
    return dp[n]=f(n-1,dp)+f(n-2,dp);
}
```

Base case
Memorization
recursive call

This method of Dynamic Programming is called Memoization or Top-Down Approach. (It is an optimization of Recursion).

TC ⟶ O(n)
SC ⟶ O(n) + O(n)
↳ array.

Now, Recursion ⟶ Tabulation. (Bottom-Up)

Part-2: Tabulation

Main logic: In this approach we start from the base case and reach the answer **bottom to top approach**

Steps to convert recursive to tabulation sol^n.

1. Declare dp[] array of size n+1.
2. Initialize the base case values, i.e i=0 & i=1 of the dp[] as 0 & 1 resp.
3. Set a loop which iterates from 2 to n and for every index set values as dp[i-1] + dp[i-2];

~~ed~~ code

```
vector<int> dp(n+1, -1);

dp[0] = 0 , dp[1] = 1;

for( int i = 2 ; i<=n ; i++)
    dp[i] = dp[i-1] + dp[i-2] ;

cout << dp[n] ;
```

TC → O(n)
SC → O(n)

Part 3: Space Optimization.

If we carefully ~~look~~ look at the relation

```
dp[i] = dp[i-1] + dp[i-2];
```

there is actually no need to maintain an array because notice. for every element all we need is it's last two elements. prev & prev2

Algorithm

- Calculate curr_i = prev + prev2;
- where prev = 1, prev2 = 0;
- After calculation: prev2 = prev;
  prev = curr_i;

for every iteration from 2 to n and print 'prev' as answer.

Code

```
int prev2 = 0;
int prev = 1.

for (int i=2; i<=n; i++)
{
    int curr_i = prev + prev2;
    prev2 = prev;
    prev = curr_i;
}
cout << prev;
```

TC ⟶ O(n)
SC ⟶ O(1).

Lecture - 2 : Climbing Stairs (18th Jan 2022)

① How to identify A problem is a DP prob?

Generally (but not limited to) if the problem statement asks :-

a) Count the total number of ways
b) Given multiple ways of doing a task, which will give it the min or max output.

In such problem we can apply recursion, then memoization for DP.
General trend.

Recursion ⟶ Memoization ⟶ Tabular form ⟶ Space Optimization

2) General Steps to Solve Problem After Identification

a) Try to represent the problem in terms of indexes.
b) Try all possible choices/ways at every index according to the problem statement.
c) If the ques^ states :

-) Count all ways → return sum of all ways.
-) find max/min → return the choice with max/min way.

for the Ques^ (climbing Stairs)

The question is very similar to fibonacci series so the recurrence is

```
if (ind==0) return 1;
if (ind==1) return 1;

return f(ind-1) + f(ind-2);
```

The memoization, Tabulation & Space Opt. is the same as fibonacci series.

Please Turn Over.

Memoisation

- Create dp[n+1] = -1.
- Our main goal is to memorize the overlapping subproblems. In this, ~~if~~ we wants to store the best soln for a particular value of n.
- Simply, we check if(dp[ind] != -1)
  yes → return dp[ind]
  otherwise, continue and store the result i.e
  dp[ind] = min(l, r);

TC → O(n)
SC → O(n).

Space - Optimization

If we notice carefully, the values required for every iterations are
dp[i], dp[i-1] & dp[i-2] like last problem we can handle this.

We Declare 2 variables prev, prev2 = 0; Calc JumpOne and JumpTwo and put their min in curr and update prev = prev2
prev2 = curr and continue.

PTO

Code

```
int f(int n, vector<int>& heights)
{
    int prev, prev2 = 0;

    for(int i=1; i<n; i++)
    {
        int l = prev + abs(heights[i]-heights[i-1]);
        int r = INT_MAX;
        if(i>1)
            r = prev2 + abs(heights[i] - heights[i-2]);

        int curr = min(l, r);
        prev2 = prev;
        prev = curr;
    }
    return prev;
}
```

TC → O(n).
SC → O(1).

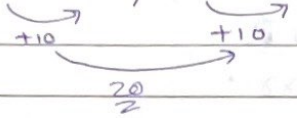
Lecture: 3 – Frog Jump (19th Jan 2022)

Problem Statement

Given no. of stairs and frog, frog has to jump from 0^{th} to $(n-1)^{th}$ stair. At the time he can only jump one (or) two steps. Every jump consumes an energy i.e abs(heights[i] - heights[j]) where heights is height of stairs.

Input N = 4
(10, 20, 30, 10)

Output 20 { 10 → 20, 20 → 10 }



Solution) The first approach which comes to mind is greedy. That is to compute min. jump but for the example below. Greedy fails.

[30, 10, 60, 10, 60, 50]

According Greedy ⇒ 30 → 10 → 10 → 50 = 60

But Non-Greedy ⇒ 30 → 60 → 60 → 50 = 40
optimal Sol^n

Steps for Recursive

Step-1) Express the problem in terms of indexes

(a) This can be easily done as their arr ind [0,1,2...n-1]
(b) We can say that f(n-1) signifies min. energy required to move from 0 to (n-1) stair.
(c) f(0) simply should give us the answer as '0' (base case)

Step-2) Trying all choices to reach the goal.

The frog can jump either by one step or by two step. We will calculate the cost of the jump from the height array. Note: The rest of the cost will be ~~jumped~~ returned by the recursive calls that we make.

Step-3) Take the minimum of all choices.

As the ques^n asks for, we return the min. of Step 2.

Note At ind == 1, we ~~fo~~ can not call f(n-2) we can only compute f(n-1).

Pseudo - Code

```
int f(ind, height[])
{
    if(ind == 0) return 0;

    int JumpOne = f(ind-1) + abs(height[ind] - height[ind-1]);

    if(ind > 1)
        int JumpTwo = f(n-2) + abs(height[ind] - height[ind-2]);

    return min(JumpOne, JumpTwo);
}
```

Tabulation

- Declare dp[] array.
- Set dp[0] = 0,
- Iterate from 1 to (n-1) and calc JumpOne and JumpTwo and set dp[i] = min(JumpOne, JumpTwo).

Pseudo-Code

```
int f(int ind, vector<int> &heights)
{
    vector<int> dp(n+1, -1);
    dp[0] = 0;

    for(int i=1; i<n; i++)
    {
        int l = dp[i-1] + abs(height[i] - height[i-1]);
        int r = INT_MAX;
        if(i>1)
            r = dp[i-2] + abs(height[i] - height[i-2]);

        dp[i] = min(l, r);
    }
    return dp[n-1];
}.
```

For Memoisation and Space optimazation refer pg 6

Lecture 4- Frog Jump with K Distance. (22nd Jan 2022)

Exactly like the last question, this time the frog can skip upto 'k' stones. We have to find minimum energy lost.

Eg) 10 4
40 10 20 70 80 10 20 70 80 60

Ans) 40
1 → 4 → 8 → 10

The code is exactly the same we just need to generalize (ind-1), (ind-2) to (ind-K) by using a loop from i=1 to K. We compute all the possible scenario.

Recursive Code with Memoization.

```
int f(int ind, vector<int>& arr, vector<int>& dp) {
    if (ind == 0) return 0;
    if (dp[ind] != -1) return dp[ind];
    int min_ans = INT_MAX;
    for(int i=1; i<=k; i++) {
        if(ind-i >= 0)
            int Jump = f(ind-k, arr, dp) + abs(arr[ind] -
                                               arr[ind-i]);
        min_ans = min(min_ans, Jump);
    }
    return min_ans;
}
```

Lecture 5: Maximum Sum of Non-Adjacent Elements
AKA House Robber (on leetcode)

Problem Statement

Given an array of 'N' positive int, return maximum subsequence such that no 2 adjacent elements are allowed.

Eg)

N = 3 , Arr = 1, 2, 4
Output = 1+4 = 5

N = 4 , Arr = 2, 1, 4, 9
Output = 11 (2+9)

Solution

Logic: As we need to find sum of subsequences, an approach that comes to mind is to generate all subs and pick the one with max. sum.

To Generate subsequence, we can use the pick/not pick technique. i.e.

1. At every index of the array we have two choices.
2. First, to pick the array element at that index and consider it in subs.
3. Second, to leave the array element at that index and not consider in subs.

Recursive And Memoization Sol"

Step 1) form the function in terms of indexes

- We can define our function f(ind) as: Max Sum of Subs from 0 to (ind).
- We need to return f(n-1) as our final answer.

Step 2) Try all the choices to reach the goal.

- Using the pick/not pick technique...
  if we pick an element then, pick = arr[ind] + f(ind-2)
  We are choosing (ind-2) because we can not choose adjacent elements.
- Next we need to ignore the current element in our subs. Non Pick = 0 + f(ind-1). As we don't pick the current element, and consider it's adjacent element.

Step 3) Take the max of all choices.

- Calculate the max between Step 2 options.

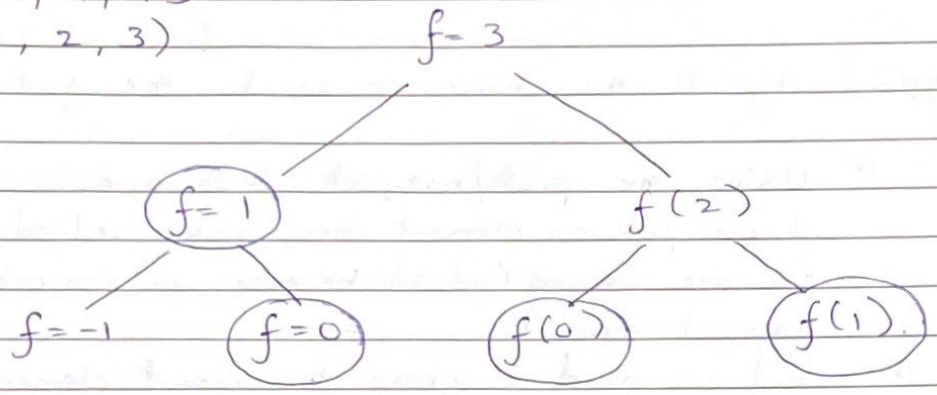
Base Case : ① If ind = 0, if we are at ind=0 this mean we had ignored ind=1 and we want to pick it Therefore we return arr[0].

② if ind < 0, this case can hit when we call f(ind-2) and we were at ind = 1; Thus we are trying to call f(-1) which is not possible and hence we return 0

Memoization.

Observe the tree below, there are overlapping sub-prob. Hence we can use memoization.

[2, 1, 4, 9]
(0, 1, 2, 3)



Simple, ① Creat DP[] set all values to -1
② before returning max, store it in dp[]
dp[ind] = max(pick, not pick).

Code

```
int f(int ind, vector<int> &arr, vector<int> &dp){
    if(ind==0) return arr[ind];
    if(ind<0) return 0;
    if(dp[ind]!=-1) return dp[ind];

    pick = arr[ind] + f(ind-2, arr, dp);

    notpick = f(ind-1, arr, dp);

    return dp[ind] = max(pick, notpick);
}
```

TC → O(N)

SC → O(N) + O(N)

Stack Space Array Space

Tabulation

1. Declare a dp[] array size (n)
2. base condition, dp[0] = arr[0]
3. Set an iterative loop which traverse the array (from 1 to n-1) and for every index calculated pick and notpick.
4. Set dp[i] = max(pick, not pick)

```
int f(int n, vector<int> &arr, vector<int> &dp)
{
    dp[0] = arr[0];

    for(int i=1; i<n; i++)
    {
        int pick = arr[i];
        if(i>1)
            pick += dp[i-2];
        int notpick = 0 + dp[i-1];

        dp[i] = max(pick, not pick);
    }
    return dp[n-1];
};
```

TC → O(N)

SC → O(N)

↳ external array dp[n+1]

Space Optimization.

Notice, for space opt. try noticing the elements required to compute in tabulation. In this case.

dp[i], dp[i-1], dp[i-2]
like we have solve for prev. problems, it's the same logic.

```
int f(int n, vector<int> &arr)
{
    int prev = arr[0], prev_2 = 0;

    for (int i=1; i<n; i++)
    {
        int pick = arr[i];
        if(i>1)
            pick += prev_2;
        int notpick = prev;
        int curr_i = max(pick, notpick);
        prev_2 = prev;
        prev = curr;
    }
    return prev;
}.
```

TC → O(N)
SC → O(1)

Lecture 6: Vacation Problem (Atcoder) (2-D)

Sample n = 3

10	40	70
20	50	60
30	60	90

Ans → 210 (70 + 50 + 90)

Greedy fails because we will consider the maximum point / activity each day, respecting the condition that activity can't be performed consecutive days. Eg.

n = 2

10	50	1
10	100	11

According to greedy → 50 + 11 ⇒ 61

But the ans is 110 (10 + 100)

Where we have to try all possible choices as our next solution. We will use recursion to generate all the possible choices.

Steps to write recurrence

① Express the problem in terms of Indexes

One clear parameter which breaks the prob. in different steps is 'n' or 'number of Day'. So 'Days/n' can be expressed in terms of indexes.

At any day we have three activities (0, 1, 2) to be performed. We can only when we know which activity we choose the previous day. Thus we need another parameter 'last' in order to track the activity done yesterday.

Now there are three options each day (0, 1, 2) which becomes 'last' of the next day. If we are starting from (n-1) day, then for that day we can choose all three options by setting the value of last = 3.

Step-2) Try out all possible choices at a given index.

Our function has 2 parameters f(day, last)

We are writing a top-down recursive func^. We start from (n-1) day to 0th day. Thus our base case will be at (day == 0)

Other than base case, whenever we perform an activity it's points will be given by arr[day][i] and to points from other days, we will let recursion do it's job. f(day-1, i), Passing 'i' so that current day's activity becomes next day's last.

Step-3) Take Maximum of all choices.

Simply we take max of all ~~op~~ sol^ provided by Recursion.

Memoization : There are over-lapping subprobs, notice if we make recursion tree. Follow the EXACT SAME steps for memoization as shown previously.

Code

```
int f(int n, int last, vector<vector<int>>&arr, vector<vector<int>>
                                                    &dp){
    if(dp[n][last] != -1) return dp[n][last];
    if(n==0){
        int maxi=0
        for(int i=0; i<3; i++){
            if(i != last){
                maxi = max(maxi, arr[n][i]);
            }
        return maxi;
    }
    int maxi = 0
    for(int i=0; i<3; i++){
        if(i != last){
            int curr = arr[n][i] + f(n-1, i, arr, dp);
            maxi = max(maxi, curr);
        }
    }
    return dp[n][last] = maxi;
}
```

TC → O(N×4×3)

There are (N×4) states and for every state we are running an iterative loop (3) times.

SC → O(N) + O(N×4) ≈ O(N)

↙ Recursion Stack ↘ 2D array

Tabulation

① Declare a dp[] of size [n][4] i.e dp[n][4]
② We initialize the base case, and we know the base case is when day=0, Thus.

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

③ Set an iterative loop from 1 to n and for every index set its value according to recursive logic.

Code

```
int f(int n, vector<vector<int>> arr) {
    vector<vector<int>> dp(n, vector<int>(4,0));
    dp[0][0] = max(arr[0][1], arr[0][2]);
    dp[0][1] = max(arr[0][0], arr[0][2]);
    dp[0][2] = max(arr[0][0], arr[0][1]);
    dp[0][3] = max(arr[0][0], max(arr[0][1], arr[0][2]));

    for(int day=1; day<n; day++) {
        for(int last=0; last<4; last++) {
            dp[day][last] = 0;
            for(int task=0; task<3; task++) {
                int pts = arr[day][task] + dp[day-1][task];
                dp[day][task] = max(pts, dp[day][task]);
            }
        }
    }
}
```

T.C. $\longrightarrow$ $O(N*4*3)$

S.C. $\longrightarrow$ $O(N*4)$

Space Optimization.

If we look closely at the relation,

$dp[day][last] = max(dp[day][last], points[day][task] + dp[day-1][task])$

Here the task can be anything from 0 to 3 and day − 1 is the previous stage of recursion. So, in order to compute any dp array value, we only require the last row to calculate it.

day \ last	0	1	2	3
0	✓	✓	✓	✓
1				
⋮				
n-1				

This row is initially filled.

This row only needs previous row values.

* So rather than storing the entire 2-D Arrays of size $N*4$, we can just store values of size 4 (say prev).
* We can take dummy array, again of size 4 (temp) and calculate the next row's value using the array we stored in step 1.
* After that whatever we move to the next day, the temp array becomes our prev for the next step.

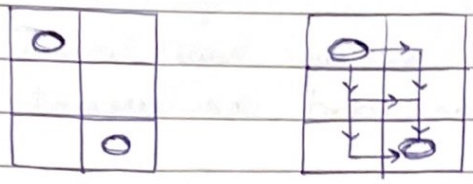
(*) At last of prev[3] will give us the soln.

