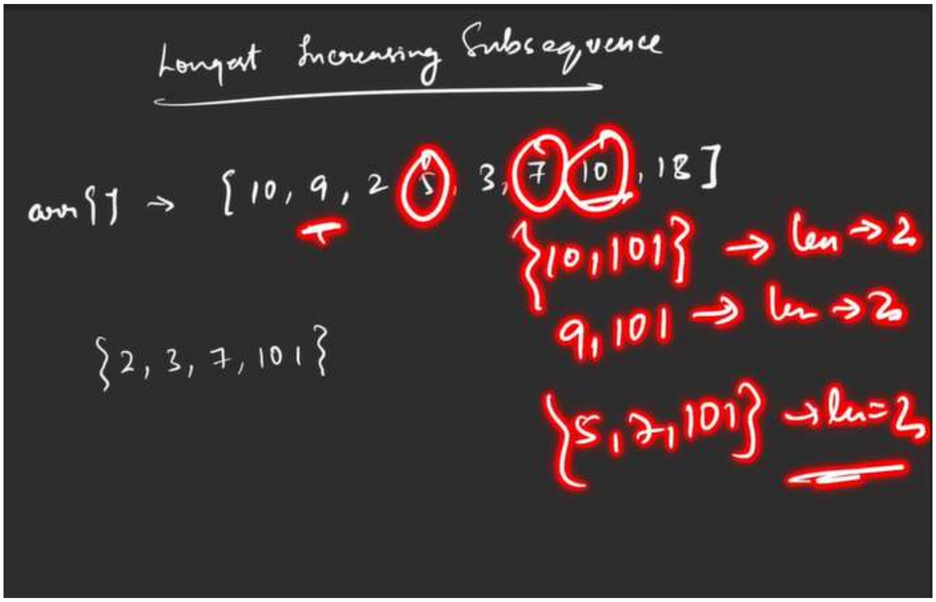
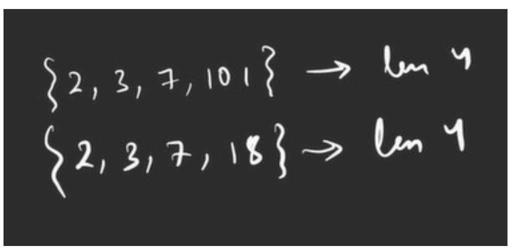
Dynamic Programming

13 September 2023 11:00 AM

Longest Common Subsequence

Contiguous sequence of an array is called sub-sequence





Return length of longest increasing subsequence.

We will follow the approach of (pick / not pick)

Brute force O(2^n)
We can print all the subsequences using power set
Check for increasing
Store the longest one and return it

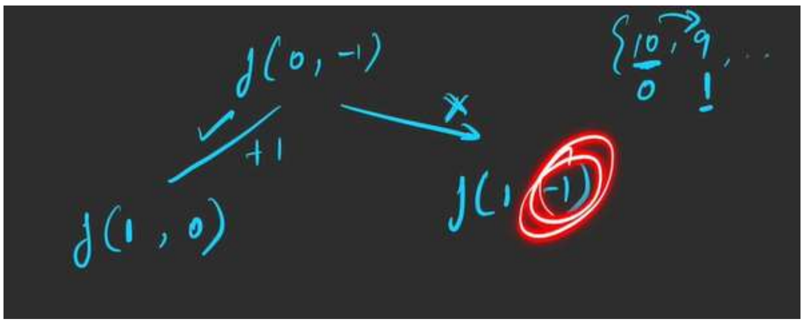
Optimised approach
We can write a recurrence relation
Express everything in terms of index
Explore all possiblities
Take max length of (pick,not pick)

We take 10 but to take or not take next element in subsequence we need to have store of previous index
So we take (index,prev index)

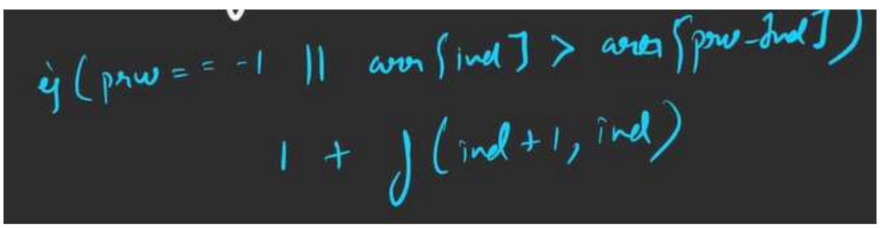
f(3,0) means length of LIS starting from index = 3 whose previous index is 0

If we do not pick any index due to any coniditon then our function becomes f(index+1, previous)
The previous index will remain same, but we skip the current index and move to next index.
And we know our f(index,prev) returns us the length so if we are not picking up a index, will it make any change in length of our LIS?
No, it does not so we return it like
return 0 + f(index+1, previous) // Not-take case

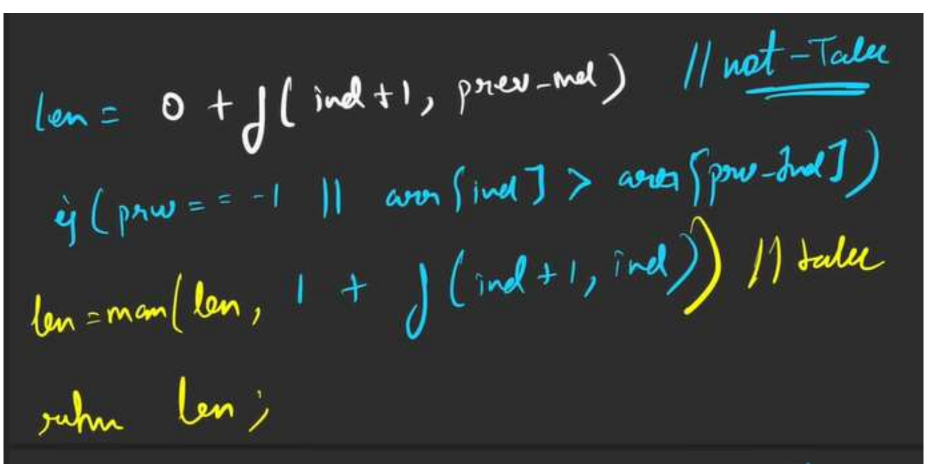
We start by picking or not picking the index = 0 where previous = -1



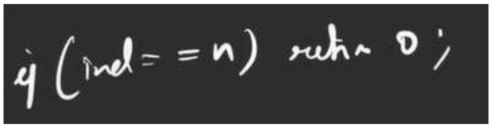
if we are **taking an index**, our previous changes and our length increases but keeping in mind the condition



So now max length is maximum of pick/not pick



What will be the base case:
If we have reached the end of array, means we do not have anything to add in length so return 0



Complexity:

TC: 2^n for take / not take
SC: O(n)

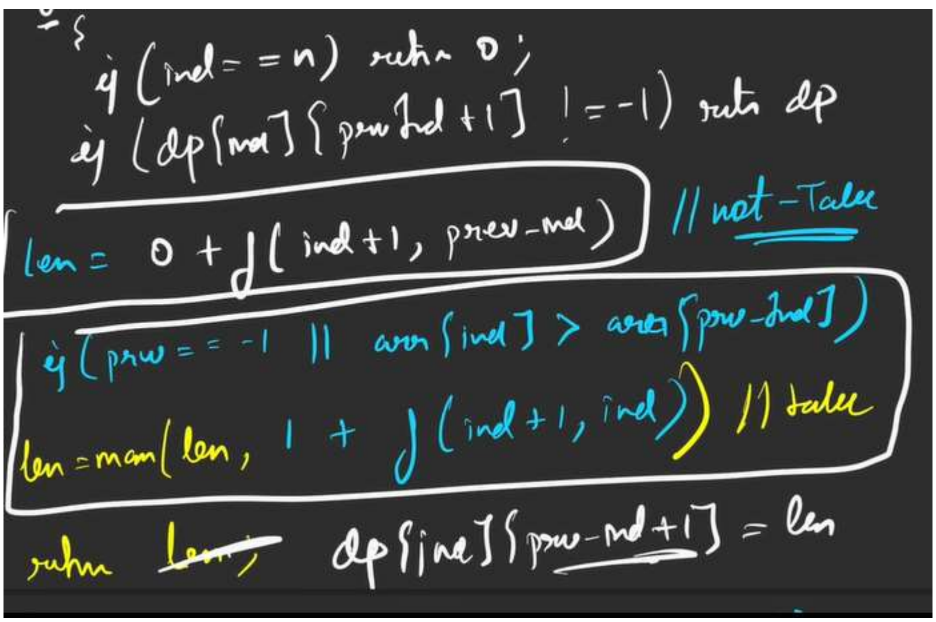
To optimise this, we look for over-lapping sub-problems

We convert them to memoization

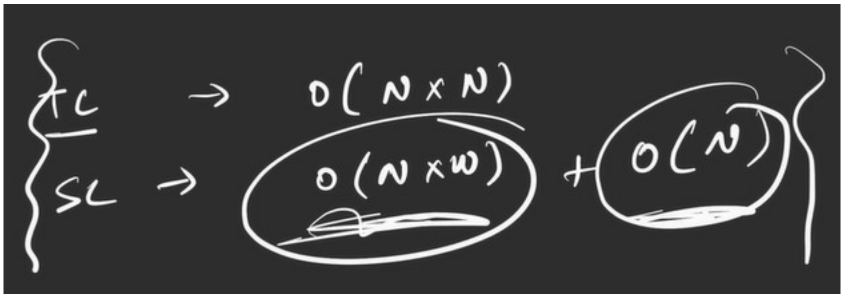
We are taking index from 0 to n, so we can take an array of [N]
We are taking index from -1 to n-1 for previous but how to store -1 in array?