```
int f(int n, vector<vector<int>>& points)
{

    vector<int> prev(4,0);

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

    for(int day = 1; day < n; day++) {
        vector<int> temp(4,0);
        for(int last = 0; last < 4; last++) {
            temp[last]
            for(int task=0; task<=2; task++) {
                if(task != last) {
                    temp[last] = max(temp[last],
                        points[day][task] + prev[task];
                }
            }
        }
    }
    prev = temp;
}

return prev[3];
}
```

Very Imp Ques^n to understand all types of Grid Probs.

Lecture : 7 → Unique Paths (Leetcode) (DP on Grids).

Sample : m = 3 , n = 2.



output - 3

Steps to form Recurrence :-

Step 1) Express the problem interms of indexes.

We are given two variable M and N, representing the dimension of a 2D matrix. for every different prob, this M & N will change.

We can define the function with two parameters i & j, where i & j represent the row and column of the matrix

∴ f(i, j) = Total amount of unique paths from matrix[0][0] to matrix[i][j]

↳ Basically it gives ans for the subproblem.

⊙ We will be doing Top-down recursion, i.e we will move from [M-1][N-1] and try to find our way to the cell [0][0].
Therefore at every index, we will try to move up and towards left.
(opposite to Ques^n because we are going down to top)

Base Case

There are two base Cases:-

① When $i=0$ & $j=0$ that is we have reached our destination so we can count the current path hence we return 1.

② When $i<0$ (or) $j<0$, it we mean we went ~~of~~ out of the matrix and couldn't find right path, hence return 0.

Step2) Try out all possible choices.

We have two choices, either we could go up (↑) (or) we could go left (←), by reducing row no. & column no. respectively.

Step3) Take maximum of all choices.

We have to count all possible paths, we will return. Sum of all choices (up & left)

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

    up = f(i-1, j);
    left = f(i, j-1);

    return up + left;
}
```

Memoization.

- Declare dp[M][N] = -1.
- Store all the states in dp, as.
  dp[M][N] = up + left.
- Check if dp[M][N] != -1, then return the calc value.

Time Complexity ⟶ O(N x M).

Space Complexity ⟶ O((N-1) + (M-1)) + O(M * N).

O((N-1) + (M-1)) → Recursion Stack Space. here, (N-1) + (M-1) is max path length

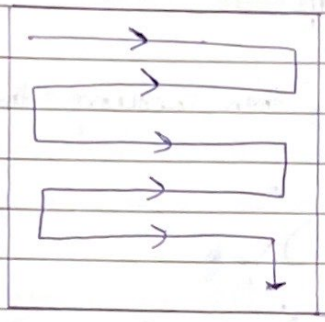
O(M * N) → DP array.

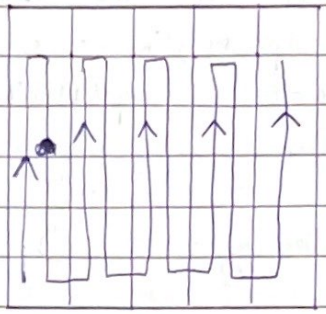
Tabulation.

Steps to Convert Recursion to tabulation

- Declare dp[][] array of siz [M][N].
- First initialize the base condition i.e dp[0][0] = 1.
- Our answer should get stored in dp[M-1][N-1]. We want to move from (0,0) to (M-1, N-1). But we can't move arbitrarily, we should move such that at a particular i & j, we have all the values required to compute dp[i][j].
- If we see the memoized code, value required for dp[i][j] are : dp[i-1][j] & dp[i][j-1]. So we only use the previous row & column value.
- We have already filled the top-left corner, if we move in any of the two ways (next page), at every

cell we do have all the previous values required to compute its value.





- We can use two Nested loops to have this traversal.
- At every cell we calculate up & left as we had done in the recursive solution and then assign the cell's value as (up + left)

Code

```
int f(int m, int n, vector<vector<int>> &dp) {

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

Time Complexity — O(M * N).

Space Complexity — O(M * N)

Space Optimization.

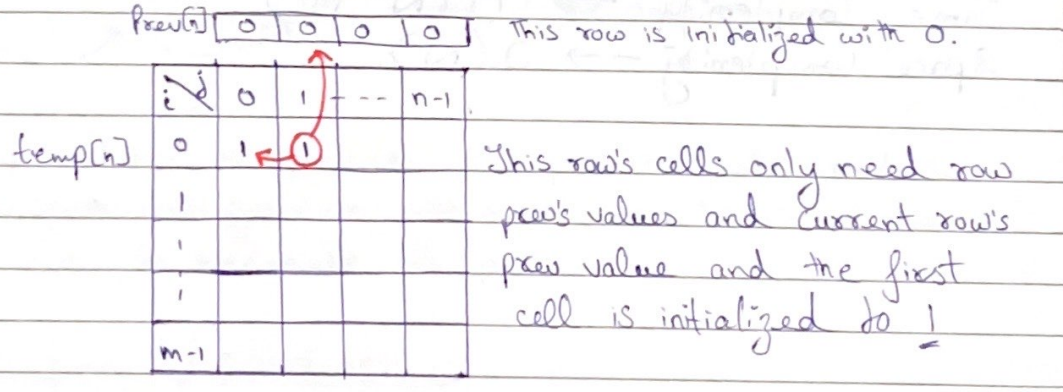
if we look closely the relation;

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

We can see we only need the previous row & column to compute dp[i][j]. Thus we can optimize even further.

Initially, we take a temp row and initialize it with 0.

Now the current row only needs the prev. row value and current row's prev. column value to compute dp[i][j].



At the next step, the temp[] becomes the prev of the step and using its values we can calc. the next row's values. At last prev[n-1] will give answer.

```
int f(int m, int n) {

    vector<int> prev(n,0);
    for(int i=0; i<m; i++) {
        vector<int> temp(n,0);
        for(int j=0; j<n; j++) {
            if(i==0 && j==0) temp[j]=1;
            Else {
                int up=0, left=0;
                if(i>0)
                    up= prev[j];
                if(j>0)
                    left = temp[j-1];
                temp[j]= ~~prev~~ up + left;
            }
        }
        prev = temp;
    }
    return prev[n-1];
}
```

Time Complexity → O(N * M)
Space Complexity → O(N).

Lecture 8: Minimum Path Sum (Leetcode) (DP on Grid)

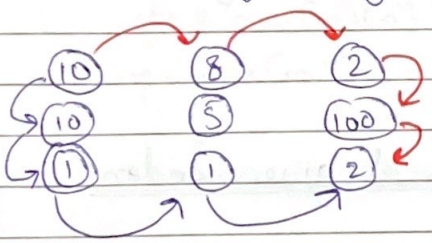
Problem Statement

We are given an `N x M` matrix with values and we ~~the~~ have to calculate the minimum sum / path to travel from top left to Bottom Right.
Note : We can move only in Rightward and downward motion.

Solution

This question is slight modification of prev problem.

Why greedy fails?
If we try greedy, then at any position we have two options, either go right or go down. Now let's take the following Eg:-



This is Greedy Sol^n i.e 122

This is the ans i.e 24

Thus we have to generate all possible paths and see which path will give us minimum sum. Thus we use Recursion to Generate all paths.

Steps to form Recurrence :-

(1) Express the Problem in terms of Indexes -
We can define a func^n with two parameters i & j where i and j represent the row and column

of the matrix.

$f(i,j) \rightarrow$ Minimum path Sum from Matrix[0][0] to matrix[i][j].

Note :- We will be doing top-down Recursion, we will move from cell [M-1][N-1] and try to find our way to the cell [0][0]. Therefore at every index, we will try to move up and towards the left.

Base Case

(1) When i=0 (&) j=0, hence we have reached our final destination and we have to add it up in the soln.

(2) If i<0 (OR) j<0, hence we have gone out of bounds and thus we need to return a big value (1e9) so as to reject this path.

(2) Try out all possibles choices at given Index

Since we are writting top-down Recursion, we reduce 'i' by 1 to go up (OR) we can reduce 'j' to left. And we also add the current index value too.

(3) Taking Minimum of all Choices

As per the question, we return the minimum as answer

Memoization.

Same → Declare 2D dp array of size M×N Store the minimum result/overlapping subproblems and return it whenever possible.

Code

```
int f(int m, int n, vector<vector<int>>& dp,
                    vector<vector<int>>& arr)
{
    if(m==0 && n==0) return arr[m][n];
    if(m<0 || n<0) return INT_MAX;
    if(dp[m][n] != -1) return dp[m][n];

    int left = INT_MAX, up = INT_MAX;

    left = arr[m][n] + f(m, n-1, dp, arr);
    up = arr[m][n] + f(m-1, n, dp, arr);

    return dp[m][n] = min(up, left);
}
```

Time Comp → O(N * M).
Space Comp → O((M-1)+(N-1)) + O(N * M).

Tabulation

Same :- Completely Same as prev problem just update the if/else condition acc. to ques^n and Memoization as shown above.

Code

```
int f(int m, int n, vector<vector<int>>& arr)
{
    vector<vector<int>> dp(m, vector(n, -1));

    for(int i=0; i<m; i++) {
        for(int j=0; j<n; j++) {
            if(i==0 & j==0) dp[i][j] = arr[i][j];
            else {
                int left, up = ~~maar~~ INT_MAX;
                if(i>0) up = arr[i][j] + dp[i-1][j];
                if(j>0) left = arr[i][j] + dp[i][j-1];

                dp[i][j] = min(up, left);
            }
        }
    }
    return dp[m-1][n-1];
}
```

Time Comp - O(N*M)
Space Comp - O(N*M).

Space Optimization.

It also Complety same, take two array prev and temp.
One for prev row and one for current columns.

Time Comp - O(N * M)
Space Comp - O(N).

Path

Lecture :9 – Minimum Sum In Triangle Grid (Triangle) (LeetCode)

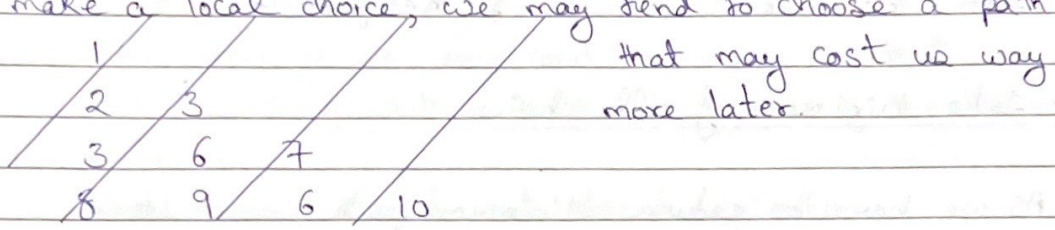
Problem

Given a Triangular Matrix. We have to calculate the minimum path sum to reach the bottom-most row.

Solution.

It is a slight modification of the prev problem

Greedy fails here too, as we know that whenever we make a local choice, we may tend to choose a path that may cost us way more later.



Steps to form the Recursive Solution :-

① Express the problem in terms of Indexes.

Our Ultimate aim is such that we reach the last row. We can defin f(i,j) such that it gives us the minimum path sum from the cell [i][j] to last row.

f(i,j) → Minimum Path Sum from matrix [0][0] to the last row of matrix.

We want to find f(0,0) and return it as our answer

Base Case :- when i == N-1, that is when we have reached last row, so the min path from that cell to the last row will be the value of cell itself, hence return mat[i][j].

Since we have only Two options, to go either down (or) down right. Observe that we'll never go out of bounds Therefore only one base case if sufficient.

(2) Try all possible choices.

Since we have 2 options, we can either increase (i) by 1 to go down (or) increase both (i) & (j) by 1 to go down-right. Make them recursive functions to get answer Add the current cell too, to the function call.

(3) Take Minimum of all choices

As we have to return Minimum path, we store the minimum of down & diagnol.

Memoization.

Same as all prev. Problems :-

1. Declare a dp[] of size (N)(M).
2. Store minimum of down & diagnol.
3. Return dp values whenever they are not (-1).

Code

```
int f(int i, int j, vector<vector<int>>& dp, vector<vector<int>>& arr)
{
    if(i==n-1) return arr[i][j];
    if(dp[i][j]!=-1) return dp[i][j];

    int down = arr[i][j] + f(i+1, j, dp, arr);
    int diagnol = arr[i][j] + f(i+1, j+1, dp, arr);
    return dp[i][j] = min(down, diagnol);
}
```

Time Comp → $O(N * N)$.

Space Comp → $O(N) + O(N * N)$.

Tabulation.

As in Recursion / Memoization, we moved from 0 to N-1, in tabulation we move from (N-1) to 0, i.e last row to first.

- Declare DP[][] of size [N][N].
- First initialize the base conditions value, i.e ~~that~~ the last row of dp matrix to the last row of the triangle matrix.
- Our answer should get stored dp[0][0].
- Since we have already filled values from (N-1) row we'll start from (N-2) and move upwards.
- Use 2 Nested loops.

Code

```
int f(vector<~~int~~vector<int>>& arr, int n){
    vector<vector<int>> dp(n, vector<int>(n,0));
    for(int i=0; i<N; i++) dp[n-1][i]= arr[n-1][i];

    for(int i=n-2; i>=0; i--){
        for(int j=i; j>=0; j--){

            int down = arr[i][j] + dp[i+1][j];
            int dignol= arr[i][j] + dp[i+1][j+1];
            dp[i][j]= min(down, diagnol);
        }
    }
    return dp[0][0];
}
```

Time Comp — O(N*N)
Space Comp — O(N*N)

Space Optimization

Observe at the relation.

dp[i][j] = matrix[i][j] + min(dp[i+1][j], dp[i+1][j+1]);

We only need the next row, to calc dp[i][j].
Thus we can optimize it.

Take a dummy [] (say front) and initialize it with triangle [] last row as done in tabulation.

Now the current row (say curr) only needs front row to calc dp[i][j].

Code

```
int f(vector<vector<int>>& arr, int n) {
    vector<int> front(n,0), curr(n,0);
    for(int i=0; i<n; i++)
        front[i] = arr[n-1][i];
    for(int i=n-2; i>=0; i--) {
        for(int j=i; j>=0; j--) {
            int down = arr[i][j] + front[j];
            int diagnol = arr[i][j] + front[j+1];
            curr[j] = min(down, diagnol);
        }
        front = curr;
    }
    return front[0];
}
```

Time Comp = O(N*N)
Space Comp = O(N).

Lecture 10 - Minimum falling Path Sum (leetcode)
(Variable Starting and Ending pt)

We are given an 'N * N' matrix and we need to find the minimum Sum path from first row to last row.
We can move in Three Direction → Down.
Bottom Left
Bottom Right

Solution.
This problem is very similiar to previous problems

The Recurrence
f(i,j) → Minimum path Sum from the first row to cell[i][j].

Since, we are computing from last row to our way up (first row).
We take every element in the last row and compute the minimum path Sum and return the smallest answer out of all of them.

Base Case

① When i==0 that mean we have reached the first row, and we just simply want to return the Element we landed on.
return arr[0][j];

② At very cell we have three options (we are writing ~~top-down~~ Bottom-up). up, upleft & upright.

for going (up) there is no problem because we'll reach top row and return the value as it is.

But for top-left & top-Right, we can go out of bounds. To prevent that we return INT_Max in order to ignore that path.

Performing Tasks & Computing Minimum

We'll make 3 Recursive calls for up, upleft & upright, compute them. Take the minimum of the three choices and store it in a dp[][].
And return dp.

Code

```
int f(int r, int c, vector<vector<int>>& dp, vector<vector<int>>& arr)
{
    if(c<0 || c>=arr.size()) return INT_MAX;
    if(r==0) return arr[0][c];
    if(dp[r][c]!=-1) return dp[r][c];
    int up = arr[r][c] + f(r-1, c, dp, arr);
    int upleft = arr[r][c] + f(r-1, c-1, dp, arr);
    int upright = arr[r][c] + f(r-1, c+1, dp, arr);

    return dp[r][c] = min(up, min(upleft, upright));
}
int main {
    int mini;
    for(int i=0; i<r; i++) {
        mini = min(mini, f(r, i, dp, arr));
    }
    return mini; }
```

~~Tabulation~~ Time Comp → O(N*N).

Space Comp → O(N) + O(N*N)

Tabulation.

1. Declare dp array size [N][N]
2. first initialize base conditions, the first row of the dp array to the first row of input matrix
3. If we look at the memoized code, values required to compute dp[i][j] is...

   dp[i][j] = dp[i-1][j] , dp[i-1][j-1] & dp[i-1][j+1].
   Thus we need values from (i-1) row.
4. Since we have filled the (i=0) row, we start from i=1 and move downwards.
5. Use 2 nested loops for traversal.

Time Comp — O(N*N).
Space Comp — O(N*N).

PTO

Code

```
int f (vector<vector<int>> arr)
{
    int n, m = ~~matrix~~si arr.size();
    vector<vector<int>> dp(n, vector<int>(m, 0));

    for(int i=0 ; i<m; i++)
        dp[0][i] = arr[0][i];

    for(int i=1 ; i<n; i++) {
        for(int j=0; j<m ; j++) {
            int up, upleft, upright= maxi;
            up= arr[i][j] + dp[i-1][j];
            if (j-1>=0)
                upleft= arr[i][j] + dp[i-1][j-1];
            if (j+1<m)
                upright = arr[i][j]+ dp[i-1][j+1];

            dp[i][j]= min(up, min(upleft, upright));
        }
    }
    int ans = INT_MAX;
    for(int i=0; i<m; i++) {
        ans = min(ans, dp[n-1][i]);
    }
    return ans;
}
```

Space Optimization

If we look closely,

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

We can see that we only need previous row, like we required in previous problems.

Initially we can take dummy [] (prev). Initialise it with input matrix first row.

Use the another curr[] and prev[] to calculate dp[i][j].

Lecture 11 - Cherry pickup 2 (3D DP) (DP on Grids).

Problem Statement:

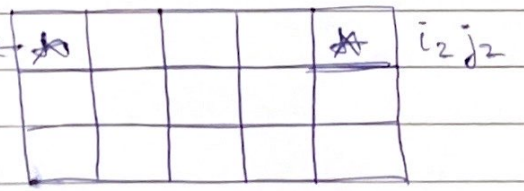
We are given 'N * M' matrix. Every cell has some cherries ~~it~~ in it.
We are given 2 users. One is standing on top left row (0, 0) and the other is standing on top right (0, M-1).
They can travel in any direction (↙ ↓ ↘) to reach the last row.
Find the maximum no. of cherries collected by both of them combined
Note If both land on the same cell then it will be counted only once.

Solution.

This question is clearly an example of ~~variable~~ fixed starting point and ~~fixed~~ variable ending point.
As we have done previously, it is recommended to solve it in top-Down Recursion fashion.

Recurrence.

(1) We need four parameters to describe the position of both users
(i_1, j_1, i_2, j_2).



If we observe, both users are at the same row and will always be at same row simultaneously irrespective of the column.
Thus we can remove the row redundancy.
And update the parameters as [i][j1][j2]

$f(i, j_1, j_2)$ ⇒ Maximum chocolates/cherries collected by Alice from cell [i][j1] and bob from cell [i][j2] till last row.

Base Case.

① When i == N-1, we would have reached last row and we will check if both the users have reached on the same cell or not. If we yes, we will return cell[i][j1]. Else return cell[i][j1]+ cell[i][j2]

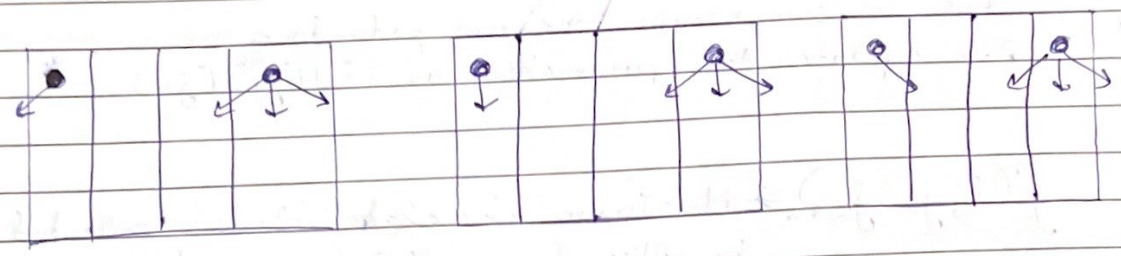
② Checks for Out of Bound Cases. We should go $j_1, j_2 < 0$ || $j_1, j_2 >= m$.

Note :- for Base Case there are 2 things to keep in mind.
① Destination
② Out of Bounds
Always write ② if first.

Try Out All Possible Choices

At every cell we have 3 options to traverse :-
(↙ ↓ ↘)

Try to Understand if we want to move both Alice And Bob together. And both of them can move in 3 Direction.



for Every Single Move of Alice, Bob has three options

Therefore, we have a total of 9 options.

Note if $(j_1 == j_2)$ as discussed in base case, we will consider chocolates collected by one of them.

Take Maximum Of All

As per question, Keep a track of Maximum path possible. Keep it stored it in 3D dp[][][].

Memoization

As Always The Same :- Create a 3D Array for dp[i][j1][j2] Initialize it with -1. Store the maxi variable in it. and return the ans.

Code

```
int f( int i, int j1, int j2, int r, int c, vector<vector<int>>& arr,
                    vector<vector<vector<int>>>& dp )
{
    if (j1<0 || j2<0 || j1>=c || j2>=c) return -1e8
    if ( i== r-1){
    {   if (j1==j2){
            return arr[i][j];
        Else
            return arr[i][j1]+ arr[i][j2];
    }
    if (dp[i][j1][j2] != -1) return dp[i][j1][j2];

    int maxi = INT_MIN;
    for(int di = -1 ; di<=1 ; di++) {
        for (int dj = -1 ; dj <= 1 ; dj++) {
            int ans;
            if (j1==j2)
                ans = arr[i][j] + f(i+1, j1+di, j2+dj, r, c, arr, dp);
            Else
                ans = arr[i][j2] + arr[i][j1] + f(i+1, j1+di, j2+dj,
                                                  r, c, arr, dp);
            maxi = max(ans, maxi);
        }
    }
    return dp[i][j1][j2] = maxi ;
}
```

Time Comp — $O(N*M*M) \times 9$

At max, there will be (N*M*M) recursive calls to solve a new problem and in every call, two nested loops together run for 9 times

Space Complexity – $O(N*M*M) + O(N)$

$O(N*M*M)$ → dp[]

$O(N)$ → recursion Stack

Tabulation

- For tabulation, we need to understand what a cell in 3D DP mean. So ~~we~~ when we say dp[2][0][3]. It means that we are getting the value of maximum chocolates collected by Alice And Bob, which Alice [2][0] and Bob [2][3].

In the recursive code, our base condition is when we reach the last row. Thus initialize. dp[n-1][][], as the base condition, dp[n-1][j1][j2]. if (j1 == j2) arr[i][j1]
Else arr[i][j1] + arr[i][j2].

- Now we will move from dp[N-2][][] to dp[0][][].
- We need 3 nested loop for the traversal.
- The outer three loops for traversal & the inner two loops are for those 9 options at every cell.
- We will then set dp[i][j1][j2] as the max of all 9 options.
- At last we will return dp[0][0][m-1] as our answer.

Time Comp → $O(N*M*M) \times 9$.
Space Comp → $O(N*M*M)$.

Code

```
int f(int n, int m, vector<vector<int>>& grid) {

    vector<vector<vector<int>>> dp(n, vector<vector<int>>
                (m, vector<int>(m, 0)));

    for(int j1=0; j1<m; j1++) {
        for(int j2=0; j2<m; j2++) {
            if(j1==j2)
                dp[n-1][j1][j2] = grid[n-1][j1];
            else
                dp[n-1][j1][j2] = grid[n-1][j1] + grid[n-1][j2];
        }
    }
    for(int i=n-2; i>=0; i--) {
        for(int j1=0; j1<m; j1++) {
            for(int j2=0; j2<m; j2++) {
                int maxi = INT_MIN;
                for(int di=-1; di<=1; di++) {
                    for(int dj=-1; dj<=1; dj++) {
                        int ans;
                        if(j1==j2)
                            ans = grid[i][j1];
                        else
                            ans = grid[i][j1] + grid[i][j2];
                        if((j1+di<0 || j1+di>=m) ||
                           (j2+dj<0 || j2+dj>=m))
                            ans += -1e8;
                        else
                            ans += dp[i+1][j1+di][j2+dj];
                        maxi = max(ans, maxi);
                    }
                }
                dp[i][j1][j2] = maxi; } } }
```

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

Lecture 31: Buy & Sell Stocks with Transaction Fees

Same Concept of Stocks II, it's only that after A Complete TRANSACTION we have to pay a fees.
So while adding the price of stock during selling SUBTRACT ~~Fees~~ Transaction as well.

The Rest Remains the Same.

Lecture 32: Longest Increasing Subsequence

It is another modification of the subsequence problem where we use the pick/not pick algo.
It's just that Since it is INCREASING, we have to keep a prev_variable to keep track, that the elements should increasing.
The Recurrence / presenting in terms of index ⇒

f(ind, prev) ⇒ At the current 'ind' what is Longest Increasing Subs Achieved

The code below will work for element >=0 NOT for negative numbers.

Time Comp ⟶ O(N x N)
Space Comp ⟶ O(N x N) + O(N)

initialize it with INT_MIN.

```
int f(int ind, int prev, vector<int>& arr) {

    if (ind == arr.size())
        return 0;
    if (dp[ind][prev] != -1)
        return dp[ind][prev];

    int notpick = f(ind+1, prev, arr, dp);
    int pick = 0;
    if (arr[ind] > prev)
        pick = 1 + f(ind+1, arr[ind], arr, dp);

    return dp[ind][prev] = max(pick, notpick);
}
```

Tabulation:

The Same Rules will be applied..

1. Bas Case
   ↳ Since we will initialize the dp[][] by 0, no need to modify / change anything.
2. Traverse the Changing parameters in reverse fashion
   ind ⇒ n-1 ⟶ 0
   prev ⇒ ind-1 ⟶ -1
3. Copy recurrence & make sure to follow the right shift rule.

Code

```
int f(vector<int> &arr) {

    int n = arr.size();
    vector<vector<int>> dp(n+1, vector<int>(n+1, 0));

    for(int ind = n-1; ind >= 0; ind--) {
        for(int prev = ind-1; prev >= -1; prev--) {

            int len = 0 + dp[ind+1][prev+1];
            if(prev == -1 || arr[ind] > arr[prev])
                len = max(len, 1 + dp[ind+1][ind+1]);

            dp[ind][prev+1] = len;
        }
    }
    return dp[0][-1+1]
```

Now, Since there is 'ind' & 'ind+1' we can space optimize it using the same old prev & ~~curr~~ curr vector's concept.

Time → $O(N^2)$
Space → $O(N) \times 2$

Tabulation 2 (SC - O(N))

Now Try understand,
We declard a dp[] of size (n), where

dp[i] → Signifies the longest IS that ends at inden (i).

Eg:-

| 5 | 4 | 11 | 1 | 16 | 8 |

– Arr

See, initially we can can that individually every element is it own LIS. Thus initialise the dp[] by 1.

① At (5) → Since there is no prev inden, we move on

dp – | 1 | 1 | 1 | 1 | 1 | 1 |
i ↗

② At (4) → We have a prev as (5), which is greater Thus not a LIS, we move on

dp – | 1 | 1 | 1 | 1 | 1 | 1 |
i ↗

③ At (11) → We have two prev 5 & 4, and we can take any one of them into LIS, Thus changing it to 1+ dp[prev] i.e

dp – | 1 | 1 | 2 | 1 | 1 | 1 |
i ↗

④ At (1) → We can't anything.

| 1 | 1 | 2 | 1 | 1 | 1 |

(5) At (16) → We keep add dp[prev] values to dp[i], and we a check of max.
Notice after taking (11) & (16) into LIS
the dp becomes 3, because 11 ⇒ 2
Thus dp[i] = 1 + dp[prev].

dp — | 1 | 1 | 2 | 1 | 3 | 1 |

(6) At (8) → Only one out of 5, 4, 1 can be picked

∴ dp — | 1 | 1 | 2 | 1 | 3 | 2 |

Keeping a global maximum we get our length = 3

```
int f(vector<int>& arr){
    int n = arr.size();
    vector<int> dp(n+1, 1);
    int ans = -1;
    for(int i = 0, i<n; i++){
        for(int j = 0; j<i ; j++){
            dp[i] = max(dp[i], 1+dp[j]);
            if (dp[j] < dp[i])
                dp[i] = max(dp[i], 1+dp[j]);
        }
        maxi = max(maxi, dp[i]);
    }
    return maxi;
}.
```

M T W T F S S

Page No.:

Date: 07

YOUVA

Binary Search.

Refer the blog in bookmarks for better understanding.

Here we will take a temp[], to ~~CAL~~ CALCULATE LIS, NOTE: To Calculate, it's not the ans

* The logic is to keep on adding elements in back if curr > prev, (OR)

* replace the curr element with first Greater element in temp using Binary Search (OR) Lower Bound function

Eg

1	7	8	4	5	6	-1	9

~~S-1)~~ S-1) temp → 1

S-2) temp → 1, 7

S-3) temp → 1, 7, 8

S-4) temp → 1, ~~7~~, 8
↓
temp → 1, 4, 8
} replaced 7 by 4, since it's the first Greater element.

S-5) temp → 1, 4, 5

S-6) temp → 1, 4, ~~5~~, 6

S-7) temp → -1, 4, 5, 6

S-8) -1, 4, 5, 6, 9] temp

The Size of temp is our answer.

```
int f(vector<int> &arr) {

    vector<int> temp;
    temp.push_back(arr[0]);

    for(int i=1; i<arr.size(); i++) {
        if (arr[i] > temp.back())
            temp.push_back(arr[i]);
        else {
            int ind = lower_bound(temp.begin(), temp.end(),
                      arr[i]) - temp.begin();

            temp[ind] = arr[i];
        }
    }
    return temp.size();
}
```

TC → O(N log N)
SC → O(N).

Lecture 383 → Largest Divisble Subset.

Try Imagine, if we sort the array and instead of find Longest Increasing by dp[i] > dp[j], we replace our logic to longiest divisble by dp[i] % dp[j] == 0

And Since the Array is sorted all pair in front will be divisble with all in back.

The code will be exactly same As Printing LIS

```
~~int~~ vector<int> f(vector<int>& arr){
    int n=nums.size(), last=0, mani=1;
    vector<int> dp(n,1), hash(n); temp;
    Sort(nums.bagin(), nums.end());
    for(int i=0; i<n; i++){ * hash[i]=i;
        for(int j=0; j<i; j++){
            if(nums[i]%nums[j]==0 && dp[i]<dp[j]+1){
                dp[i]=dp[j]+1;
                hash[i]=j;
            }
        }
        if(dp[i]>mani){
            mani=dp[i];
            last=i;
        }
    }
    temp.push_back(nums[last]);
    while(last != hash[last]){
        last=hash[last];
        temp.push_back(nums[last]);
    }
    return temp;
}
```

M T W T F S S
Page No.:
Date: 180
YOUVA

Lecture 34: Largest Chain String

This question also works on the previous logic, Simply Sord the String according to their size in decreasing and check if there is only one extra character between the current and it's next String. If yes add it to your answer otherwise don't.