We will do coordinate shifting and shift
-1 -> 0
0 -> 1
1 -> 2
.
.
.
n-1 -> n
So we take an array of [N+1]

So now our code changes to:



Complexity changes to



Code

```
int longestLengthSubsequence(int index, int prev, int arr[], int n)
{
    if(index == n) return 0;
    // Not - take case
    int len = 0 + longestLengthSubsequence(index+1,prev,arr,n);
    // Take case
    if(prev==-1 || arr[index]> arr[prev])
    {
        //  prev = -1 means first index
        // arr[index]> arr[prev] means it makes a valid increasing s
ubsequence
        // add +
1 in length and move to next index, taking current index as previous
        // store the max length of take and not-
takes case as we need longest increasing subsequence
        len = max(len, 1+ longestLengthSubsequence(index+
1,index,arr,n));
    }
    return len;
}


int longestIncreasingSubsequence(int arr[], int n)
{
    return longestLengthSubsequence(0,-1,arr,n);
}
```

Memoization code

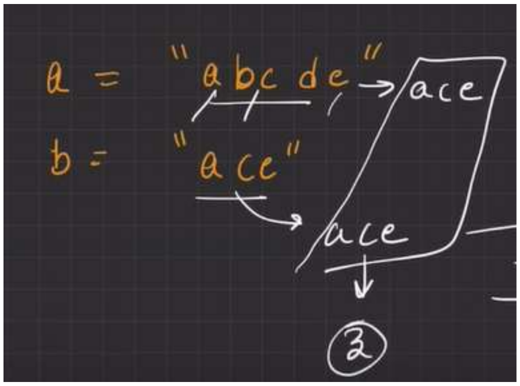
```
int longestLengthSubsequence(int index, int prev, int arr[], int n, 
vector<vector<int>> &dp)
{
    if(index == n) return 0;
    if(dp[index][prev+1] != -1) return dp[index][prev+1];
    // Not - take case
    int len = 0 + longestLengthSubsequence(index+1,prev,arr,n,dp);
    // Take case
    if(prev==-1 || arr[index]> arr[prev])
    {
        //  prev = -1 means first index
        // arr[index]> arr[prev] means it makes a valid increasing s
ubsequence
        // add +
1 in length and move to next index, taking current index as previous
        // store the max length of take and not-
```

```
takes case as we need longest increasing subsequence
        len = max(len, 1+ longestLengthSubsequence(index+
1,index,arr,n,dp));
    }
    return dp[index][prev+1] = len;
}


int longestIncreasingSubsequence(int arr[], int n)
{
    // Memoization
    vector<vector<int>> dp(n,vector<int> (n+1,-1));
    return longestLengthSubsequence(0,-1,arr,n,dp);
}
```

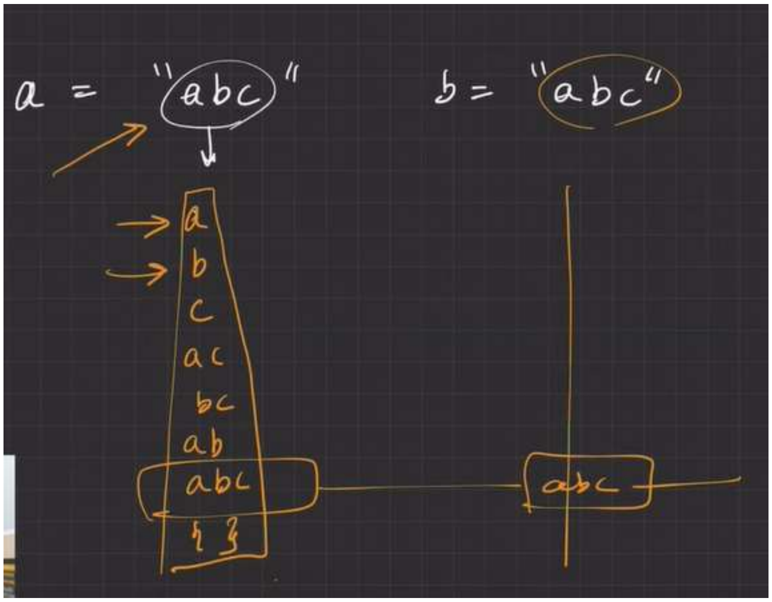
Longest Common Subsequence

Given 2 strings, return longest subsequence from both string which are common to both strings, and is present in both the strings.
Subsequence means the relative ordering should be same.

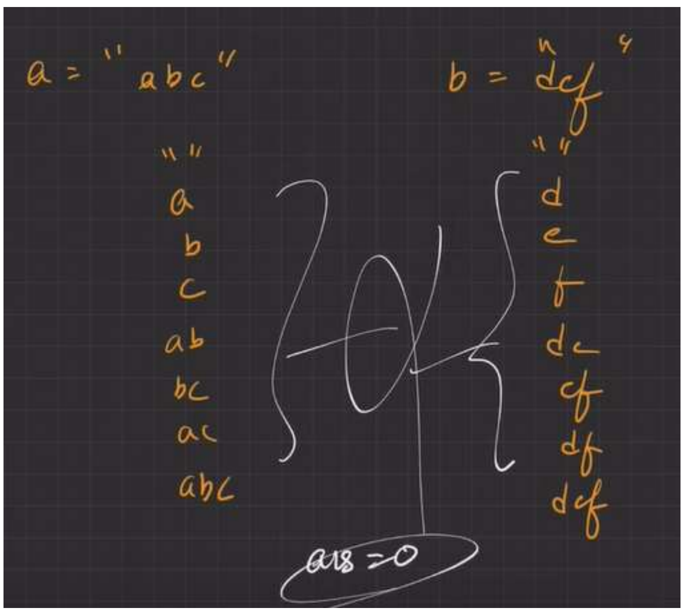
Like in below example:
"ace" is common in both strings with same relative ordering
So we return length = 3



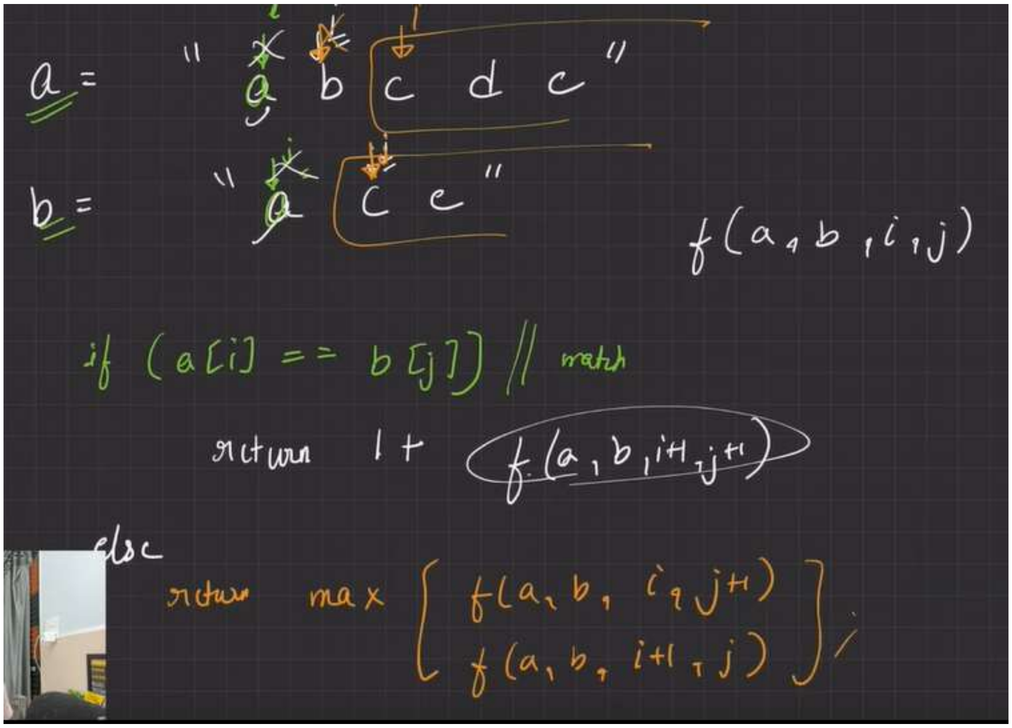
Another example:



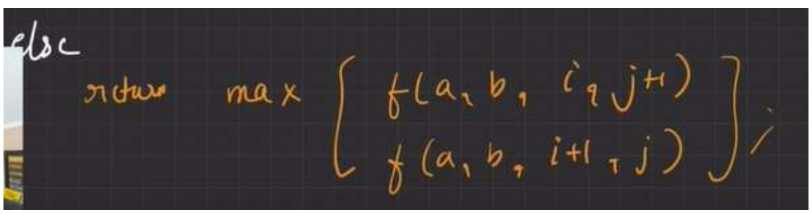
In below example, there is nothing common so ans = 0



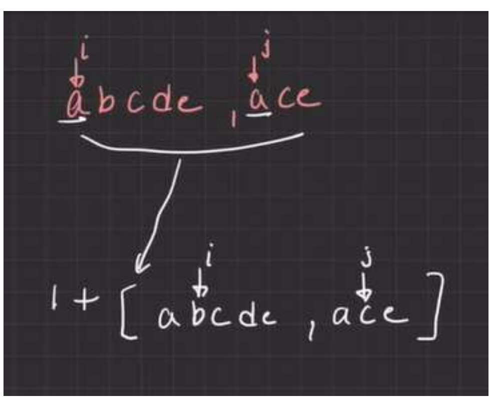
We will use recursion and put i and j in both string
Is arr1[i] == arr2[j] // match
Return 1 + f(arr1,arr2,i+1,j+1)



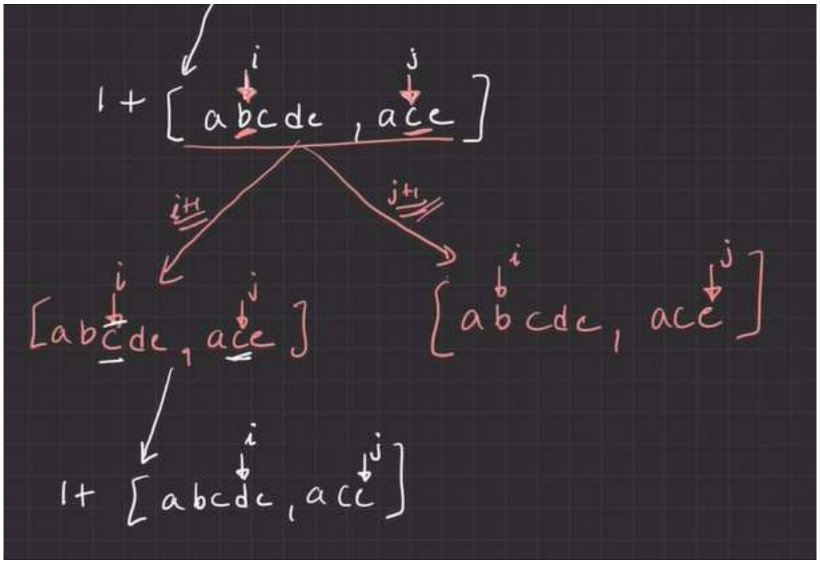
If they do not match we increase i first then we increase j. we tak maximum of both



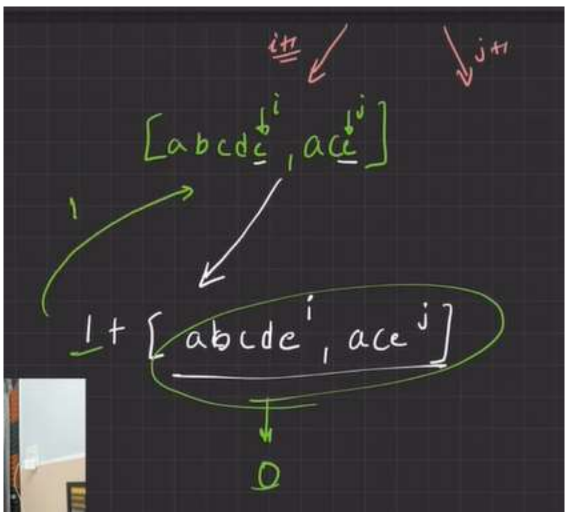
Working Tree



Next step



Once i and j reach out of array, we hit the base case, we return 0



Code: [gives TLE]

```
int solve(string s, string t, int i, int j)
{
    // base case
    if(i == s.length())
    {
        return 0;
    }
    if(j == t.length())
    {
        return 0;
    }
    // store ans
    int ans = 0;
    // if characters match move both pointer
    if(s[i]==t[j])
    {
        ans = 1 + solve(s,t,i+1,j+1);
    }
    else{
```

```
        // both do not match
        // ek baar i ko aage badhao, j ko rehene do
        // ek baar j ko aage badhao, i ko rehene do
        // get the maximum as result
        ans = max(solve(s,t,i+1,j), solve(s,t,i,j+1));
    }
    return ans;
}


int lcs(string s, string t)
{
    return solve(s,t,0,0);
}
```