```
int f(vector<string> & arr) {

    int n=arr.size(), maxi=-1;
    vector<int> dp(n,1);
    sort(arr.begin(), arr.end(), comp);

    for(int i=0; i<n; i++) {
        for(int j=0; j<i, j++) {
            if(check(arr[i], arr[j] && dp[i]<dp[j]+1)
                dp[i]=dp[j]+1;
        }
        maxi = max(maxi, dp[i]);
    }
    return maxi;
}

bool comp(string & a, String & b)
    return a.size() > b.size();
```

PTO

```
bool Check( string &a, string &b){
    if(a.size() != b.size()+1)
        return false;

    int i=0, j=0;
    while(i<a.size()){
        if(a[i]==b[j]){
            i++;
            j++;
        }
        else
            i++;
    }
    return i==a.size() && j==b.size();
}
```

We need to check that if after iterating over longer both the strings are completely exhausted if yes, then return true, Otherwise return false.

#Lecture 35 - Longest Bitonic Subsequence.

It says the sequence can be any of 4 forms

(1) First Increasing, Then Decreasing.
(2) Only Increasing.
(3) Only Decreasing.
(4) First Decreasing, then Increasing.

for This, Now try to Understand.

Arr = 1, 11, 2, 10, 4, 5, 2, 1

Ans ⇒ 6 (1, 2, 10, 5, 2, 1)

Now see, if we calculate LIS, dp we get

dp[] ⇒ | 1 | 2 | 2 | 3 | 3 | 4 | 2 | 1 |

Now if we reverse the arr (or) make dp[] from Last to ~~be~~ first. we can find Decreasing but with LIS logic.

Arr = 1, 2, 5, 4, 10, 2, 11, 1.

dp[] = | 1 | 2 | 3 | 3 | 4 | 2 | 5 | 1 |

Now if we Add both dp tables and Subtract (1) because of common element we will get, but reverse dp2 because the indexing got reversed.

ans = | 1 | 6 | 3 | 6 | 5 | 6 | 3 | 1 |

ans = 6

Lecture 36 :- Number Of Longest Increasing Subsequence.

We just need to tell how many LIS are present in an array.

arr = [1, 3, 5, 4, 7]

Ans ⇒ 2 ({1,3,5,7}, {1,3,4,7}).

for this keep a cnt[] initialised to 1, .

```
if (nums[i] > nums[j] && dp[i] < dp[j]+1)
        dp[i] = dp[j]+1;
        cnt[i] = cnt[j]  // make them same, as in
                         that since till here we probably have
                         only 1 or a variable 'x' ways so we
                         carry it forward.

else if (nums[i] > nums[j] && dp[i] == dp[j]+1)
        cnt[i] += cnt[j]  // in the Abv eg; this would
                          occur at nums[i] = 7; where we find
                          multiple value same and we encounter
                          2 paths which lead to LIS = 4, Thus we
                          add the Ans
```

** Code Remains exactly the same, with abv changes and then keep a global variable to store ans, whereever maxi == dp[i]; add it's cnt[i] value to ans.

Lecture 28 : Stocks II

We are given an array 'prices' where price of stocks are given, we can at a time buy one stock and we have to sell it before buying another stock, we can repeat this process as many times as want, to maximise the ~~result~~ profit

prices[] = [7, 1, 5, 3, 6, 4]

Output = 7 [(1, 5) + (3, 6)]

Now try to understand and observe that every stock we have two option Pick / Not pick. But we need an extra variable which will tell us, if we can ~~by~~ buy the current stock on that particular day or not.

We have to maximize our output, thus we take maximum of all cases.

Think if we are buying a stock we have to SUBTRACT that stock's price from ~~as~~ ans. ~~Add~~ And we'll ADD when we sell.

Maintain a ~~by~~ 'buy' variable if buy = 1, purchase it (or) ignore it, else buy = 0 sell it (or) Don't sell it.

Base if (ind == n), then irrespective we have bought (or) sell the stock, 'return 0' for no loss.

Note : After every call keep updating the 'buy' variable, accordingly.

Memoisation

```
int f(int ind, int buy, vector<int>& arr,
                        vector<vector<int>>& dp) {

    if(ind == arr.size())
        return 0;
    if(dp[ind][buy] != -1)
        return dp[ind][buy];

    if(buy == 1)
        dp[ind][buy] = max(-arr[ind] + f(ind+1, 0, arr, dp),
                           f(ind+1, 1, arr, dp));
    else
        dp[ind][buy] = max(arr[ind] + f(ind+1, 1, arr, dp),
                           f(ind+1, 0, arr, dp));

    return dp[ind][buy];
}
```

Time Complexity = O(N x 2)

Space Complexity = O(N x 2) + O(N)

Tabulation

Same procedures ① Base Case Manipulation.
② Write variables (i, buy)
③ Copy Recurrence.

```
int solve ( vector<int> arr ){
    int n= arr.size();
    vector<vector<int>> dp(n+1, vector<int>(2,0));

    dp[n][0]=0 , dp[n][1]=0; // base cases

    for(int i=n-1; i>=0; i++){
        for(int j=0; j<=1; j++){
            if(j==1)
                profit = max(-arr[i]+dp[i][0], dp[i][i]
            else
                profit= max(arr[i]+ dp[i][1], dp[i][0]);

                dp[i][j]= profit;
        }
    }
    return dp[0][1];
}
```

Time comp → O(N x 2)
Space comp → O(N x 2).

We can space optimize it by using prev and curr.

Lecture 29: Stocks III

The question is exactly the same as last question the only difference is that we can ~~one~~ (buy and sell) stocks at max '2' times.

That means we are allowed to perform transactions only twice

1 Transaction = ~~By~~ (Buy And Selling) of Stock.

Thus we can use the same and just add one more variable like the knapsack problem. To keep a track of No of Transactions made.
Try to understand only the case : when we had bought a stock, and when we are planning to sell it only then it is counted as A TRANSACTION, otherwise it won't

Base Case will add one more case when (CAP == 0)
That means we have reached the limit to purchase and thus we can return 0;

Time Complexity = $O(N \times 2 \times 3)$
Space Complexity = $O(N \times 2 \times 3) + O(N)$
↓
Auxiliary Stack Space.

Memoization

Create a dp[][][] to store ans.

```
int f(int ind, int buy, int cap, vector<int>& arr,
                    vector<vector<vector<int>>>& dp) {

    if (ind==n || cap==0)
        return 0;

    if(buy==1)
        dp[ind][buy][cap] = max(-arr[ind] + f(ind+1, 0, cap, arr,
                                              dp),
                                f(ind+1, 1, cap, arr, dp));
    else
        dp[ind][buy][cap] =
            max(arr[ind]+f(ind+1, 1, cap-1, arr, dp),
                f(ind+1,0,cap, arr, dp));

    return dp[ind][buy][cap];
};
```

Tabulation.

Same Rules : ① Base Case.
② Variables (i, buy, cap)
③ Copy Recurrence

Time Complexity — O(N x 2 x 3).
Space Complexity — O(N x 2 x 3).

```
int Solve (vector<int> & arr) {
    int n = arr.size();
    vector<vector<vector<int>>> dp(n+1, vector<vector<int>>(3,
                                   vector<int>(3, 0)));

    for(int i = n-1 ; i >= 0 ; i--) {
        for(int j = 0; j <= 1 ; j++) {
            for(int z = 1 ; z <= 2 ; z++) {
                if (j)
                    dp[i][j][z] = max(-arr[i] + dp[i+1][0][z],
                                      dp[i+1][1][z]);
                Else.
                    dp[i][j][z] = max(arr[i] + dp[i+1][1][z-1],
                                      dp[i+1][0][z] );
            }
        }
    }
    return dp[0][1][2]; // original State where we
                        made F calls.
}
```

If the Question is moulded, and we are given 'K' transactions instead of '2', just replace values and everything else will remain the same.

Space optimize by using curr[][], prev[][]

Lecture 30 → Buy & Sell Stocks with a Cooldown.

Complete same as Stock II, it's just that after a complete transaction, we have to serve a cooldown period of 1 stock.

prices[] = [1, 2, 3, 0, 3]
output = 3
↳ (2-1) cooldown[3] (0, 3)

So, in the same code in Stock II, in the else condition where we complete the TRANSACTION by selling we just do ind+2.

```
int f(int ind, int buy, vector<int>& arr) {
    if (ind >= n) → change
        return 0;
    if (dp[ind][buy] != -1)
        return dp[ind][buy];

    if (buy)
        dp[ind][buy] = max(-arr[ind]+f(ind+1, 0, arr),
                           f(ind+1, 1, arr));
    else
        dp[ind][buy] = max(arr[ind] + f(ind+2, 0, arr),
                           f(ind+1, 0, arr));

    return dp[ind][buy];
}
```

only change

Lecture-12: Subset Sum Equals to Target.

Problem

given an array arr[] with N positive integers. We need to find that if there exists a subset / subsequence with sum equal to 'K'. If yes return true / false.

Eg)

arr - [1, 2, 3, 4]

K = 4

Ans) True

Steps to form Recursive Solution.

S-1) Express the Problem in terms of Indexes.

We can ~~int~~ initially say f(n-1, target) which means that we need to find whether there exists a subsequence in the array from index 0 to (n-1), whose sum is equal to target. Generalizing.

f(ind, target) → check whether a subsequence exist in the array from index 0 to ind, whose sum is equal to index.

Base Case:

- if target == 0; it means we have already found a subsequence from the previous step, we can return true.
- if ind == 0; we need to check if arr[ind] == target if yes, return true else return false.

S-2) Try All possible Stuff.

- We will use the 'pick/not pick' technique. i.e.

(a) Include the current element in the subsequence.

We first try to find a subsequence without considering the current element, and make a recursive. f(n-1, target).

(b) Include the element in the subsequence.

Simple, this time we include the element arr[ind] in our subsequence and make recursive call to. f(n-1, target - arr[ind])

Note: only when target $\geq$ arr[ind].

3-3) Return (taken / not taken)

Among the taken / not taken we just need to find one 'true' and then return true for our function

Memoisation

- As usual, we create a ~~dp[n][k+1]~~ dp[n][k+1]. The size of the input array is 'n'; so index always lies between '0' to 'n-1'. The target can take any value between '0' to 'k'
- initialize with '-1'.
- follow as usual.

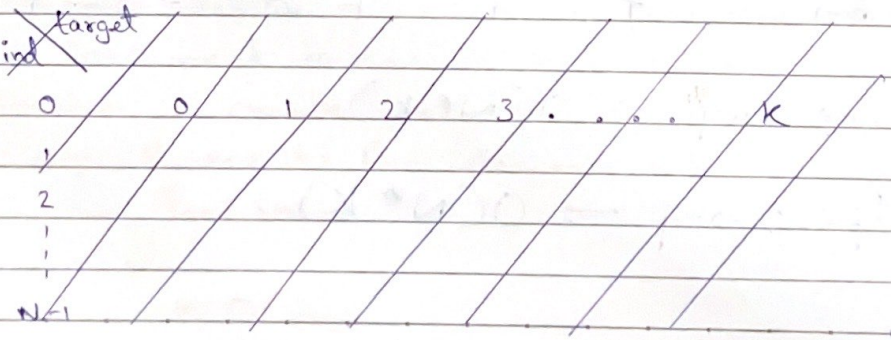
Code

```
bool f(int ind, int target, vector<int> &arr, vector<vector<int>>
                                                    &dp){
    if(ind==0)
        return arr[0]==target;
    if(target==0)
        return true;
    if(dp[ind][target]!=-1)
        return dp[ind][target];

    bool notpick = f(ind-1, target, arr, dp);
    bool pick = false;
    if(arr[ind]<=target)
        pick = f(ind-1, target-arr[ind], arr, dp);

    return dp[ind][target]= pick || notpick.
}
```

Time Comp — $O(N * K)$
Space Comp — $O(N * K) + O(N)$.

Tabulation.

Create the same dp[][] as in memoization. We can set it's type to bool and set all to false.



ind \ target	0	1	2	3	...	K
0	T	F	F	F	...	F
1	F	F	F	F	...	F
2	!	!	!	!		!
!	!	!	!	!		!
!	!	!	!	!		!
N-1	F	F	F	F		F

As always we first initialize the base condition from recursion :-

① If target == 0, ind can take any value from 0 to n-1 Thus we set the first column as true.

② The first row dp[0][] indicates that only the first element of the array is considered, therefore for the target value equal to arr[0], only call with that target will be true.
So set dp[0][arr[0]] = True.

③ Then we will set two nested loops for traversal the dp[][] as discussed previously.

④ Return dp[n-1][K] as answer.

ind \ target	0	1	2 ← if this was arr[0].	—	—	—	K
0	T	F	T	-	-		F
1	T	f	f	-	-	-	f
2	T	f	f	-	-	-	f
!	!	!	!	-	-	-	f
!	!	!	!	-	-	-	f
n-1	T	f	f	-	-	-	f

Time Comp → O(N * K)

Space Comp → O(N * K)

Code

```
bool f(int n, target, vector<int> &arr) {
    vector<vector<bool>> dp(n, vector<bool>(k+1, false));

    for(int i=0; i<n; i++)
        dp[i][0] = true;
    if(arr[0] <= target)
        dp[0][arr[0]] = True;
    for(int i=1; i<ind; i++) {
        for(int j=1; j<=target; j++){
            bool notpick = dp[i-1][j];
            bool pick = false;
            if(arr[i] <= target)
                pick = dp[i-1][target-arr[0]];

            dp[i][j] = pick || not pick;
        }
    }
    return dp[n-1][k];
}.
```

Space Optimization

$$dp[ind][target] = dp[ind-1][target] \;||\; dp[ind-1][target-arr[ind]]$$

Thus to calculate this we only need the prev. row values. Hence we don't need to main an entire array.

Note: Whenever we create a new row (say curr), we need to explicitly set it's first element to be true. acc. to base condition.

Space Comp → O(K).

Lecture 13 :- Partition Array into two ~~ways~~ Arrays to Minimise Sum Difference.

This problem all depends on prev. Lecture. From previous Lecture in Tabulation Code. If you try to understand. Then In tabulation the last row tells me whether is it possible to have subset sums from (0 → Target).

So all we have to do ~~to~~ is iterate over the last row, check if dp[n-1][S1] == true, then automatically S2 = ~~total~~ Sum - S1, and then keep a track of mini = min(mini, abs(S2-S1));

The problem was to simply return the minimum diff when partitioned. Ex [1, 2, 3, 4].

[1, 4] = 5 , [2, 3] = 5

Thus min Diff = 5-5 = 0

Time Comp → O(N * Target).

Space → O(N * Target)

PTO

Code

```
int minimumDiff(vector<int>& arr) {

    int totalSum = 0
    for(auto &p : arr)
        totalSum += p;
    vector<vector<bool>> dp(arr.size, vector<bool>(totalSum+1, false));

    for(int ind = 0; ind < arr.size(); ind++)
        dp[ind][0] = true;
    if(arr[0] <= totalSum)
        dp[0][arr[0]] = true;

    for(int ind = 1; ind < arr.size(); ind++) {
        for(int target = 1; target <= totalSum; target++) {
            bool notpick = dp[ind-1][target];
            bool pick = false.
            if(arr[ind] <= target)
                pick = dp[ind-1][target-arr[ind]];
            dp[ind][target] = pick || notpick;
        }
    }

    int ans = 1e9;
    for(int S1 = 0; S1 <= totalSum; S1++) {
        if(dp[arr.size()-1][S1] == true) {
            int S2 = totalSum - S1;
            ans = min(ans, abs(S2-S1));
        }
    }

    return ans;
}
```

Lecture 14: Count Subsets with Sum K.

The problem is exactly same as Lecture 12, the only difference is that here we have to calculate total no. of subsets equal to target.

I/p arr = [1, 2, 2, 3] ; K = 3.

O/p 3 {[1,2], [1,2], [3]}

Code (Memoization)

```
int f(int ind, int target, vector<int>& arr,
                    vector<vector<int>>& dp)
{
    if (target == 0)
        return 1;
    if (ind == 0)
        return arr[ind] == target;
    if (dp[ind][target] != -1)
        return dp[ind][target];

    int notpick = f(ind-1, target, arr, dp);
    int pick = 0;
    if (arr[ind] <= target)
        pick = f(ind-1, target-arr[ind], arr, dp);

    return dp[ind][target] = pick + notpick;
}
```

Tabulation → (i) Base Case
(ii) Look At Changing Variables/parameters with loops
(iii) Copy Recursion

Code

```
int f(int K ; vector<int>& arr) {

    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(K+1, 0));

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

    if (arr[0] <= k)
        dp[0][arr[0]] = 1;

    for(int i=1 ; i<n ; i++) {
        for(int j = 1; j <= target ; j++) {
            int notpick = dp[i-1][j];
            int pick = 0 ;
            if(arr[i] <= j)
                pick = dp[i-1][j - arr[i]];

            dp[i][j] = pick + notpick;
        }
    }

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

Lecture 15: 0/1 Knapsack

It's a classic knapsack problem. In which There are 'N' items, ~~items~~ Value Array, weight of each ~~Array~~ item and 'W' of bag. We have to calculate maximum value.

V	5	4	8	6
wt	1	2	3	4

N = 4 , W = 5

Ans → 13

This problem is also another DP on subsequences question. where we have the option to pick (or) notpick.

for Memoization

1. Express in terms of indexes - 'n' and 'wt'
2. Do All possible stuff - pick (or) not pick.
3. Take maximum of all.

Declare a DP[][] of N rows & W+1 colums.

Thus, the recurrence

f(ind, W) → Maximum value of items from index 0 to ind, with capacity W.

Base Case

if ind == 0, we can pick it if it's weight is less than (or) equal remaining weight of knapsack. Otherwise we Return 0.

Code

```
int f(ind ind, int W, vector<int>& wt, vector<int>& val,
                    vector<vector<int>>& dp) {

    if (ind == 0) {
        if (wt[ind] <= W)
            return val[ind];
        else
            return 0;
    }
    if (dp[ind][W] != -1) return dp[ind][W];
    int notpick = 0 + f(ind-1, W, wt, val, dp);
    int pick = INT_MIN;
    if (wt[ind] <= W)
        pick = val[ind] + f(ind-1, W-wt[ind], wt, val, dp);

    return dp[ind][W] = max(pick, notpick);
}
```

Time Comp - $O(N \cdot W)$.

Space Comp - $O(N \cdot W) + O(N)$

$O(N \cdot W)$ ↙ dp[][] ↘ recursion stack space.

Tabulation.

- At ind == 0 we are considering the first element, if the capacity of the knapsack is greater than the weight of the first item, we return val[0] as answer
- Use nested loop for traversing over ind & W.
- Copy the Recurrence.

```
int f (int n, int W, vector<int>& val, vector<int>& wt)
{
    vector<vector<int>>dp(n, vector<int>(W+1, 0));

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

    for(int ind=1; ind<n; ind++) {
        for(int cap=0; cap<=W; cap++) {

            int notpick = 0 + dp[ind-1][cap];
            int pick = INT_MIN;
            if(wt[ind]<=cap)
                pick = val[ind] + dp[ind-1][cap-wt[ind]];

            dp[ind][cap]= max(pick, notpick);
        }
    }
    return dp[n-1][W];
}
```

Time Comp - O(NW)
Space Comp - O(NW)

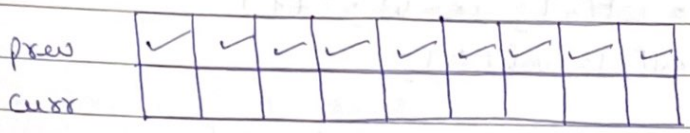
Space optimization.

look At the relation.

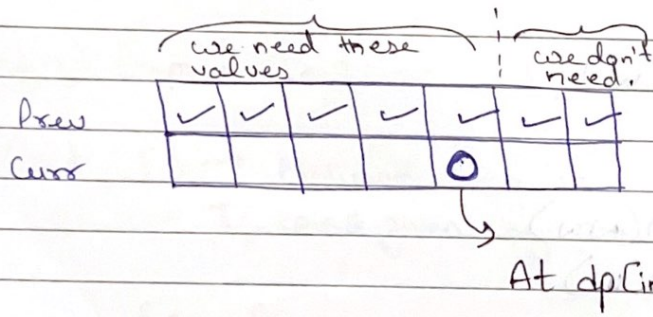
dp[ind][cap] = max(dp[ind-1][cap], dp[ind-1][cap-wt[ind]]);

Like always we only need a previous row to calculate current state. Thus we don't need to store full array. But we can do it in One Row/Array.

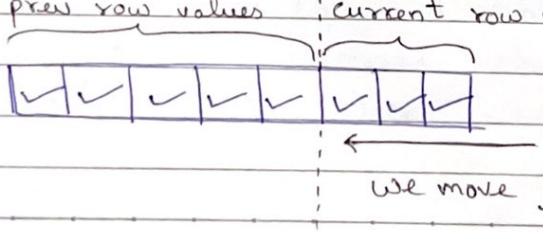
Notice, when we fill in 2-row method (curr & prev).



we initialize first row and using its values we calculate next row.
we can see from recurrence dp[ind-1][cap] & dp[ind-1][cap-wt[ind]].
We can say that if we are at column CAP, we only need values in left side and none in right side. as shown, because (CAP-wt[ind]) will always be less than CAP.



Since we don't value of Right, we start calculating from Right and we will overwrite on prev. ~~to~~ row from Right itself.



Code

```
int f(int n, int W, vector<int> & wt, vector<int> val)
{
    vector<int> prev(W, 0);

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

    for(int ind = 1; ind < n; ind++) {
        for(int cap = W; cap >= 0; cap--) {
            int notpick = 0 + prev[cap];
            int pick = INT_MIN;
            if(wt[ind] <= cap)
                pick = val[ind] + prev[cap - wt[ind]];

            prev[cap] = max(pick, notpick);
        }
    }
    return prev[W];
}
```

Time Comp - O(NW).
Space Comp - O(W).

Lecture 16 - Minimum Coins (Leetcode: Coin Change)

Problem Link: We are given a target Sum and a 'N' sized array which represents unique denominations for coins. We have tell minimum no. of coins required to fulfill target amount and if it's not possible then return -1

Eg:-
arr = [1, 3, 2]
tar = 7

Ans = 3 → [3, 3, 1]

It's a typical Subsequence Problem where the main idea is to pick/not pick. But, here we can pick an element ∞ amount of times. Thus when we can pick an element we won't move to the next element. Instead we would max out on that and then move on.

Step-1) Express In terms Indexes.

f(ind, T) → Minimum coins required to form target T, coins given by arr[0...ind].

Step-2) Base Case.

If we are at ind==0 with a target (T), Then we need to understand, that we first have to check if the Amount And denomination at [0] is divisible COMPLETELY. If yes. Return T/coins[0] otherwise return INT_MAX. Eg.
arr[0]=4, T=12, since T%.arr[0]==0, we return T/arr[0]

Step-3) Try out all possible Stuff.

NOTPICK → The target Sum won't be affected and we will move to the next denomination. Recursive call is made to f(ind-1, T) we add 0 to our answer since we didn't pick it

Pick → We add '1' to our answer, since we are picking it. Subtract ~~arr[ind]~~ arr[ind] from Target Sum and call on the same denomination till target Sum less than the arr[ind] denomination.

Step 4) Return the Minimum of Take And Not take.

Memoization: By creating DP array of size [N][Amount+1].

Time Comp → O(N*T).

Space Comp → O(N*T) x O(N).

```
int MinimumCoins (vector<int>& arr, int T) {
    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(T+1, -1));
    int ans = f(n-1, arr, T, dp);
    if (ans >= 1e8)
        return -1;
    Return ans;
}
```

PTO

```
int f(int ind, vector<int>& arr, int T, vector<vector<int>>&dp)
{
    if (ind == 0) {
        if (T % arr[0] == 0)
            return T / arr[0];
        Else
            return 1e8;
    }
    if (dp[ind][T] != -1)
        return dp[ind][T];

    int NOTPICK = 0 + f(ind-1, arr, T, dp);
    int PICK = 1e8;
    if (T >= arr[ind])
        pick = 1 + (ind, arr, T - arr[ind], dp);

    return dp[ind][T] = min(PICK, NOTPICK);
}
```

Tabulation.

As Always try to first convert the recursive Base Cases i.e
At ind == 0 ; we are considering the first element if T % arr[ind] == 0, we initialize it to T / arr[ind] otherwise 1e8.
Traverse the 2D dp[][] with same recurrence as above.

PTO

Code

```
int f(vector<int>& arr, int T) {
    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(T+1, 0));

    for(int i=0; i<=T; i++) {
        if(i % arr[0] == 0)
            dp[0][i] = i / arr[0];
        else
            dp[0][i] = 1e8;
    }

    for(int i=1; i<n; i++) {
        for(int j=0; j<=T; j++) {

            int notpick = dp[i-1][j];
            int pick = 1e8;
            if(j >= arr[i])
                pick = 1 + dp[i][j-arr[i]];
            dp[i][j] = min(pick, notpick);
        }
    }

    return dp[n][T] >= 1e8 ? -1 : dp[n][T];
}
```

Time → O(N * T)

Space → O(N * T)

Space Optimization.

for Space optimization. use prev in place of dp[ind-1] and during initialization and curr during updation and return prev.

#Lecture 17: Coin Change 2

It's exactly the same problem as last Lecture. But this time we only need to calculate Total no. of ways, to reach target.

N = 3 Arr = {1, 2, 3} , target = 4

Ans → 4
{1, 1, 2}
{1, 3}
{1, 1, 1, 1}
{2, 2}

```
int f(vector<int>& arr, int T) {

    vector<vector<int>> dp(arr.size(), vector<int>(T+1, 0));
    for(int i = 0; i <= T; i++) {
        if (i % arr[0] == 0)
            dp[0][i] = i / arr[0];
    }
    for(int i=1; i<arr.size(); i++) {
        for(int j=0; j<=T; j++) {
            int notpick = dp[i-1][j];
            int pick = 0;
            if(arr[i] <= j)
                pick = dp[i][j-arr[i]];
            dp[i][j] = pick + notpick;
        }
    }
    int n = arr.size();
    return dp[n-1][T];
}
```

Lecture 18 : Unbounded Knapsack.

The Same problem as Knapsack, but here we have unlimited supplies WITH LIMITED Knapsack weight. We have to find Max profit

N = 3 W = 10
wt = { 2, 4, 6 }
val = { 5, 11, 13 }

Ans ⇒ 27
{ 11, 11, 5 }

It's the Same questions Logic as done in previous questions that when we pick the element we don't move to the next, instead we stay there and maximize our profit.

Memoization.

```
int f(vector<int>& wt, vector<int>& val, vector<vector<int>>& dp,
                                          int ind, int W){
    if(ind == 0)
        return (int)(W/wt[0]) * val[0];
    if(dp[ind][W] != -1)
        return dp[ind][W];

    int notpick = 0 + f(wt, val, dp, ind-1, W);
    int pick = 0;
    if(wt[ind] <= W)
        pick = val[ind] + f(wt, val, dp, ind, W-wt[ind]);

    return dp[ind][W] = max(notpick, pick);
}
```

Tabulation.

```
int f(vector<int>& wt, vector<int>& val, int W, int n) {

    vector<vector<int>> dp(n, vector<int>(W+1, 0));

    for(int i=0; i<=W; i++)
        dp[0][i] = (int)(i/wt[0]) * val[0];

    for(int i=1; i<n; i++) {
        for(int j=0; j<=W; j++) {

            int notpick = dp[i-1][j];
            int pick = 0;
            if(wt[i] <= j)
                pick = dp[i][j-wt[i]];
            dp[i][j] = max(pick, notpick);
        }
    }

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

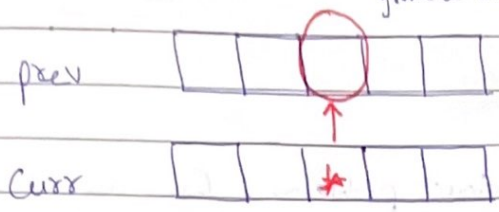
Space Optimization.

Observe At the recurrence.

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

Notice the values required: ~~dp[ind-1][cap dp[i]~~
dp[i-1][j] & dp[i][j-wt[i]], we can say that if we are at a column 'j', we will only require value from red box from prev and values will be from curr. Thus we don't need to store an entire array.

Only the same column from a row Above.




```
int F(vector<int>& wt, vector<int>& val, int W, int n) {

    vector<int> curr(W+1, 0);

    for(int i=0; i<=W; i++)
        curr[i] = ((int) W/wt[0]) * val[0];

    for(int i=1; i<n; i++) {
        for(int j=0; j<=W; j++) {

            int notpick = curr[j];
            int pick = INT_MIN;
            if(wt[i]<=j)
                pick = val[i] + curr[j-wt[i]];
            curr[j] = max(notpick, pick);
        }
    }

    return curr[W];
}
```

Lecture 19 : Rod Cutting Problem

This problem is similiar to previous problems. We are given a rod length 'n' and an array which tells the price of each piece. And each piece size is equal to their index no.

Eg. n = 5 , arr = [2, 5, 7, 8, 10].
Ans = 12

Possible partitions are.
[1, 1, 1, 1, 1] → max_cost (2+2+2+2+2) = 10.
[1, 1, 1, 2] → (2+2+2+5) = 11
⋮
[1, 2, 2] → [2+5+5] = 12

Same Concept of pick/not pick where we pick an element and don't move to the next since we want to max out on that

Base Case :- Try to understand, at ind == 0, it means the rod is being cut of length '1'. That means 'N' (whatever length left before reaching ind==0) Multiplied by arr[0] will give max cost at ind == 0.

Rest it is the same code for Memoization. using a dp[][] of size (n) & (n+1), where n → current index.
(n+1) → rod length left.

Code

Lecture-12: Subset Sum Equals to Target.

Problem

given an array arr[] with N positive integers. We need to find that if there exists a subset / subsequence with sum equal to 'K'. If yes return true / false.

Eg)

arr - [1, 2, 3, 4]

K = 4

Ans) True

Steps to form Recursive Solution.

S-1) Express the Problem in terms of Indexes.

We can ~~int~~ initially say f(n-1, target) which means that we need to find whether there exists a subsequence in the array from index 0 to (n-1), whose sum is equal to target. Generalizing.

f(ind, target) → check whether a subsequence exist in the array from index 0 to ind, whose sum is equal to index.

Base Case:

- if target == 0; it means we have already found a subsequence from the previous step, we can return true.
- if ind == 0; we need to check if arr[ind] == target if yes, return true else return false.

S-2) Try All possible Stuff.

- We will use the 'pick/not pick' technique. i.e.

(a) Include the current element in the subsequence.

We first try to find a subsequence without considering the current element, and make a recursive. f(n-1, target).

(b) Include the element in the subsequence.

Simple, this time we include the element arr[ind] in our subsequence and make recursive call to. f(n-1, target - arr[ind])

Note: only when target $\geq$ arr[ind].

3-3) Return (taken / not taken)

Among the taken / not taken we just need to find one 'true' and then return true for our function

Memoisation

- As usual, we create a ~~dp[n][k+1]~~ dp[n][k+1]. The size of the input array is 'n'; so index always lies between '0' to 'n-1'. The target can take any value between '0' to 'k'
- initialize with '-1'.
- follow as usual.

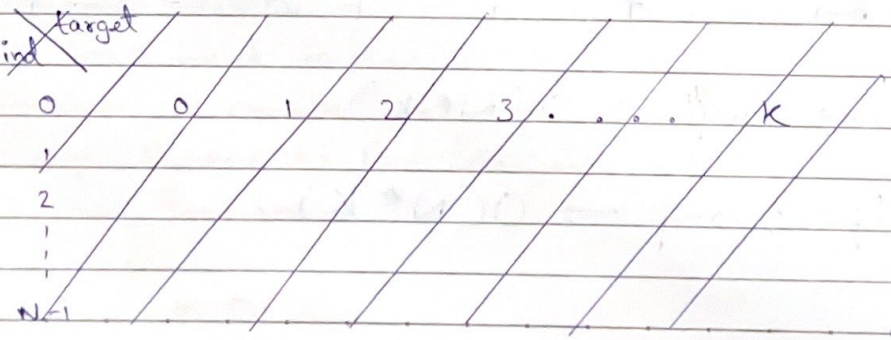
Code