Memoization (Top-down approach)

```
int solve(string s, string t, int i, int j, vector<vector<int>> &dp)
{
    // base case
    if(i == s.length())
    {
        return 0;
    }
    if(j == t.length())
    {
        return 0;
    }
    // if ans is already there in dp, do not make calls just return
the ans
    if(dp[i][j] != -1)
    {
        return dp[i][j];
    }
    // store ans
    int ans = 0;
    // if characters match move both pointer
    if(s[i]==t[j])
    {
        ans = 1 + solve(s,t,i+1,j+1,dp);
    }
    else{
        // both do not match
        // ek baar i ko aage badhao, j ko rehene do
        // ek baar j ko aage badhao, i ko rehene do
        // get the maximum as result
        ans = max(solve(s,t,i+1,j,dp), solve(s,t,i,j+1,dp));
    }
    return dp[i][j] = ans;
}


int lcs(string s, string t)
{
    // Memoization
    // we see i and j are changing so we need 2D dp
    vector<vector<int>> dp(s.length(), vector<int> (t.length(),-1));
    return solve(s,t,0,0,dp);
```

```
}
```

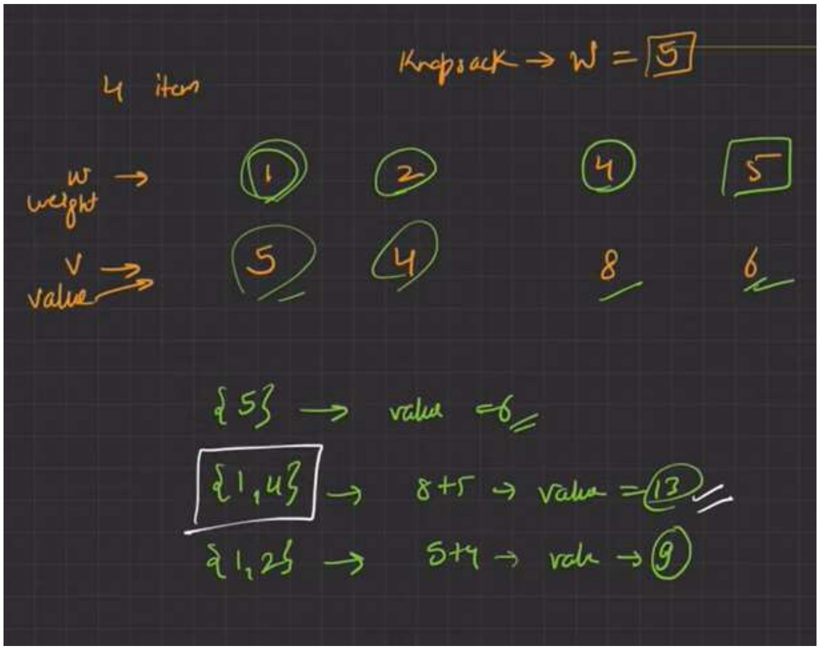
Bottom-up DP (Tabulation)

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

int lcs(string s, string t)
{
    // using tabulation
    int n = s.length();
    int m = t.length();
    return solveTab(s,t,n,m);
}
```

0/1 KnapSack Problem || learn 2-D DP Concept

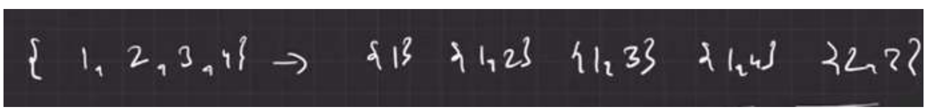
A thief has a bag which can carry only 'W' weight, he has to theft some items such that the weight is maximum and within "W".

Example:

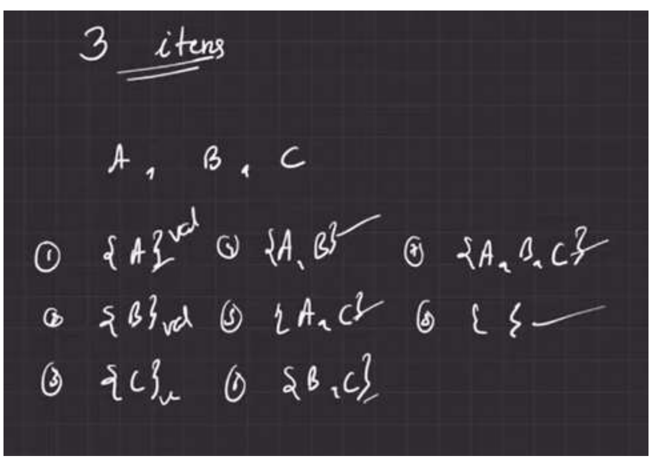


First approach (Brute force)

We are taking combination of items, pick/not pick
Like taking an subset

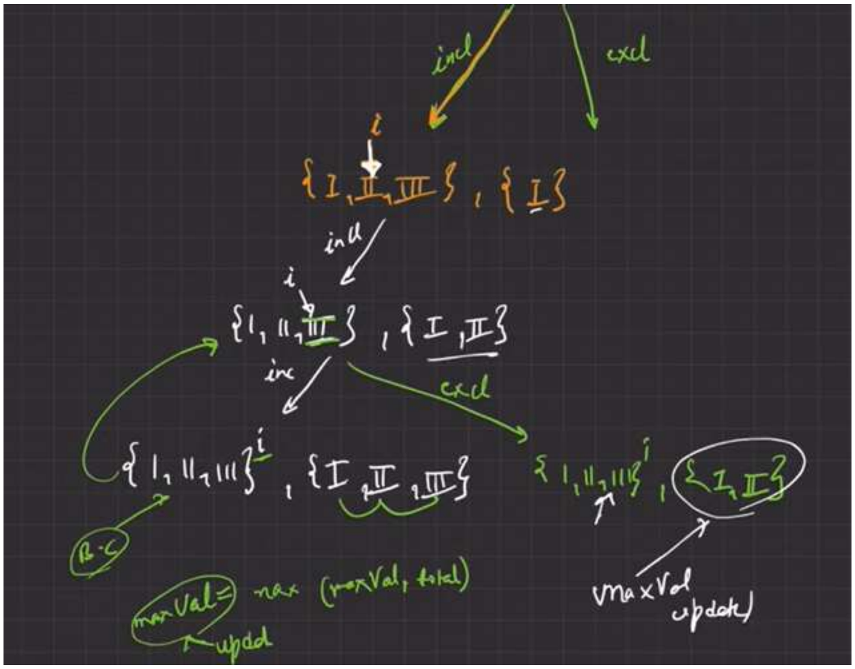


We take our combination of all weight/values and return maximum value out of it.
Let say we have 3 items A,B,C so I can have 8 combinations.



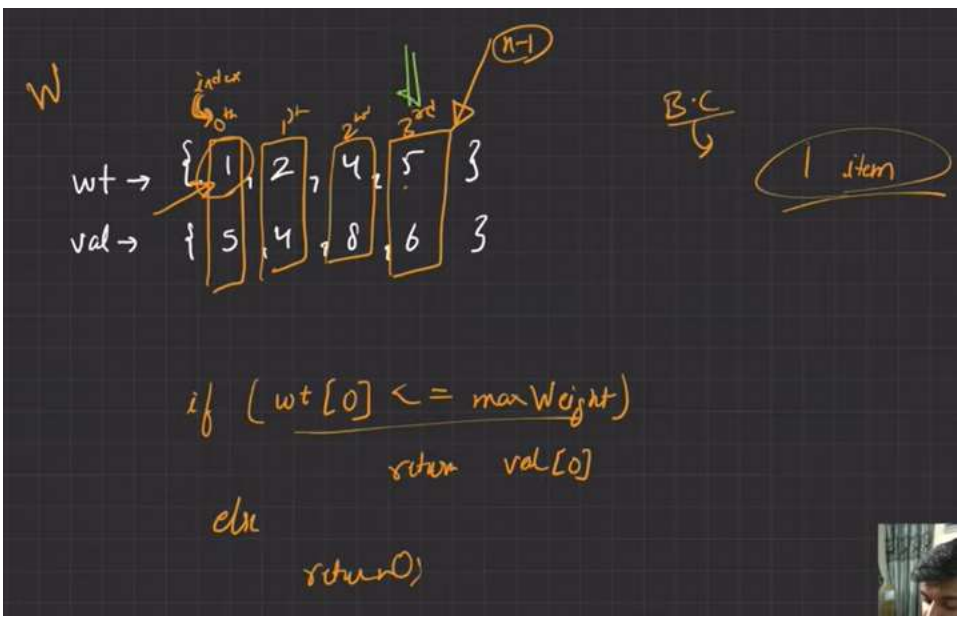
So we can use (include / not include) approach using recursion

We take a pointer and a DS
We include, we move pointer and include item in DS till Index does not go beyond array.size()
We take a variable maxValue = INT_MIN and we update it everytime base case reaches with DS size



Code:

We will start from last index and do (include/ not include) thing
Base case becomes
If index==0, we check if weight is < maxWeight, we include it val[0] else we return 0



```
int solve(vector<int> &values,vector<int> &weights, int index, int c
apacity)
{
    // base case
    // if we have only one item left, whether we can include or not
depends on space we have left in our knapsack
    // we are starting from end of the array, means we are at the in
dex = 0 when we have base case reached so
```

```
    if(index==0)
    {
        if(weights[0] <= capacity)
        {
            return values[0];
        }
        else{
            // we cannot include last element so return 0
            return 0;
        }
    }

    // now we either include a item or we exclude it
    // if we inlude weight[i] we do capacity - weight[i]
    // we + values[i] in our answer also
    int include = 0;
    // when can we include something?
    // if we can fit it in our knapsack
    if(weights[index] <= capacity)
    {
        // if included, move to next index means index-1 as we are g
oing last -> first index
        include = values[index] + solve(values,weights,index-1,capac
ity - weights[index]);
    }
    // if excluded, capacity remains same
    // we move to next index that is index - 1 as we start from last
 index
    // we add 0 value in answer
    int exclude = 0 + solve(values,weights,index-1,capacity);
    // our ans will be the one maximum(include,exclude)
    int ans = max(include,exclude);
    return ans;

}

int maxProfit(vector<int> &values, vector<int> &weights, int n, int
w)
{
    return solve(values,weights,n-1,w);
}
```

Memoisation

1. Create DP array and initialise it with -1
2. Store result of recursive call in DP array
3. Check in base case if answer is in DP array, no need of further calculation return from DP array

We create 2D DP because our 2 parameters are changing, index and capacity so we use index and capacity in 2D DP.

```
int solve(vector<int> &values,vector<int> &weights, int index, int c
apacity,vector<vector<int>> &dp)
```

```
{
    // base case
    // if we have only one item left, whether we can include or not
depends on space we have left in our knapsack
    // we are starting from end of the array, means we are at the in
dex = 0 when we have base case reached so
    if(index==0)
    {
        if(weights[0] <= capacity)
        {
            return values[0];
        }
        else{
            // we cannot include last element so return 0
            return 0;
        }
    }
    // check DP array for ans
    if(dp[index][capacity] != -1)
    {
        return dp[index][capacity];
    }

    // now we either include a item or we exclude it
    // if we inlude weight[i] we do capacity - weight[i]
    // we + values[i] in our answer also
    int include = 0;
    // when can we include something?
    // if we can fit it in our knapsack
    if(weights[index] <= capacity)
    {
        // if included, move to next index means index-1 as we are g
oing last -> first index
        include = values[index] + solve(values,weights,index-1,capac
ity - weights[index],dp);
    }
    // if excluded, capacity remains same
    // we move to next index that is index - 1 as we start from last
 index
    // we add 0 value in answer
    int exclude = 0 + solve(values,weights,index-1,capacity,dp);
    // our dp[index]
[capacity] will be the one maximum(include,exclude)
    dp[index][capacity] = max(include,exclude);
    return dp[index][capacity];

}

int maxProfit(vector<int> &values, vector<int> &weights, int n, int
w)
{
     // creating 2D dp array of n rows and w+1 columns for index and
     capacity as these are only changing parameters
    vector<vector<int>> dp(n, vector<int> (w+1,-1));
    return solve(values,weights,n-1,w,dp);
}
```

Tabulation (Bottom-up DP)

1. Inside Tabulation, create your own DP array initialised with 0
2. Analyse base case

```
int solveTabulation(vector<int> &values, vector<int> &weights,int n,
 int capacity)
{
    vector<vector<int>> dp(n, vector<int>(capacity+1,0));
    // we analyse the base case
    // base case runs for weight[0]
    for(int w = weights[0]; w<=capacity; w++)
    {
            if(weights[0] <= capacity)
            {
                dp[0][w] = values[0];
            }
            else{
                dp[0][w] = 0;
            }
    }
    // check other cases
    // our rows are of size = n
    // so our index will go from 0 to n-1, in base case we have done
 for 0th row, so we run ouer loop from i = 1 to i<n
    // our capacity will start from 0 to capacity so inner loop runs
 from 0 to <= capacity
    for(int index = 1; index<n; index++)
    {
        for(int w = 0; w<=capacity; w++)
        {
            int include = 0;
            if(weights[index] <= w)
            {
                include = values[index] + dp[index-1]
[w - weights[index]];
            }
            int exclude = 0 + dp[index-1][w];
            dp[index][w] = max(include,exclude);

        }

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


int maxProfit(vector<int> &values, vector<int> &weights, int n, int
w)
{
    return solveTabulation(values,weights,n,w);
}
```

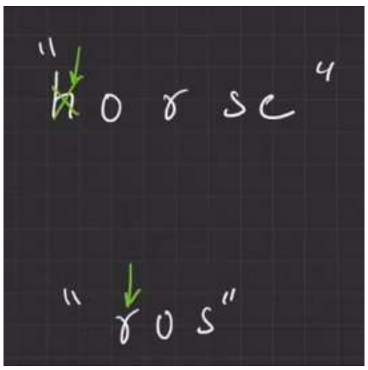
Edit Distance

Given two strings word1 and word2, return the minimum number of operations required to convert word1 to word2.
You have the following three operations permitted on a word:

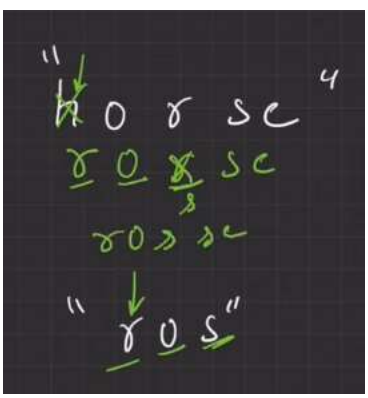
1. Insert a character
2. Delete a character
3. Replace a character

Let say cost for performing each operation is 1

Let say we have 2 strings "horse" and "ros"
We compare first characters we reolace 'h' with 'r'



We move to next 'o'
It matches
Now we go to next "r" of horse, not matching so make it "s"

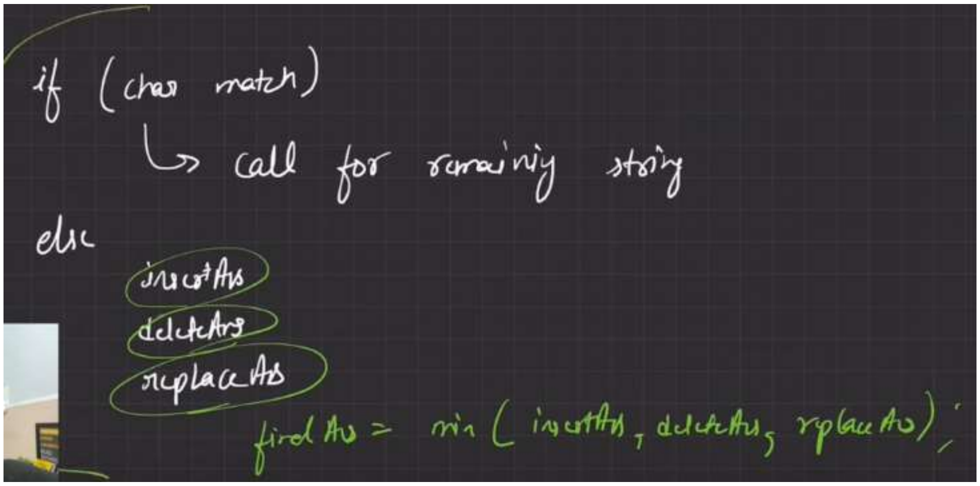


Now ros are mathced in both, delete 's' and 'e' from horse
So it took 4 operations replace,replace,delete,delete to comvert horse to ros

There can be other ways also, return minimum number of operations

If(character matches) we call function for remaining string
If they do not match
We insert and get an ansI
We delete and get an ansD

We replace and get an ansR
Our final answer is min(ansI, ansD, ansR)

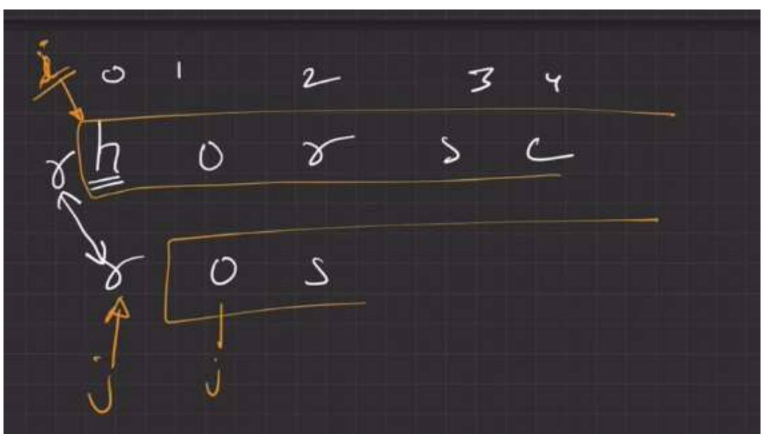


Base case:

i is out of string 1 means string 1 is smaller than string 2
Then we return the number of characters by which string 2 is larger than string 1
That is (string2.length - j)

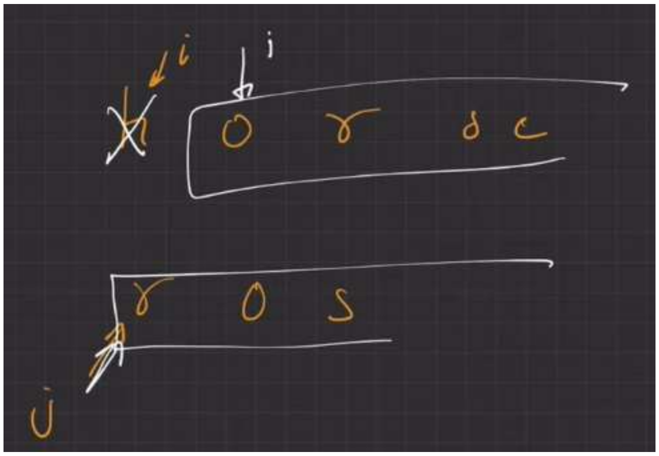
j is out of string 2 then return (a.length - i)

In our insert case, i will remain at its place as we are inserting string2[j] in string1 so i remains as it is, j becomes j + 1




```
//insert
int insertAns = solve(a, b, i, j+1);
```

For Delete
i will move by 1 and j remains as it is



```
//delete
int deleteAns = solve(a, b, i+1, j);
```

For Replace

i,j both move by 1

```
//insert
int insertAns = 1 + solve(a, b, i, j+1);
//delete
int deleteAns = 1 + solve(a, b, i+1, j);
//replace
int replaceAns = 1 + solve(a, b, i+1, j+1);

ans = min(insertAns, min(deleteAns, replaceAns));
```

Code:

```
int distance(string s1, string s2, int i,int j)
  {
     // base case
     if(i== s1.length())
     {
        // means s1 is smaller than s2
        // so return remaining elements of s2 as these many operations will be needed
to match s1 and s2 so return
        return s2.length() - j;
     }

     if(j == s2.length())
     {
        // means s2 smaller than s1
        // return remaining element of s1
        return s1.length() - i;
     }

     int ans = 0;

     // if both characters are equal, no operation needed just move to comparing next
```

```
indexes
      if(s1[i]==s2[j])
      {
         return distance(s1,s2,i+1,j+1);
      }
      else{
         // both not equal, we need to perform insert,delete,replace
         // insert
         // i remain as it is, j move by 1
         int insertAns = 1 + distance(s1,s2,i,j+1);

         // delete
         // j remains as it is, i move by 1
         int deleteAns = 1 + distance(s1,s2,i+1,j);

         // replace
         // j and i both move
         int replaceAns = 1 + distance(s1,s2,i+1,j+1);

         // store minimum of all in ans
         ans = min(insertAns, min(deleteAns, replaceAns));
      }

      return ans;
   }

   int minDistance(string word1, string word2) {
      return distance(word1,word2,0,0);
   }
```

Memoization code

i and j are changing so we use 2D DP

```
int distance(string s1, string s2, int i,int j,vector<vector<int>> &
dp)
    {
        // base case
        if(i== s1.length())
        {
            // means s1 is smaller than s2
            // so return remaining elements of s2 as these many oper
ations will be needed to match s1 and s2 so return
            return s2.length() - j;
        }

        if(j == s2.length())
        {
            // means s2 smaller than s1
            // return remaining element of s1
            return s1.length() - i;
        }

        // check dp
```

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

        int ans = 0;

        // if both characters are equal, no operation needed just mo
ve to comparing next indexes
        if(s1[i]==s2[j])
        {
            return distance(s1,s2,i+1,j+1,dp);
        }
        else{
            // both not equal, we need to perform insert,delete,repl
ace
            // insert
            // i remain as it is, j move by 1
            int insertAns = 1 + distance(s1,s2,i,j+1,dp);

            // delete
            // j remains as it is, i move by 1
            int deleteAns = 1 + distance(s1,s2,i+1,j,dp);

            // replace
            // j and i both move
            int replaceAns = 1 + distance(s1,s2,i+1,j+1,dp);

            // store minimum of all in ans
            ans = min(insertAns, min(deleteAns, replaceAns));
        }

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


  int editDistance(string word1, string word2) {
        vector<vector<int>> dp(word1.length(), vector<int> (word2.le
ngth(),-1));
        return distance(word1,word2,0,0,dp);
    }
```