```
bool f(int ind, int target, vector<int> &arr, vector<vector<int>>
                                                    &dp){
    if(ind==0)
        return arr[0]==target;
    if(target==0)
        return true;
    if(dp[ind][target]!=-1)
        return dp[ind][target];

    bool notpick = f(ind-1, target, arr, dp);
    bool pick = false;
    if(arr[ind]<=target)
        pick = f(ind-1, target-arr[ind], arr, dp);

    return dp[ind][target]= pick || notpick.
}
```

Time Comp — $O(N * K)$
Space Comp — $O(N * K) + O(N)$.

Tabulation.

Create the same dp[][] as in memoization. We can set it's type to bool and set all to false.



ind \ target	0	1	2	3	...	K
0	T	F	F	F	...	F
1	F	F	F	F	...	F
2	!	!	!	!		!
!	!	!	!	!		!
!	!	!	!	!		!
N-1	F	F	F	F		F

As always we first initialize the base condition from recursion :-

① If target == 0, ind can take any value from 0 to n-1 Thus we set the first column as true.

② The first row dp[0][] indicates that only the first element of the array is considered, therefore for the target value equal to arr[0], only call with that target will be true.
So set dp[0][arr[0]] = True.

③ Then we will set two nested loops for traversal the dp[][] as discussed previously.

④ Return dp[n-1][K] as answer.

ind \ target	0	1	2 ← if this was arr[0].	—	—	—	K
0	T	F	T	-	-		F
1	T	f	f	-	-	-	f
2	T	f	f	-	-	-	f
!	!	!	!	-	-	-	f
!	!	!	!	-	-	-	f
n-1	T	f	f	-	-	-	f

Time Comp → O(N * K)

Space Comp → O(N * K)

Code

```
bool f(int n, target, vector<int> &arr) {
    vector<vector<bool>> dp(n, vector<bool>(k+1, false));

    for(int i=0; i<n; i++)
        dp[i][0] = true;
    if(arr[0] <= target)
        dp[0][arr[0]] = True;
    for(int i=1; i<ind; i++) {
        for(int j=1; j<=target; j++){
            bool notpick = dp[i-1][j];
            bool pick = false;
            if(arr[i] <= target)
                pick = dp[i-1][target-arr[0]];

            dp[i][j] = pick || not pick;
        }
    }
    return dp[n-1][k];
}.
```

Space Optimization

$$dp[ind][target] = dp[ind-1][target] \;||\; dp[ind-1][target-arr[ind]]$$

Thus to calculate this we only need the prev. row values. Hence we don't need to main an entire array.

Note: Whenever we create a new row (say curr), we need to explicitly set it's first element to be true. acc. to base condition.

Space Comp → O(K).

Lecture 13 :- Partition Array into two ~~ways~~ Arrays to Minimise Sum Difference.

This problem all depends on prev. Lecture. From previous Lecture in Tabulation Code. If you try to understand. Then In tabulation the last row tells me whether is it possible to have subset sums from (0 → Target).

So all we have to do ~~to~~ is iterate over the last row, check if dp[n-1][S1] == true, then automatically S2 = ~~total~~ Sum - S1, and then keep a track of mini = min(mini, abs(S2-S1));

The problem was to simply return the minimum diff when partitioned. Ex [1, 2, 3, 4].

[1, 4] = 5 , [2, 3] = 5

Thus min Diff = 5-5 = 0

Time Comp → O(N * Target).

Space → O(N * Target)

PTO

Code

```
int minimumDiff(vector<int>& arr) {

    int totalSum = 0
    for(auto &p : arr)
        totalSum += p;
    vector<vector<bool>> dp(arr.size, vector<bool>(totalSum+1, false));

    for(int ind = 0; ind < arr.size(); ind++)
        dp[ind][0] = true;
    if(arr[0] <= totalSum)
        dp[0][arr[0]] = true;

    for(int ind = 1; ind < arr.size(); ind++) {
        for(int target = 1; target <= totalSum; target++) {
            bool notpick = dp[ind-1][target];
            bool pick = false.
            if(arr[ind] <= target)
                pick = dp[ind-1][target-arr[ind]];
            dp[ind][target] = pick || notpick;
        }
    }

    int ans = 1e9;
    for(int S1 = 0; S1 <= totalSum; S1++) {
        if(dp[arr.size()-1][S1] == true) {
            int S2 = totalSum - S1;
            ans = min(ans, abs(S2-S1));
        }
    }

    return ans;
}
```

Lecture 14: Count Subsets with Sum K.

The problem is exactly same as Lecture 12, the only difference is that here we have to calculate total no. of subsets equal to target.

I/p arr = [1, 2, 2, 3] ; K = 3.

O/p 3 {[1,2], [1,2], [3]}

Code (Memoization)

```
int f(int ind, int target, vector<int>& arr,
                    vector<vector<int>>& dp)
{
    if (target == 0)
        return 1;
    if (ind == 0)
        return arr[ind] == target;
    if (dp[ind][target] != -1)
        return dp[ind][target];

    int notpick = f(ind-1, target, arr, dp);
    int pick = 0;
    if (arr[ind] <= target)
        pick = f(ind-1, target-arr[ind], arr, dp);

    return dp[ind][target] = pick + notpick;
}
```

Tabulation → (i) Base Case
(ii) Look At Changing Variables/parameters with loops
(iii) Copy Recursion

Code

```
int f(int K ; vector<int>& arr) {

    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(K+1, 0));

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

    if (arr[0] <= k)
        dp[0][arr[0]] = 1;

    for(int i=1 ; i<n ; i++) {
        for(int j = 1; j <= target ; j++) {
            int notpick = dp[i-1][j];
            int pick = 0 ;
            if(arr[i] <= j)
                pick = dp[i-1][j - arr[i]];

            dp[i][j] = pick + notpick;
        }
    }

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

Lecture 15: 0/1 Knapsack

It's a classic knapsack problem. In which There are 'N' items, ~~items~~ Value Array, weight of each ~~Array~~ item and 'W' of bag. We have to calculate maximum value.

V	5	4	8	6
wt	1	2	3	4

N = 4 , W = 5

Ans → 13

This problem is also another DP on subsequences question. where we have the option to pick (or) notpick.

for Memoization

1. Express in terms of indexes - 'n' and 'wt'
2. Do All possible stuff - pick (or) not pick.
3. Take maximum of all.

Declare a DP[][] of N rows & W+1 colums.

Thus, the recurrence

f(ind, W) → Maximum value of items from index 0 to ind, with capacity W.

Base Case

if ind == 0, we can pick it if it's weight is less than (or) equal remaining weight of knapsack. Otherwise we Return 0.

Code

```
int f(ind ind, int W, vector<int>& wt, vector<int>& val,
                    vector<vector<int>>& dp) {

    if (ind == 0) {
        if (wt[ind] <= W)
            return val[ind];
        else
            return 0;
    }
    if (dp[ind][W] != -1) return dp[ind][W];
    int notpick = 0 + f(ind-1, W, wt, val, dp);
    int pick = INT_MIN;
    if (wt[ind] <= W)
        pick = val[ind] + f(ind-1, W-wt[ind], wt, val, dp);

    return dp[ind][W] = max(pick, notpick);
}
```

Time Comp - $O(N \cdot W)$.

Space Comp - $O(N \cdot W) + O(N)$

$O(N \cdot W)$ ↙ dp[][] ↘ recursion stack space.

Tabulation.

- At ind == 0 we are considering the first element, if the capacity of the knapsack is greater than the weight of the first item, we return val[0] as answer
- Use nested loop for traversing over ind & W.
- Copy the Recurrence.

```
int f (int n, int W, vector<int>& val, vector<int>& wt)
{
    vector<vector<int>>dp(n, vector<int>(W+1, 0));

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

    for(int ind=1; ind<n; ind++) {
        for(int cap=0; cap<=W; cap++) {

            int notpick = 0 + dp[ind-1][cap];
            int pick = INT_MIN;
            if(wt[ind]<=cap)
                pick = val[ind] + dp[ind-1][cap-wt[ind]];

            dp[ind][cap]= max(pick, notpick);
        }
    }
    return dp[n-1][W];
}
```

Time Comp - O(NW)
Space Comp - O(NW)

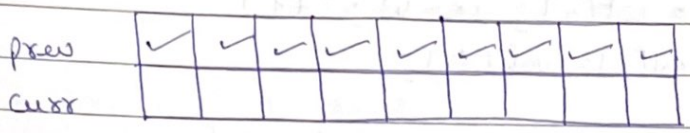
Space optimization.

look At the relation.

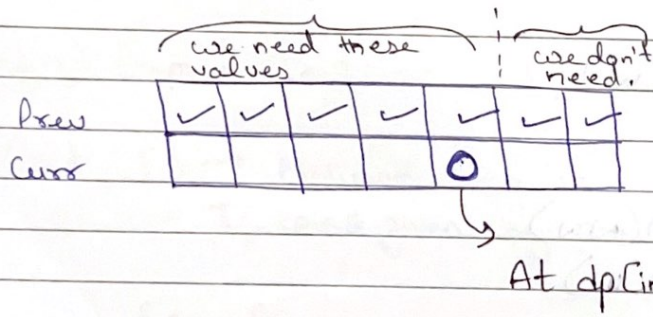
dp[ind][cap] = max(dp[ind-1][cap], dp[ind-1][cap-wt[ind]]);

Like always we only need a previous row to calculate current state. Thus we don't need to store full array. But we can do it in One Row/Array.

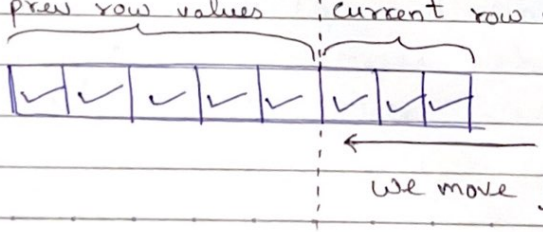
Notice, when we fill in 2-row method (curr & prev).



we initialize first row and using its values we calculate next row.
we can see from recurrence dp[ind-1][cap] & dp[ind-1][cap-wt[ind]].
We can say that if we are at column CAP, we only need values in left side and none in right side. as shown, because (CAP-wt[ind]) will always be less than CAP.



Since we don't value of Right, we start calculating from Right and we will overwrite on prev. ~~to~~ row from Right itself.



Code

```
int f(int n, int W, vector<int> & wt, vector<int> val)
{
    vector<int> prev(W, 0);

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

    for(int ind = 1; ind < n; ind++) {
        for(int cap = W; cap >= 0; cap--) {
            int notpick = 0 + prev[cap];
            int pick = INT_MIN;
            if(wt[ind] <= cap)
                pick = val[ind] + prev[cap - wt[ind]];

            prev[cap] = max(pick, notpick);
        }
    }
    return prev[W];
}
```

Time Comp - O(NW).
Space Comp - O(W).

Lecture 16 - Minimum Coins (Leetcode: Coin Change)

Problem Link: We are given a target Sum and a 'N' sized array which represents unique denominations for coins. We have tell minimum no. of coins required to fulfill target amount and if it's not possible then return -1

Eg:-
arr = [1, 3, 2]
tar = 7

Ans = 3 → [3, 3, 1]

It's a typical Subsequence Problem where the main idea is to pick/not pick. But, here we can pick an element ∞ amount of times. Thus when we can pick an element we won't move to the next element. Instead we would max out on that and then move on.

Step-1) Express In terms Indexes.

f(ind, T) → Minimum coins required to form target T, coins given by arr[0...ind].

Step-2) Base Case.

If we are at ind==0 with a target (T), Then we need to understand, that we first have to check if the Amount And denomination at [0] is divisible COMPLETELY. If yes. Return T/coins[0] otherwise return INT_MAX. Eg.
arr[0]=4, T=12, since T%.arr[0]==0, we return T/arr[0]

Step-3) Try out all possible Stuff.

NOTPICK → The target Sum won't be affected and we will move to the next denomination. Recursive call is made to f(ind-1, T) we add 0 to our answer since we didn't pick it

Pick → We add '1' to our answer, since we are picking it. Subtract ~~arr[ind]~~ arr[ind] from Target Sum and call on the same denomination till target Sum less than the arr[ind] denomination.

Step 4) Return the Minimum of Take And Not take.

Memoization: By creating DP array of size [N][Amount+1].

Time Comp → O(N*T).

Space Comp → O(N*T) x O(N).

```
int MinimumCoins (vector<int>& arr, int T) {
    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(T+1, -1));
    int ans = f(n-1, arr, T, dp);
    if (ans >= 1e8)
        return -1;
    Return ans;
}
```

PTO

```
int f(int ind, vector<int>& arr, int T, vector<vector<int>>&dp)
{
    if (ind == 0) {
        if (T % arr[0] == 0)
            return T / arr[0];
        Else
            return 1e8;
    }
    if (dp[ind][T] != -1)
        return dp[ind][T];

    int NOTPICK = 0 + f(ind-1, arr, T, dp);
    int PICK = 1e8;
    if (T >= arr[ind])
        pick = 1 + (ind, arr, T - arr[ind], dp);

    return dp[ind][T] = min(PICK, NOTPICK);
}
```

Tabulation.

As Always try to first convert the recursive Base Cases i.e
At ind == 0 ; we are considering the first element if T % arr[ind] == 0, we initialize it to T / arr[ind] otherwise 1e8.
Traverse the 2D dp[][] with same recurrence as above.

PTO

Code

```
int f(vector<int>& arr, int T) {
    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(T+1, 0));

    for(int i=0; i<=T; i++) {
        if(i % arr[0] == 0)
            dp[0][i] = i / arr[0];
        else
            dp[0][i] = 1e8;
    }

    for(int i=1; i<n; i++) {
        for(int j=0; j<=T; j++) {

            int notpick = dp[i-1][j];
            int pick = 1e8;
            if(j >= arr[i])
                pick = 1 + dp[i][j-arr[i]];
            dp[i][j] = min(pick, notpick);
        }
    }

    return dp[n][T] >= 1e8 ? -1 : dp[n][T];
}
```

Time → O(N * T)

Space → O(N * T)

Space Optimization.

for Space optimization. use prev in place of dp[ind-1] and during initialization and curr during updation and return prev.

#Lecture 17: Coin Change 2

It's exactly the same problem as last Lecture. But this time we only need to calculate Total no. of ways, to reach target.

N = 3 Arr = {1, 2, 3} , target = 4

Ans → 4
{1, 1, 2}
{1, 3}
{1, 1, 1, 1}
{2, 2}

```
int f(vector<int>& arr, int T) {

    vector<vector<int>> dp(arr.size(), vector<int>(T+1, 0));
    for(int i = 0; i <= T; i++) {
        if (i % arr[0] == 0)
            dp[0][i] = i / arr[0];
    }
    for(int i=1; i<arr.size(); i++) {
        for(int j=0; j<=T; j++) {
            int notpick = dp[i-1][j];
            int pick = 0;
            if(arr[i] <= j)
                pick = dp[i][j-arr[i]];
            dp[i][j] = pick + notpick;
        }
    }
    int n = arr.size();
    return dp[n-1][T];
}
```

Lecture 18 : Unbounded Knapsack.