Tabulation Approach (Bottom up approach)

```
int solveTabulation(string a, string b)
    {
        // make dp array
        vector<vector<int>> dp(a.length()+
1, vector<int> (b.length()+1,0));

        // convert the base cases
        for(int j = 0;j<b.length();j++)
        {
           // in a.length vali row fill
           dp[a.length()][j] = b.length() - j;
        }
```

```
        for(int i = 0;i<a.length();i++)
        {
           // in b.length vali row fill
           dp[i][b.length()] = a.length() - i;
        }

        // now for other cases
        // we go from bottom to up
        for(int i = a.length()-1;i>=0; i--)
        {
            for(int j = b.length()-1; j>=0 ; j--)
            {
                int ans = 0;
                // if both characters are equal, no operation needed
 just move to comparing next indexes
                if(a[i]==b[j])
                {
                    ans = dp[i+1][j+1];
                }
                else{
                    // both not equal, we need to perform insert,del
ete,replace
                    // insert
                    // i remain as it is, j move by 1
                    int insertAns = 1 + dp[i][j+1];
                    // delete
                    // j remains as it is, i move by 1
                    int deleteAns = 1 + dp[i+1][j];
                    // replace
                    // j and i both move
                    int replaceAns = 1 + dp[i+1][j+1];
                    // store minimum of all in ans
                   ans = min(insertAns, min(deleteAns, replaceAns));
                }
                dp[i][j] = ans;
            }
        }
        return dp[0][0];
    }

int editDistance(string word1, string word2) {
    return solveTabulation(word1,word2);
}
```

Maximum sum increasing subsequence (Prerequisite: LIS)

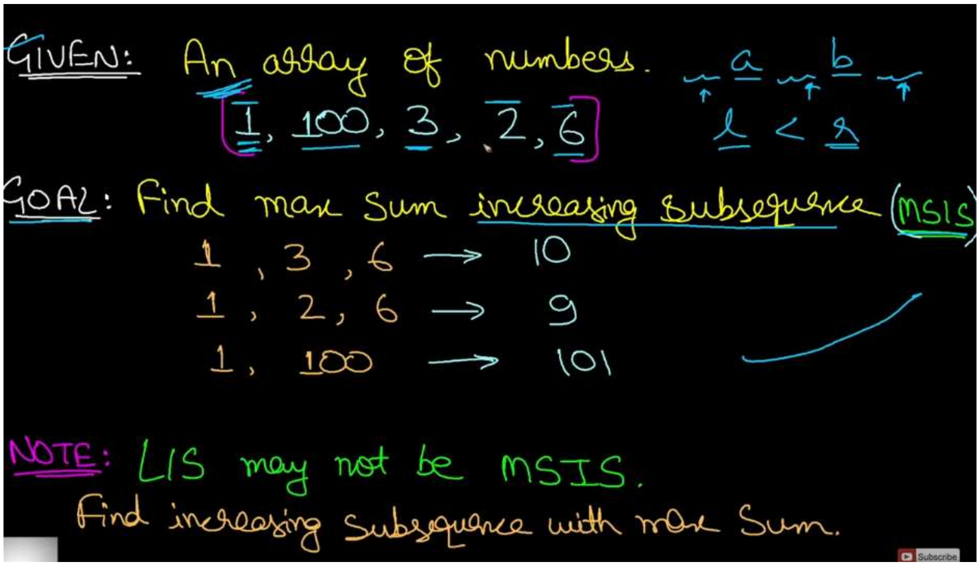
Given an array of n positive integers. Find the sum of the maximum sum subsequence of the given array such that the integers in the subsequence are sorted in strictly increasing order i.e. a strictly increasing subsequence.

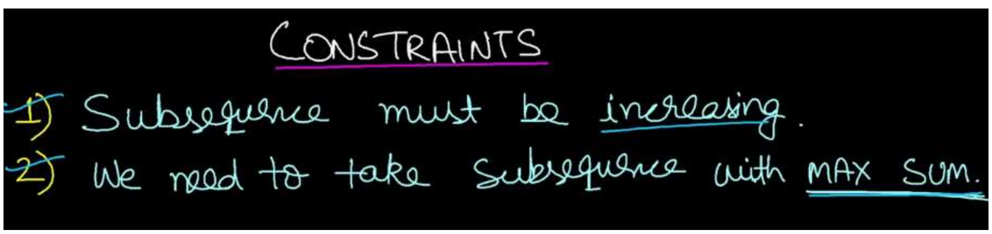
Example 1:

Input: N = 5, arr[] = {1, 101, 2, 3, 100}

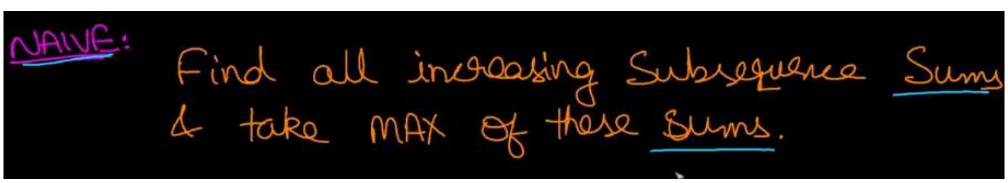
```
Output: 106
Explanation:The maximum sum of a
increasing sequence is obtained from
{1, 2, 3, 100}
```

LIS may not be MSIS (Maximum Sum increasing Subsequence)
We do not need longest increasing subsequence
We need maximum sum increasing subsequence



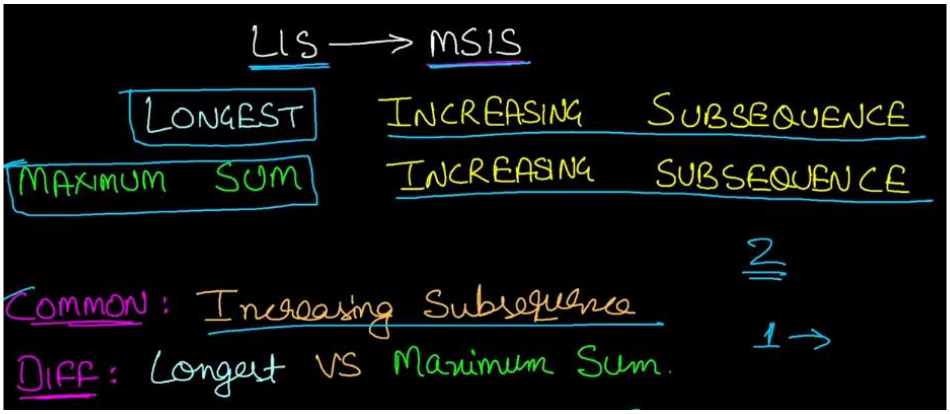


Naïve Approach gives TLE:



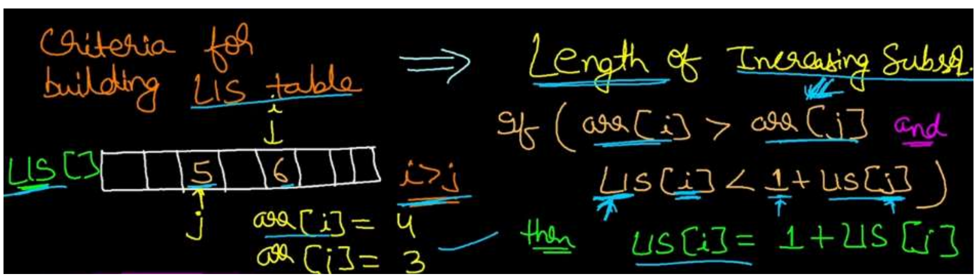
Optimised Approach

Here also we need an increasing subsequence



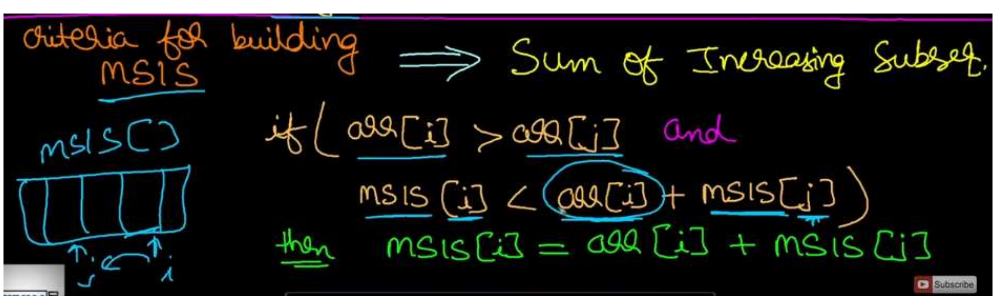
In LIS we keep track of longest length
In MSIS, we will keep track of maximum sum

Criteria we followed for LIS:
Keeping in mind, i pointer is ahead j pointer
If arr[i] > arr[j] means increasing sequence &&
LIS[i] < 1 + LIS[j] means length is more than LIS[j], then we have one more longest increasing subsequence so store it.
We are doing +1 above because including an element means adding one more element in the sequence



Criteria for MSIS:

Same for increasing sequence
We check for maximum sum also, MSIS[i] < arr[i] + MSIS[j]
Means we can store that sum, where i is current index .
Here we are doing + arr[i] because including the element means adding its value inside maximum sum



TC: O(n^2)

Code:

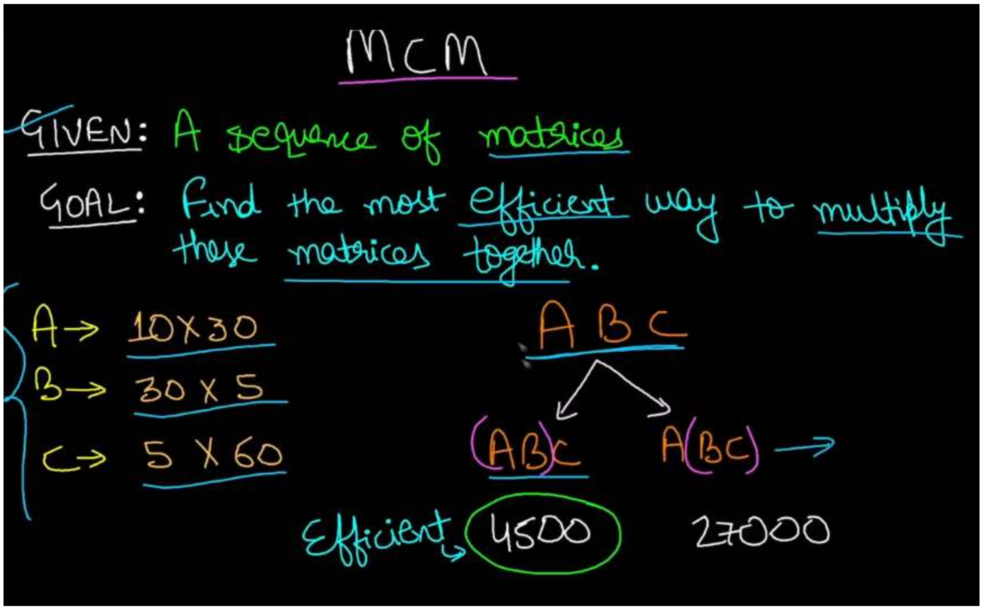
```
int solve(vector<int> &arr, int n)
{
    int maxi = 0;
    int MSIS[n];

    for(int i = 0;i<n;i++)
    {
        // fill MSIS
        MSIS[i] = arr[i];
    }

    // Fill MSIS with maximum increasing subsequence sum for any ind
ex
    for(int i = 1;i<n;i++)
    {
        for(int j = 0;j<i;j++)
        {
            if(arr[i]>arr[j] && MSIS[i] < MSIS[j] + arr[i])
            {
                // for 0th index there is no one behind so arr[0] is
 its max value
                // for other index we check from 0 till that index
                // if it is forming increasing subsequence which we
check by arr[i] > arr[j]
                // if MSIS has lower value, update it
                // we can update MSIS[i]
                MSIS[i] = arr[i] + MSIS[j];
            }
        }
    }

    // get max sum value
    for(int i = 0;i<n;i++)
    {
        if(maxi < MSIS[i])
        {
            maxi = MSIS[i];
        }
    }

    return maxi;
}

int maxIncreasingDumbbellsSum(vector<int> &arr, int n)
{
    return solve(arr,n);
}
```

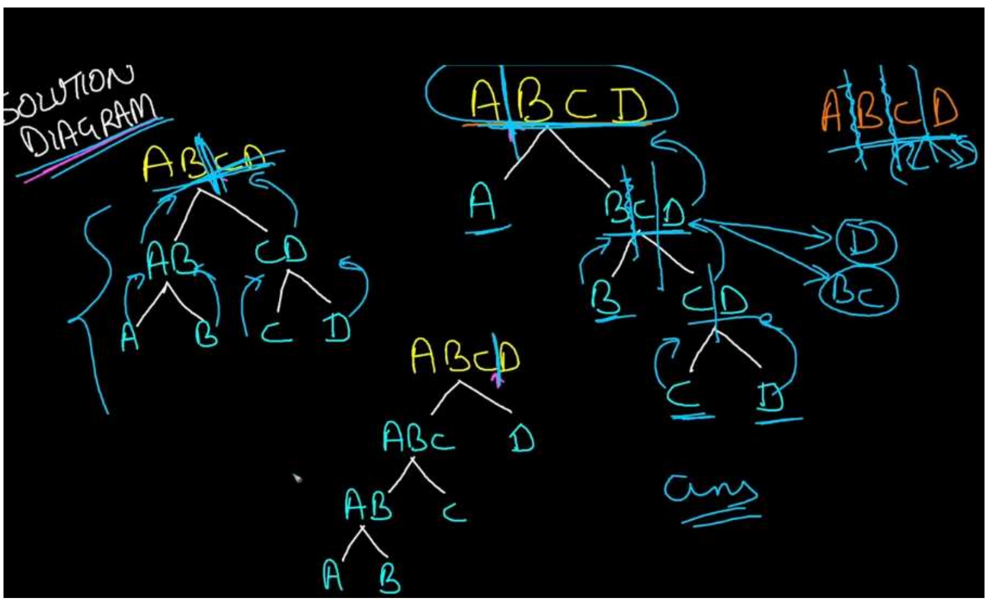
Matrix Chain Multiplication

Given a sequence of matrices, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications.
The dimensions of the matrices are given in an array **arr[]** of size **N** (such that N = number of matrices + 1) where the **ith** matrix has the dimensions **(arr[i-1] x**

arr[i]).

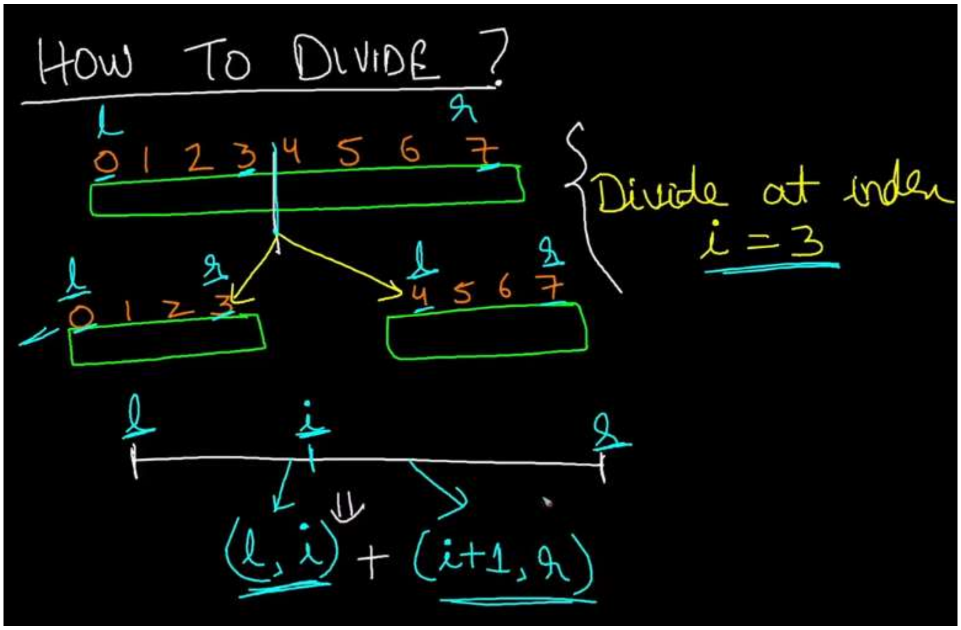
Approach to MCM Problems



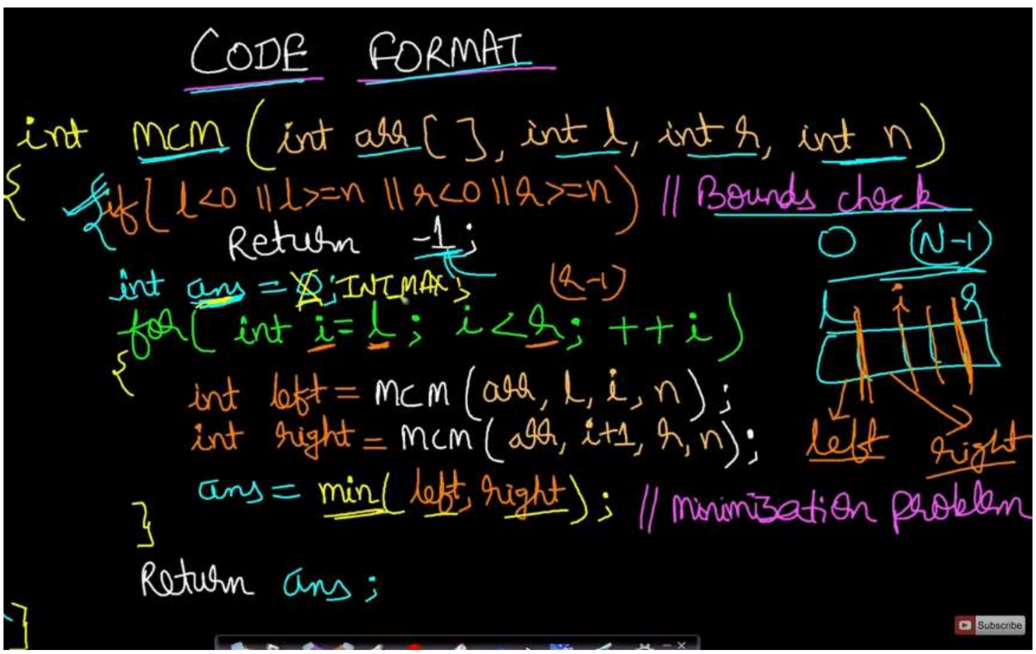


For partitioning there must be a left limit and right limit between which our division will take place.
We can solve each partition using recursion and then we can use max/min according to our answer and consider the max/min answer while returning in recursion based on our problem statement

What is the idea behind division?

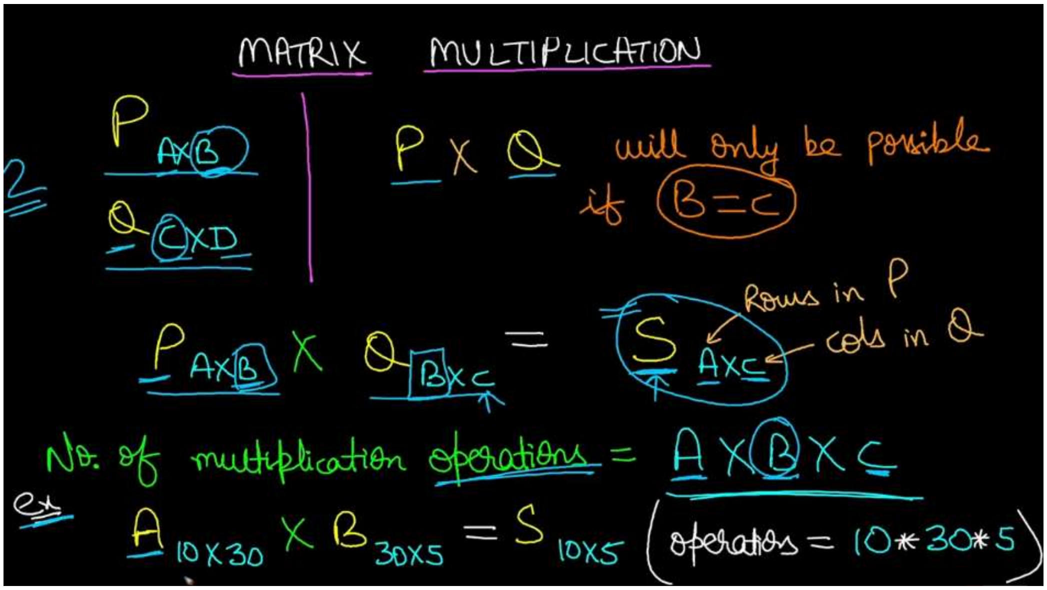


Code Format:
To solve the divided array we can use below code.
But, we need to do division from many parts to explore all the possiblities.



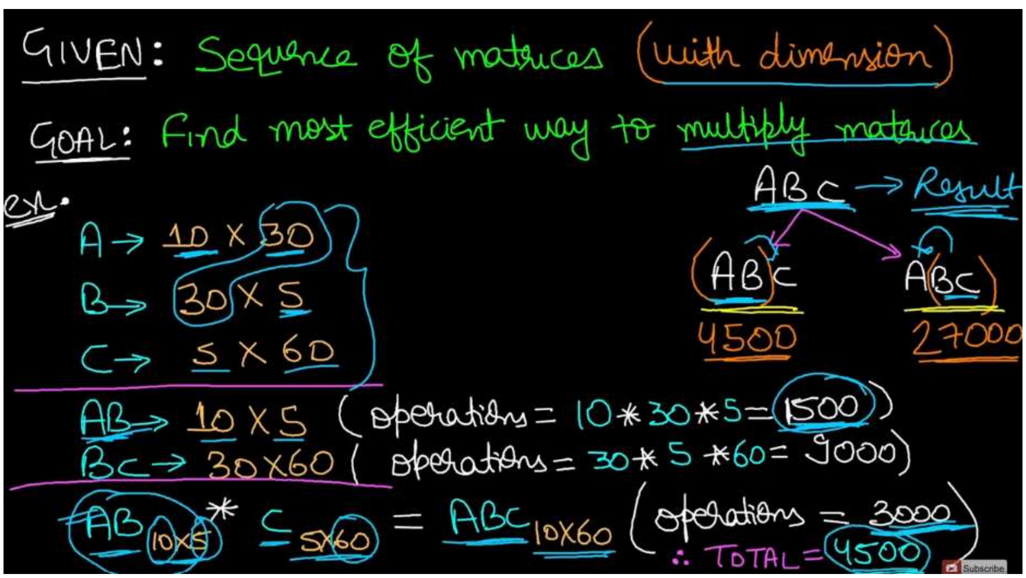
Actual Solution starts from here

In order to multiply 2 matrices, both matrices should have same number of columns.