The Same problem as Knapsack, but here we have unlimited supplies WITH LIMITED Knapsack weight. We have to find Max profit

N = 3 W = 10
wt = { 2, 4, 6 }
val = { 5, 11, 13 }

Ans ⇒ 27
{ 11, 11, 5 }

It's the Same questions Logic as done in previous questions that when we pick the element we don't move to the next, instead we stay there and maximize our profit.

Memoization.

```
int f(vector<int>& wt, vector<int>& val, vector<vector<int>>& dp,
                                          int ind, int W){
    if(ind == 0)
        return (int)(W/wt[0]) * val[0];
    if(dp[ind][W] != -1)
        return dp[ind][W];

    int notpick = 0 + f(wt, val, dp, ind-1, W);
    int pick = 0;
    if(wt[ind] <= W)
        pick = val[ind] + f(wt, val, dp, ind, W-wt[ind]);

    return dp[ind][W] = max(notpick, pick);
}
```

Tabulation.

```
int f(vector<int>& wt, vector<int>& val, int W, int n) {

    vector<vector<int>> dp(n, vector<int>(W+1, 0));

    for(int i=0; i<=W; i++)
        dp[0][i] = (int)(i/wt[0]) * val[0];

    for(int i=1; i<n; i++) {
        for(int j=0; j<=W; j++) {

            int notpick = dp[i-1][j];
            int pick = 0;
            if(wt[i] <= j)
                pick = dp[i][j-wt[i]];
            dp[i][j] = max(pick, notpick);
        }
    }

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

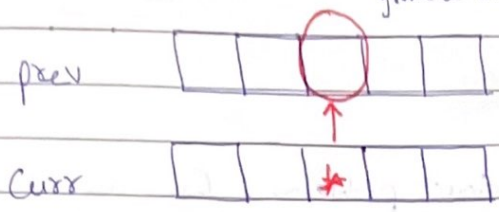
Space Optimization.

Observe At the recurrence.

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

Notice the values required: ~~dp[ind-1][cap dp[i]~~
dp[i-1][j] & dp[i][j-wt[i]], we can say that if we are at a column 'j', we will only require value from red box from prev and values will be from curr. Thus we don't need to store an entire array.

Only the same column from a row Above.




```
int F(vector<int>& wt, vector<int>& val, int W, int n) {

    vector<int> curr(W+1, 0);

    for(int i=0; i<=W; i++)
        curr[i] = ((int) W/wt[0]) * val[0];

    for(int i=1; i<n; i++) {
        for(int j=0; j<=W; j++) {

            int notpick = curr[j];
            int pick = INT_MIN;
            if(wt[i]<=j)
                pick = val[i] + curr[j-wt[i]];
            curr[j] = max(notpick, pick);
        }
    }

    return curr[W];
}
```

Lecture 19 : Rod Cutting Problem

This problem is similiar to previous problems. We are given a rod length 'n' and an array which tells the price of each piece. And each piece size is equal to their index no.

Eg. n = 5 , arr = [2, 5, 7, 8, 10].
Ans = 12

Possible partitions are.
[1, 1, 1, 1, 1] → max_cost (2+2+2+2+2) = 10.
[1, 1, 1, 2] → (2+2+2+5) = 11
⋮
[1, 2, 2] → [2+5+5] = 12

Same Concept of pick/not pick where we pick an element and don't move to the next since we want to max out on that

Base Case :- Try to understand, at ind == 0, it means the rod is being cut of length '1'. That means 'N' (whatever length left before reaching ind==0) Multiplied by arr[0] will give max cost at ind == 0.

Rest it is the same code for Memoization. using a dp[][] of size (n) & (n+1), where n → current index.
(n+1) → rod length left.

Code

Lecture 20: Longest Common Subsequence.

The states that we would be given 2 ~~dp~~ strings and we count no. of characters which are ~~comm~~ common and in a subsequence
Ex.

a b c d e f a b e f

Ans ⇒ 4 ("abef")

Now try to understand that like problems in DP on subsequences we performed pick/not pick. Similiary in DP on strings we perform match/not match. So we compare each character and if it is a match, +1 to ans and reduce both string pointers by -1. (Since we are starting from end of string).
Now, if it doesn't match, try to under. by the following example

c a | a c
↑ ↑

§ Since currently they are not matching and if reduce ~~any~~ both pointers then we may loose ~~on~~ future ans.
Thus, we make '2' recursive calls, where we reduce one pointer in both string in different recursive calls. So as to cover all possibilities.

Base Case :- Since we are reducing and comparing strings, till ind == 0, the base case would be after that case i.e. ind < 0, this signifies string is over and we return 0; because there is no string left to compare.

Create a dp[][] of size that is equal to size of string1 & string2

Code

```
int f(int i, int j, string &s, string &t,
                    vector<vector<int>> &dp) {

    if(i<0 || j<0)
        return 0;
    if(dp[i][j] != -1) return dp[i][j];

    if(s[i] == t[j])
        return 1 + f(i-1, j-1, s, t, dp);

    return 0 + max(f(i, j-1, s, t, dp), f(i-1, j, s, t, dp));
}
```

make these dp[i][j] =

main recurrence when s[i] != t[j]

Time — O(N x M).

Space — O(N x M) + O(N+M)

A.S.S - Auxilary Stack Space

Try to make recursion and notice why (N+M).

Tabulation

Now, Since we know hoco make tabulation, Copy Base case. But in this case, base case is at (-1) which in tabulation form is not possible thus we make a right shift at every index. And base case change and become

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

Simply, intialize the dp[][] as equal to 0. Copy recurrence and proceed as ~~follows~~ always.

Code

```
int f(string &s, string &t) {

    int n = s.size(), m = t.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++){
            int notsame = 0 + max(dp[i-1][j], dp[i][j-1]);
            int same = 0
            if(s[i-1] == t[j-1])
                same = 1 + dp[i-1][j-1];
            dp[i][j] = max(notsame, same);
        }
    }

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

Time → O(NM)
Space → O(NM)

Lecture 21 Longest Common Substring.

Now, in the case of substring, it has to be contiguous
i.e ABCD | ABZD.
Ans 2 ('AB')

for this we can use the concept of longest common subs... but the part of not matching.

$$\max(f(i-1, j), f(i, j-1))$$

cannot be used because this function creates all possibilities i.e subsequences too!
Thus for not matching cases make everything = 0.
Lets Do a Dry Run.

ABCD | ABZD

i ↓ \ j →	0	1	2	3	4	
		A	B	C	D	
0	0	0	0	0	0	Base Case
1	A	1	0	0	0	
2	B	0	2	0	0	
3	Z	0	0	0	0	
4	D	0	0	0	1	
	Base Case ↑					

Take the maximum of all nos.

Code

```
int f(string &s, string &t){

    int n = s.size();
    int m = t.size();

    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));
    int ans = 0;
    for(int i=0; i<=n; i++){
        for(int j=1; j<=m; j++){
            if(dp[s[i-1]==t[j-1]){
            dp[i][j] ans = 1+ dp[i-1][j-1];
              ans = max(ans, dp[i][j]);
            }
        }
    }
    return ans;
}
```

Lecture 22: Longest Palindromic Subsequences

~~Trees only on~~

The brute force method is to take all Subsequences store them, & check which are palindrome and print the longest one.

~~The op~~ The optimized way is to ~~tell~~ notice that string S = 'b b b a b b a b'

LPS = b b b a b b b.

Notice if we take another string t = b a b b a b b b i.e the reverse of 'S'. And now try to ~~make~~ find the longest Common Subsequence.

Notice when you find Common Subsequence of 2 strings which are reverse of each other. It will eventually be a palindrome.

Lecture 23: Minimum ~~Steps~~ Insertion Steps to make a String Palindrome.

Now, first try to understand that we can make any ~~str~~ string into a palindrome. How?

Suppose we have,

a a b c d.

The most naive Soln is to take another string same as this, reverse and add it to original.

i.e,

a a b c d d c b a a. it is now a palindrome.

But this means the no. of insertions is 'n' (length of Str) which is not optimal.

So how approach such question. Simple Think. of it as, we first calculate the longest palindrome subsequence, take them out and then the rest left we will just club them together and reverse it and add wherever necessary. Eg.

'aabcd'

here 'aa' is a palindrom.

'bcd', take another dcb and add to beginning i.e

dcbaabcd.

Since Quest^{n} doesn't want string, it only requires no. of char's added. Simply calculate longest palindrome and ~~it~~ subtract from total length.

Shortest Common Supersequence : Lecture 24

The problem states that we are given 2 strings and we need to create another string which would consist of both in it. But we have to tell the SHORTEST.

A = "BRUTE" , B = "GROOT"

Ans ⇒ C = "BGRUOOTE"
This word has both 'BRUTE' & 'GROOT' with minimum length i.e 8

Now first try to understand the logic, observe if calculate Longest Common Subsequence in both A & B. i.e 'RT', then the minimum length would always be.

$$n + m - LCS$$

- n : length of string 1
- m : length of string 2
- LCS : Common char.

Now, as we did in print LCS, we take the dp Table. Start from the bottom most cell make our way up we check for the following condition.

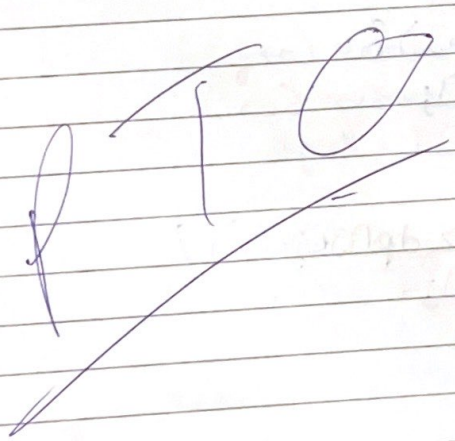
```
if (S[i-1] = t[j-1]).  // That means it's a common
    ans += S[i-1];      char; add to the ans and
    i--, j--;           move diagnoally up
```

Now, we check if the element up or Left one larger., if up is greater we go up and add 'S' char. to the ans., and if we go left then we add 't'

```
if (dp[i-1][j] > dp[i][j-1])
        ans += s[i-1];
        i--;
else
        ans += t[j-1];
        j--;
```

We do this process till either one of them reaches '0' and then break and if one reach '0', there has to be some characters left in other, we simply add those characters to ~~left~~ ans.

~~int~~ ~~string~~ f(~~string~~ s, ~~string~~ t) {.



```
string f(string s, string t){
    int n=s.size(), m=t.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++){
            int notmatch= max(dp[i-1][j], dp[i][j-1]);
            int match = 0;
            if(s[i-1] == t[j-1])
                match = 1+dp[i-1][j-1];
            dp[i][j]= max(match, notmatch);
        }
    }
    string ans;
    int i=n, j=m;
    while(i>0 && j>0){
        if(s[i-1] == t[j-1]){
            ans += s[i-1];
            i--, j--;
        }
        else if(dp[i-1][j] > dp[i][j-1]){
            ans += s[i-1];
            i--;
        }
        else{
            ans += t[j-1];
            j--;
        }
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

Distinct Subsequences. : Lecture 25

It is another standard pick/not pick problem. (or) as DP in strings we say match/notmatch....

All we have to do is count ~~str~~ how many times string 't' occures in string S. For Recurrence our parameter will be (i, j) that are indexes of string S & t, starting at end.

Since it is a counting problem, we return the ~~subse~~ summation of pick/notpick.

#Memoization

```
int f(int i, int j, string &s, string &t,
                    vector<vector<int>>&dp){
if(dp[i][j]!=-1) return dp[i][j]
                if (j<0) return 1;
                if (i<0) return 0;
                int notpick = f(i-1, j, s, t, dp);
                int pick = 0
                if (s[i]==t[j])
                    pick = f(i-1, j-1, s, t, dp);

                return dp[i][j] = pick + notpick;
}
```

For base case try to understand,
S = bagbbag , t = bag.

if i<0, then that means we traversed whole 'S' and couldn't find a match.

Whereas if j<0, then that means we found whole of 't', because we decrease 'j' only when it's a match

Tabulation.

Same Methodology, convert base cases → Copy recurrence

```
int Solve (String s, String t) {
    int n = s.size();
    int m = t.size();
    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));

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

    for (int i=1; i<=n; i++) {
        for(int j=1; j<=m; j++) {
            int notpick = dp[i-1][j];
            int pick = 0;
            if (s[i-1] == t[j-1])
                pick = dp[i-1][j-1];
            dp[i][j] = pick + notpick;
        }
    }
    return dp[n][m];
}
```

Understanding base case, like we did in LCS problem, shifting everything to the right, because we can't write case for (i<0) & (j<0) in tabulation format.

Lecture 26 : Edit Distance.

In this Question we are given 3 ways to edit a 'S' to make it into string 't'. That are →

① Insert
② Remove
③ Replace.

We have solved similiar questions before, the underlying principle is the same. i.e match / notmatch.

S = "horse" , t = "ros"

S-1) → replace 'h' → 'r'
S-2) → remove 'r' at S[2]
S-3) → remove 'e' at S[4]

and we got our string equal in minimum steps possible

We know the recurrence will f(i, j) where i & j are indexes of S & t resp., Starting from end we compare each character and have the following choices

Match	Not Match
If it is a match, we don't need to perform any editing and simply decrease the size of both strings and move on.	① Insert → we do an imaginary insertion, at a place ahead of curr char, which is obviously equal to t[j], and Thus, we stay at same place in String 'S' and decrease 't' and continue comparision. ② Remove → Simply Decrease string 'S' and keep 't' same.

③ Replace → If we are gonna replace a char in S, then it will obv be same as t, thus we can decrease both strings by 1, since we made them equal.

* Base Case

Try Understand, the Base Case would arise when either one of them is exhausted, and try to understand when either one of them is exhausted we just have to return the leftover of ~~this~~ other string length for complete transformation.

And Since we want minimum steps required for transformation, we simply take minimum of all 4 action.

- → Match
- → Insert
- → Remove
- → Replace

Time Complexity → O(N × M).

Space Complexity → O(N × M) + O(N + M)

O(N × M) → dp arr[][]

O(N + M) → Auxiliary stack.

Code (Memoization).

```
int f(string &s, string &t, int i, int j, vector<vector<int>> &dp){
    if(j<0)
        return i+1;
    if(i<0)
        return j+1;
    if(dp[i][j]!=-1)
        return dp[i][j];

    int match = 1e4;
    if(s[i]==t[j]);
        match = f(s,t,i-1,j-1,dp);
    int insert = 1+ f(s,t,i,j-1,dp);
    int remove = 1+ f(s,t,i-1,j,dp);
    int replace = 1+ f(s,t,i-1,j-1,dp);

    dp[i][j] = min(min(match,insert), min(replace, remove));
    return dp[i][j];
}
```

Tabulation.

Same Rules + Shifting Right by one., try understanding bases cases that when i==0 → we have values of j from 0 → m & similiarly for j==0.

```
int Solve (String S, String t){
    int n = S.size(), m = t.size();
    vector<vector<int>> dp(n+1, vector<int>(m+1, 0));

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

    for(int i=1 ; i<=n ; i++) {
        for(int j=1 ; j<=m; j++) {
            int match = 1e4;
            if(S[i-1] == t[j-1])
                match = dp[i-1][j-1];
            int insert = 1+ dp[i][j-1];
            int remove = 1+ dp[i-1][j];
            int replace = 1+ dp[i-1][j-1];

            dp[i][j]= min(min(match, insert),
                          min(replace, remove));
        }
    }
    return dp[n][m];
}
```

Time Comp → O(N×M)

Space Comp → O(N×M)

#Lecture 27: Wildcard Matching

This is yet another match / not match problem with modification, here we have two Special Char.

① "?" → this means that we can match it with any other char of other string.

② "*" → This means we can match any Subsequence with other string. Even NULL char.

We are given a string 'S' and after checking acc to ques we have to tell if it is equal to string 't'. Return T/F.

S = "AB?D" , t = "ABCD"

Then we follow the same pattern as follows, if (S[i] == t[j]) call the function & decrease the value of i & j, where i = n-1 & j = m-1, But with an extra condition as, if S[i] == '?' still it's a match and we do the same function call.

Now if we step on a '*', things get a bit tricky.

S = "AB*EF" t = "ABCDEF"

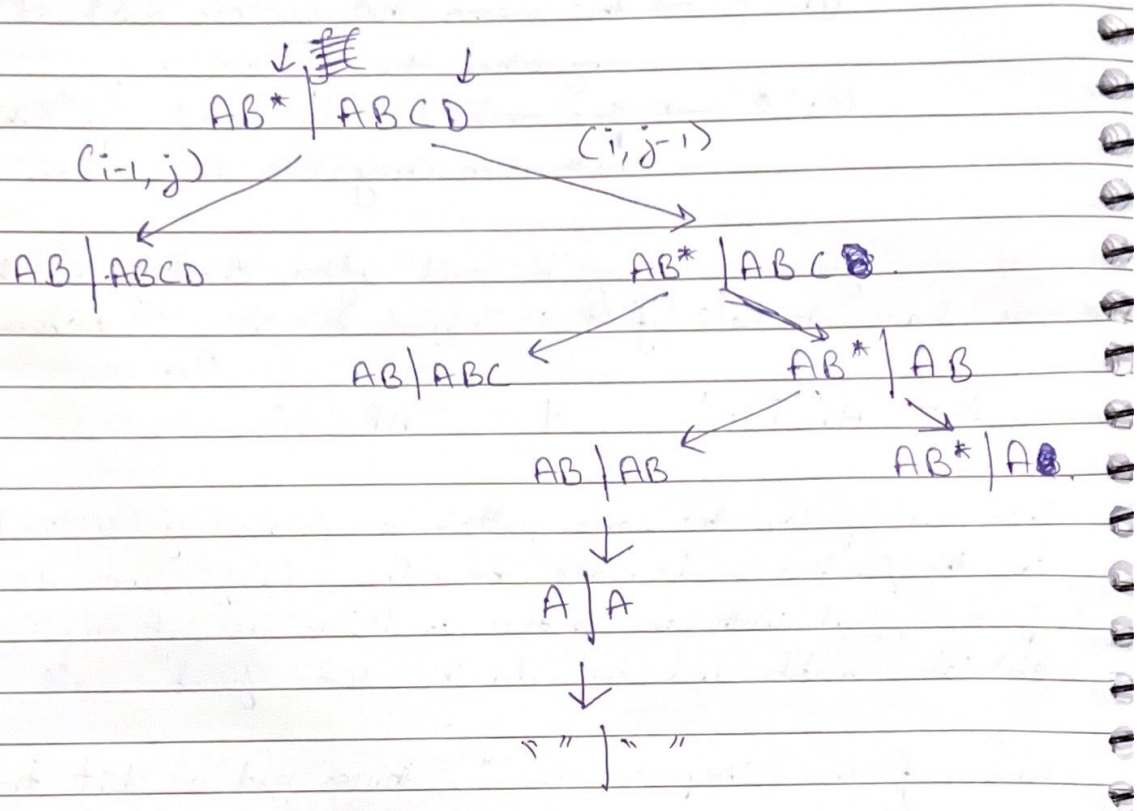
Since we start from end, we'll have a clear match on E & F without any problem.

But after that we don't know how much should * Contain So as to make S == t. for that try out all possible Subsequences. Try to understand, the recurrence f(i-1, j) & f(i, j-1); This simply means take and not take with stoppage at '*'.

We can have all possible states with these 2 recurrences. And if either of them returns True. We return true.

Let's ~~Do~~ Dry Run it

S = "AB*"
t = "ABCD"



This Recursive calls, helps us to calculate all possibilities.

Now, Let's try to understand Base Cases.

The base case logic, would be same as previous question, we see which and when a string exhausts.

* if both Strings exhaust → Match found return True.
* if string 'S' exhaust → Their is still 't' remaining to be match with NULL string 'S'. Return false.

PTO

* If String 't' exhausts → if only string S remains, try to understand observe that the remaining characters have to be '*' to ~~xxx~~ treat them as NULL. If any char exists Return False, Else Return true.

Convert the code into Memoization dp[n][m];

```
bool f(int i, int j, string &s, string &t, vector<vector<int>> &dp){
    if(i<0 && j<0)
        return true;
    if(i<0 && j>=0)
        return false;
    if(i>=0 && j<0){
        for(int x=0; x<=i; x++){
            if(s[x] != '*')
                return false;
        }
        return true;
    }
    if(dp[i][j] != -1)
        return dp[i][j];
    if(s[i]==t[j] || s[i]=='?')
        return dp[i][j]= f(i-1,j-1,s,t,dp);
    if(s[i]=='*')
        return dp[i][j]= f(i-1,j,s,t,dp) || f(i,j-1,s,t,dp);
    return dp[i][j]= false;
}
```

Time Comp — O(N x M)
Space Comp — O(N x M) + O(N+M)

Tabulation

Same logic of shifting towards Right, Observe and think carefully about base cases

```
bool Solve (String S, String t){
    int n= s.size(), m= t.size();
    vector<vector<bool>> dp(n+1, vector<bool>(m+1, false));
    dp[0][0] = true
    for(int i=1; i<=n; i++){
        for(int x=1; x<=i, x++){
            if (s[x-1] != '*'){
                flag= false;
                break;
            }
        }
        dp[i][0] = flag == false ? false : true;
    }
    for (int i=1; i<=n; i++){
        for(int j=1; j<=m; j++){
            if (s[i-1] == t[j-1] || s[i-1] == '?')
                dp[i][j] = dp[i-1][j-1];
            else if (s[i-1] == '*')
                dp[i][j] = dp[i-1][j] || dp[i][j-1];
            else
                dp[i][j] = false;
        }
    }
    return dp[n][m]
}
```

Time — O(N x M)
Space — O(N x M)

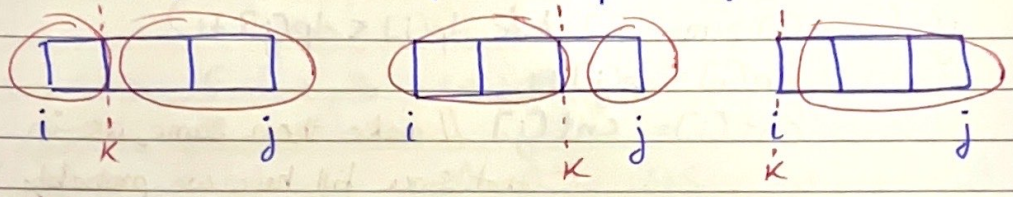
Lecture 37 : Partition DP

① Matrix Chain Multiplication.

Partion DP is used solve question of a similiar pattern, Somewhere where we have to use partition to ~~sto~~ solve part (1) & then part (2)

Eg:- We are given an arr[], starting pt → i
ending pt → j

we can make a partition (K) anywhere then solve the two parts and compute as per requirement. as shown



for MCM → Here we have to tell the minimum cost to Multiple N Matrix... we would be given an (N+1) sized [], which would lead (N) size matrix Eg.

Arr[] → | 10 | 20 | 30 | 40 |

This means Mat A → (10×20)
Mat B → (20×30)
Mat C → (30×40)

Now if we cal (AB)C → Then the cost would

(10×20×30) + (20×30×40)
⇒ 6000 + 24000
⇒ 30000 operations

And if we calc A(BC) ⇒ Then.

$(AB) \times C$ ⇒ $(10 \times 20 \times 30) + (10 \times 30 \times 40)$

$[]_{10\times30}$ $[]_{30\times40}$ ⇒ 6000 + 12000

⇒ 18000

for $A \times (BC)$ ⇒ $(20 \times 30 \times 40) + (10 \times 20 \times 40)$

$[]_{10\times20}$ $[]_{20\times40}$ ⇒ 24000 + 8000

⇒ 32000

Thus Multiplying (AB)C is much more optimal.

Rules

① Start with entire block/Array

* f(i, j)
  start ↘ end

② Try all partitions

* Run a loop to try all partitions.

③ Return the best possible 2 partitions.

PTO

Think in regard with array.

[10, 20, 30, 40, 50]
A B C D
i j

we can access a matrix dimension by (i) & (i-1)
Start → i , end → j

$f(i,j) = f(1,4)$ → return the minimum multiplication to multiply mat 1 → mat 4
Index

Base Case

Try imagining we will hit the base case at $(i == j)$; at that point there's calculation possible thus return 0;

Try possible Partition / Calculations

We ~~took~~ loop K → i − (j − 1)
0 1 2 3 4
i j

[10, 20, 30, 40, 50]
A B C D

K = 1 → f(i, K) , f(K+1, j)

K = 2 → f(1, 2) , f(3, 4)

K = 3 → f(1, 3) , f(4, 4)

Understand for f(1,1) , f(2, 4)

A (10×20) B C D (20×30) × (30×40) × (40×50)

(20×40) × (40×50)

(20×50)

And this same (BC)D == B(CD), look

(20×30) × (30×40) × (40×50)

(20×50) × (30×50)

(20×50)

Same

No. of operations for (A) × (BCD) will be
(10×20) (20×50)

→ (10×20×50) + the rest calculation from (BCD) done by recursion.

A[i-1] * A[k] * A[j]

This is the pattern because (i-1) → 0th index
(k) → 1st index
(j) → 4th index

For Memoisation → [2D] array..

```
int f(int i, int j, vector<int>& arr, vector<vector<int>>
                                        & dp) {

    if (i==j)
        return 0;
    if (dp[i][j] != -1)
        return dp[i][j];

    int mini = 1e9;
    for(int k=i ; k<j ; k++) {

        int cost = arr[i-1] * arr[k] * arr[j]
                 + f(i, k, arr, dp) + f(k+1, j, arr, dp);

        mini = min(cost, mini);
    }
    return dp[i][j] = mini;
}
```

Time Complexity → $O(N*N)*N \sim O(N^3)$

Space Complexity → $O(N^2) + O(N)$
↳ ASS.

Tabulation

The Same Rules As always

1. Copy Base Case
2. Write Down Changing parameters * i & j traverse in opp. order as in Memoisation.

* $i \Rightarrow n-1 \longrightarrow 1$

* $j \Rightarrow i+1 \longrightarrow 1$

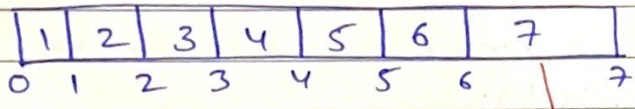
Try to understand that 'j' can never be to the left of i, it's always gonna be towards right.

```
int f(vector<int>& arr) {
    int n = arr.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));

    for(int i = 1; i < N; i++)
        dp[i][i] = 0; // Base Case

    for(int i = n-1; i >= 1; i--) {
        for(int j = i+1; j < N; j++) {
            int mini = 1e9;
            for(int k = i; k < j; k++) {
                int cost = arr[i-1] * arr[k] * arr[j]
                           + dp[i][k] + dp[k+1][j];

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

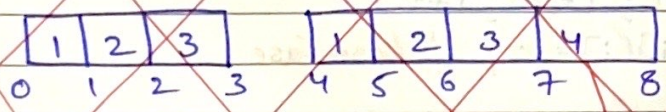
Lecture 38: Minimum cost to Cut the Stick

ATQ, we will be given a cuts[], cuts[i] represend where we want to make a cut, and the to make that cut it would equal to the remainging length Eg.

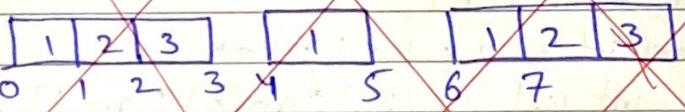
cuts[] = 3, 5, 1, 4.



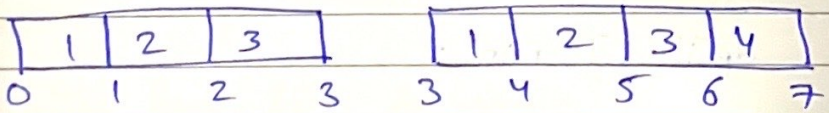
first cut at 3 → cost = 7.



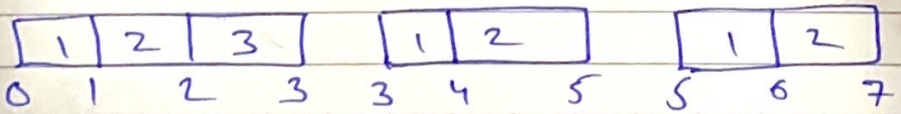
Then at 5 ⇒ cost = 7 + 4.



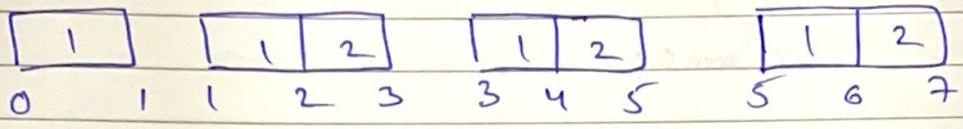
cut at 3 ⇒ cost = 7.



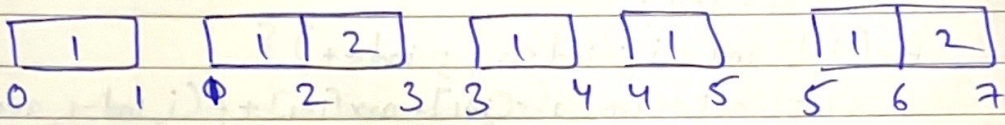
cut at 5 ⇒ cost = 7 + 4.



cut at 1 ⇒ cost ⇒ 7+4+3.



at 4 ⇒ cost ⇒ 7+4+3+2 ⇒ 16



length of Stick.

we will add '0' and 'n' to calculate the cost because we the length of the curr cut, to find that we need ~~ten~~ length

Add '0' & 'n' and sort the cuts[]

Using Partition DP to calc

↳ cuts[j+1] - cuts[i-1] + f(i, ind-1) + f(ind+1, j)

Clearly Base Case is at i > j → can't do anything return 0.

Use 2D[][] for Memoization.

(1). m = size of cuts[].

```
int f(int i, int j, vector<int>& arr, vector<vector<int>>& dp){

    if (i>j)
        return 0;

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

    int mini = 1e9;
    for(int ind = i; ind<=j; ind++){
        int cost = abs(arr[j+1] - arr[i-1] + f(i, ind-1, arr, dp)
                   + f(ind+1, j, arr, dp);
        mini = min(mini, cost);
    }
    return dp[i][j] = mini;
}.
```

TC → $O(M^2 \times M) \approx O(M^3)$; SC → $O(M^2) + O(M)$ (ASS)

Tabulation

* Copy base Case
* Traverse the changing parameters in reverse fashion
  - i ⇒ m → 1
  - j ⇒ 1 → m
* Copy Recurrence

TC → $O(M^3)$
SC → $O(M^2)$

```
int f (vector<int>& cuts) {
    int m = cuts.size();
    cuts.push_back(0);
    cuts.push_back(m);
    vector<vector<int>> dp(m+2, vector<int>(m+2,0));

    for(int i= m ; i >= 1 ; i--) {
        for(int j=1 ; j<=m ; j++) {
            if(i>j)
                continue;
            for(int ind=i ; ind<=j ; ind++) {
                cost = abs(cuts[j+1] - cuts[i-1] +
                        dp[i][ind-1] + dp[ind+1][j];
                cost=
                mini = min(mini, cost);
            }
            dp[i][j]=mini;
        }
    }
    return dp[1][m];
}
```