Multiplication is an Associative operation so in which order we multiply does not matter
(AB)C or A(BC) will have same result but we need minimum number of operations to do this multiplication

In below example, we take 1500 operations to convert A and B to (AB)
Now it takes 3000 operations more to convert (AB)*(C) to ABC
So total operations will be 3000 + 1500 = 4500

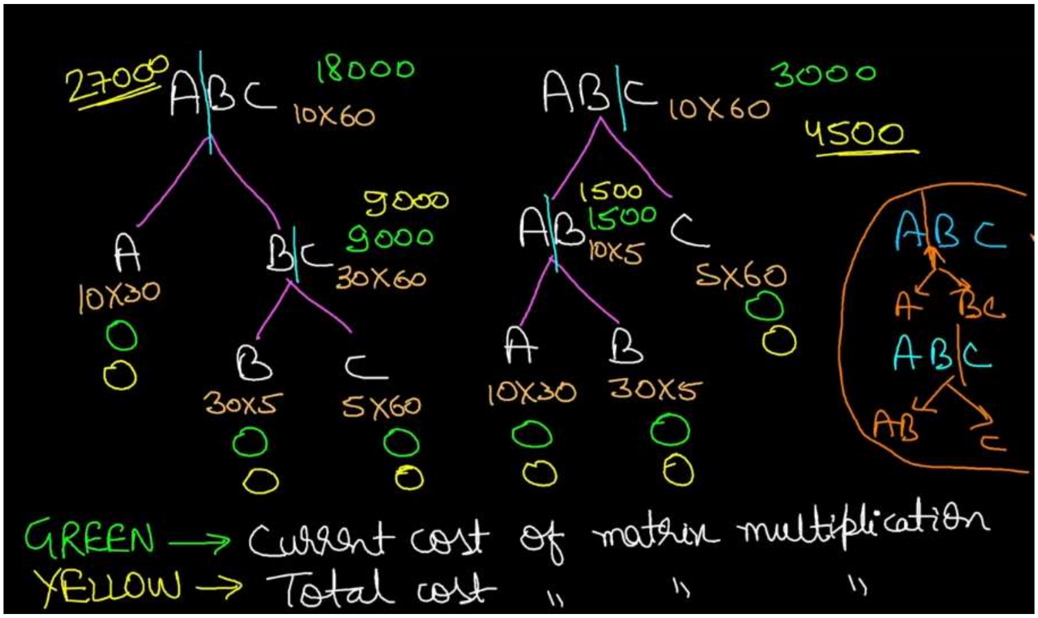


Now if we do (BC) * A, we take 27000 operations so this way is not optimised.

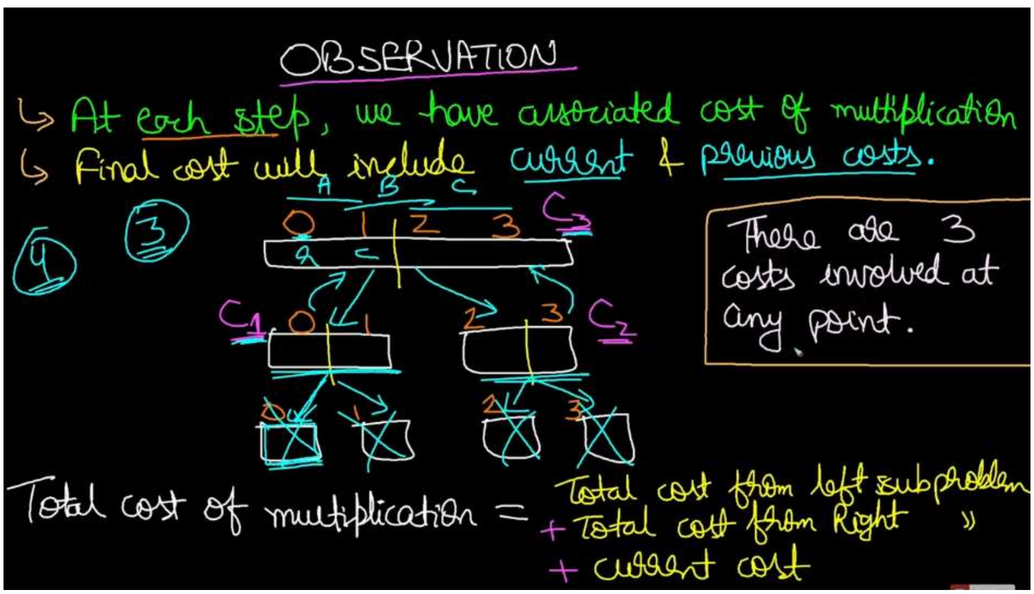
Approach

Current Cost = Cost of current matrix element [ith matrix]
Total Cost = Current cost + other cost (coming from recursion)

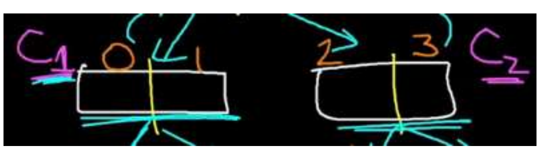
For a single matrix Current cost and Total cost both are = 0



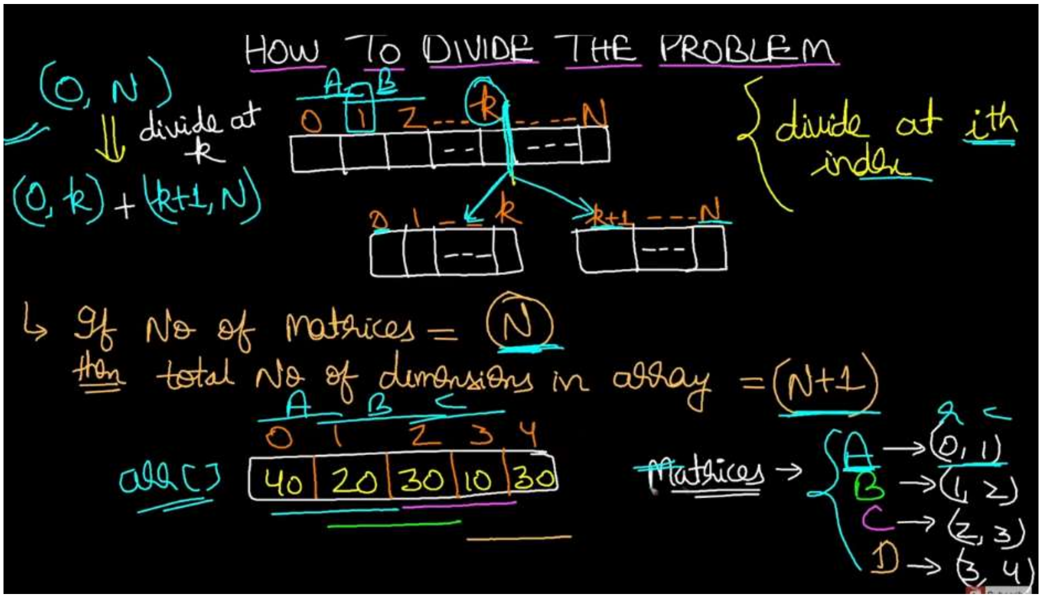
Observations



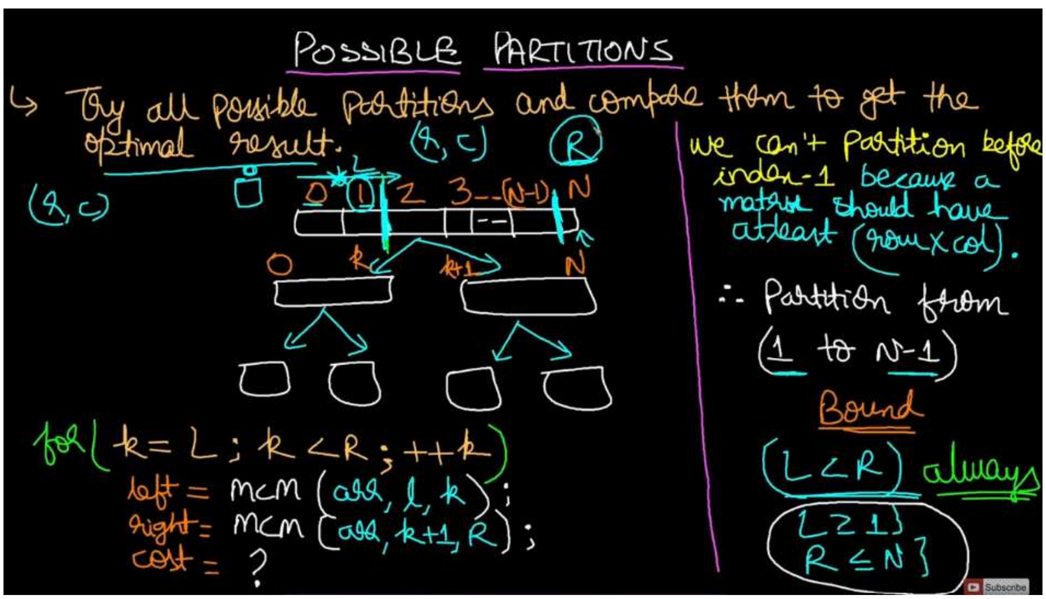
Our Leaf node will look like



Because we need L and R to divide the matrix so this will be our base case

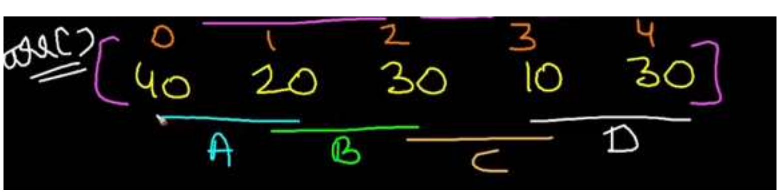


To do parition we need atleast one row and one column
So we cannot partition in index = 0 and 1 we need to partition after index = 1
And we cannot parition after index = N so we need to partition between index = 2 to N
Keeping in mind, left < right and L >= 1 and R<=N



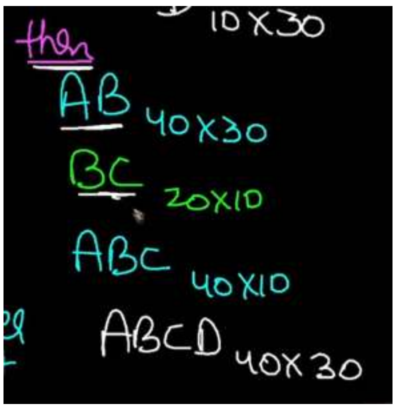
In our problem statement, our array contains the matrix values where arr[i-1] is row of ith matrix and arr[i] is column of ith matrix

A, B, C, D, are our 4 matrices. Whose dimensaions are shown at indexes

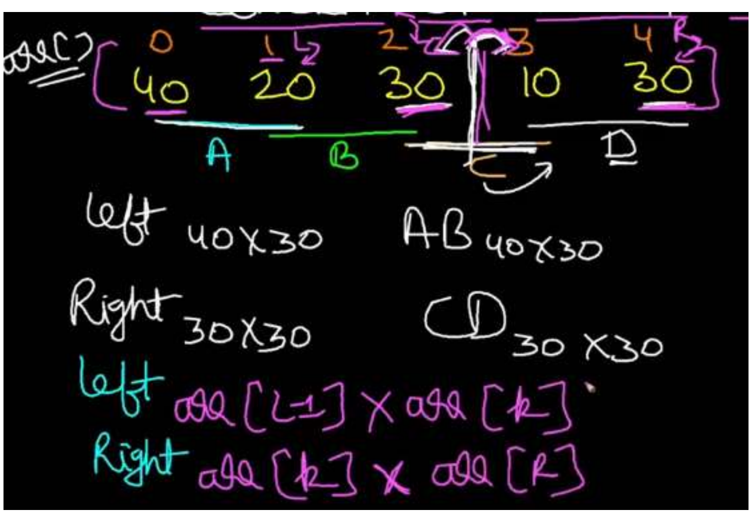


Our matrices are like:

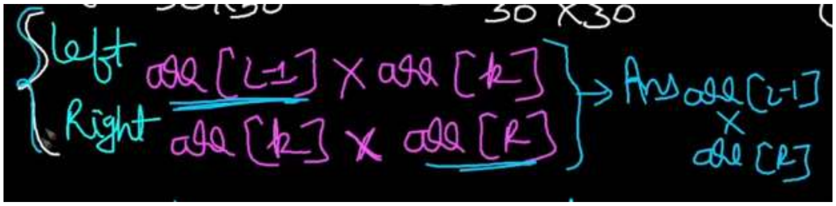
We can only multiply 2 matrices if there column number is same.
For ABC, we check number of column of AB and number of column of C, if they are same. Then, **ABC has number of rows = number of rows of AB and number of column = number of column of C**



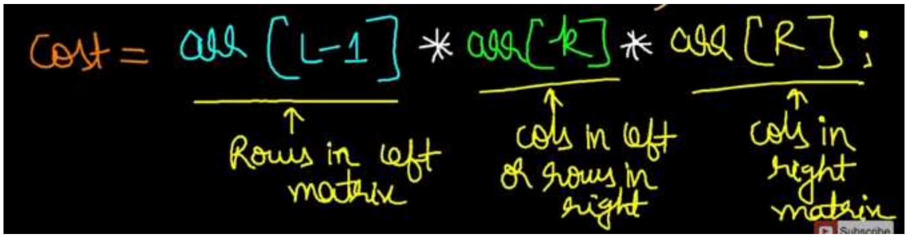
So, Dimensions of left partition result will be arr[L-1] X arr[K]
Dimensions of right partition result will be arr[K] X arr[R]



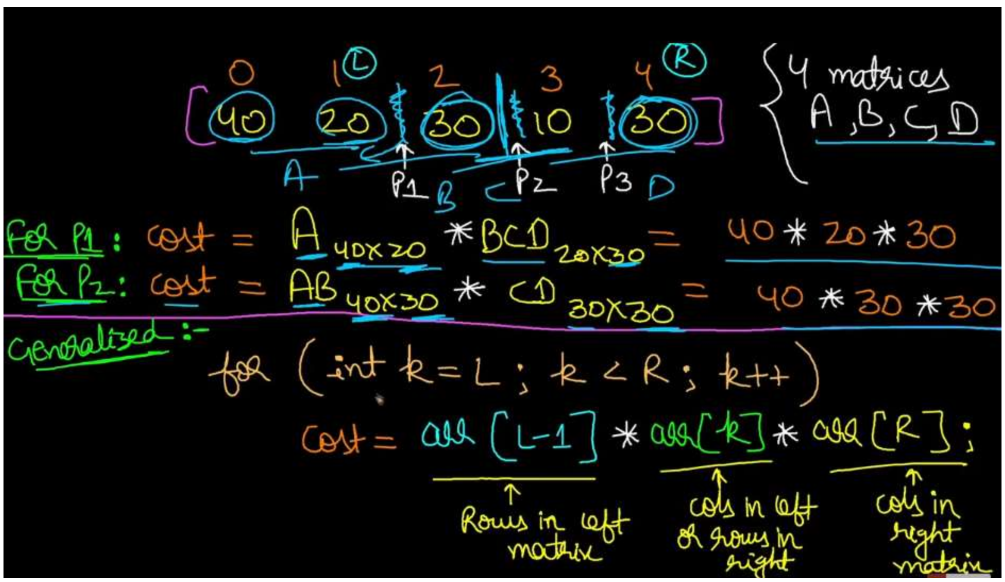
Resultans Answer will have dimension arr[L-1] X arr[R]



Cost will be:

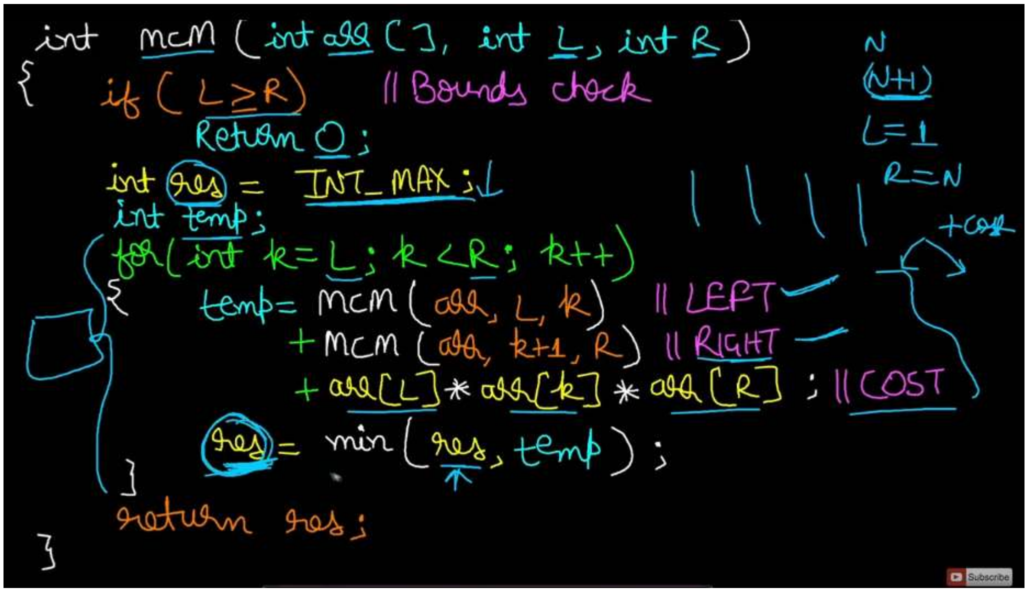


Pseudo-Code:



Code:

L = 1, R = N initially

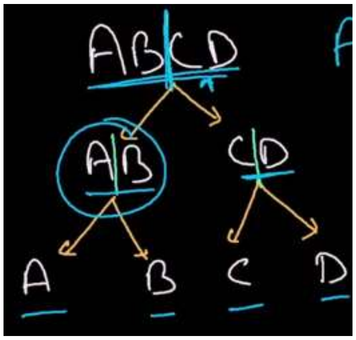


Code (Recursive) (gives TLE):

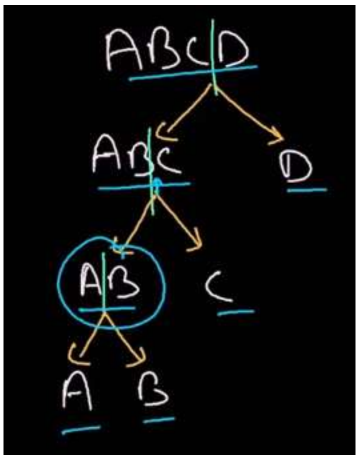
```
int solve(int N, int arr[], int left, int right)
{
    if(left>=right)
    {
        return 0;
    }

    int res = INT_MAX;
    int temp = 0;

    for(int k = left; k<=right-1; k++)
    {
        temp = solve(N,arr,left,k) + solve(N,arr,k+
1,right) + (arr[left-1]*arr[k]*arr[right]);
        res = min(res,temp);
    }

    return res;
}

int matrixMultiplication(int N, int arr[])
{
    return solve(N,arr,1,N-1);
}
```

Code (Memoised)

If we partition from AB|CD

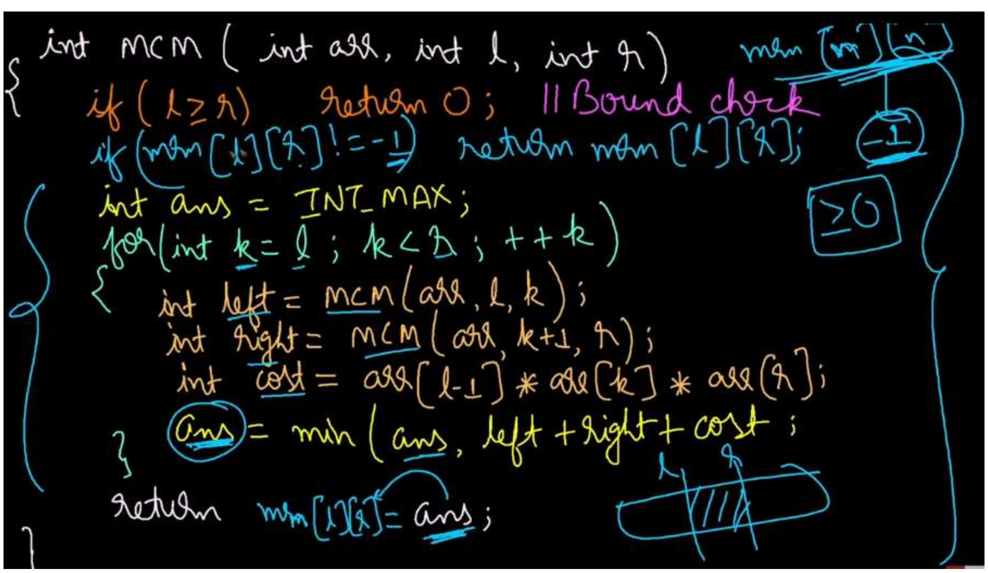


If we partition from ABC | D



We see, in both case **there are many overlapping subproblems**
In actual answer, we partition from all points
A | B | C | D
And try to get minimum answer and return it.
We can take an 2D vector of size [n][n]

Memoized Code




```
int solve(int N, vector<int> arr, int left, int right, vector<vector
<int>> &dp)
```

```
{
    if(left>=right)
    {
        return 0;
    }

    if(dp[left][right] != -1)
    {
        return dp[left][right];
    }

    int res = INT_MAX;
    int temp = 0;

    for(int k = left; k<=right-1; k++)
    {
        temp = solve(N,arr,left,k,dp) + solve(N,arr,k+
1,right,dp) + (arr[left-1]*arr[k]*arr[right]);
        res = min(res,temp);
    }

    return dp[left][right] = res;
}

int matrixMultiplication(vector<int> &arr, int N)
{
    vector<vector<int>> dp(N, vector<int> (N,-1));
    return solve(N,arr,1,N-1,dp);
}
```