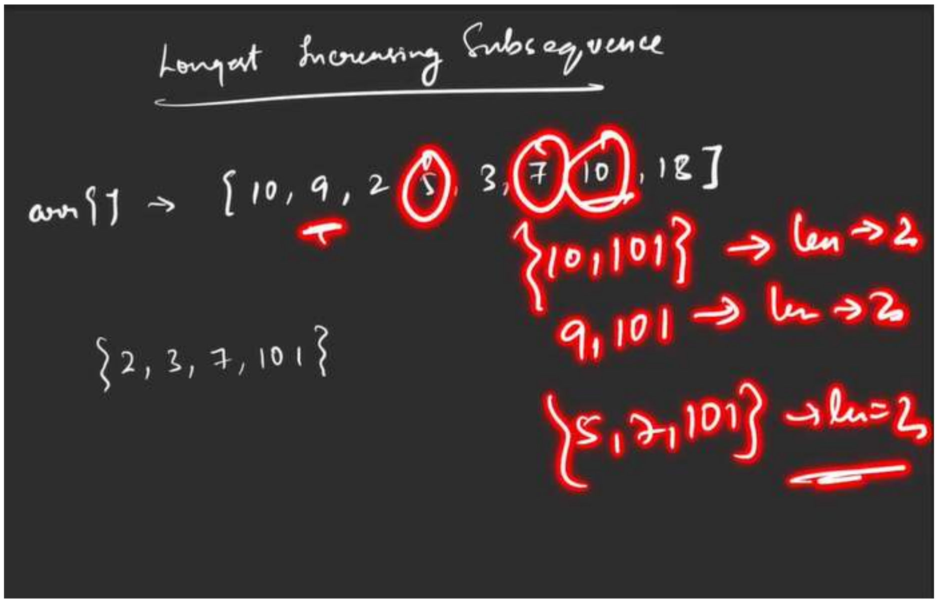
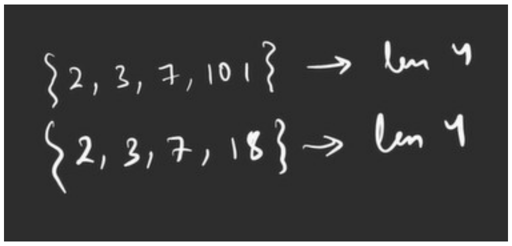
Dynamic Programming

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

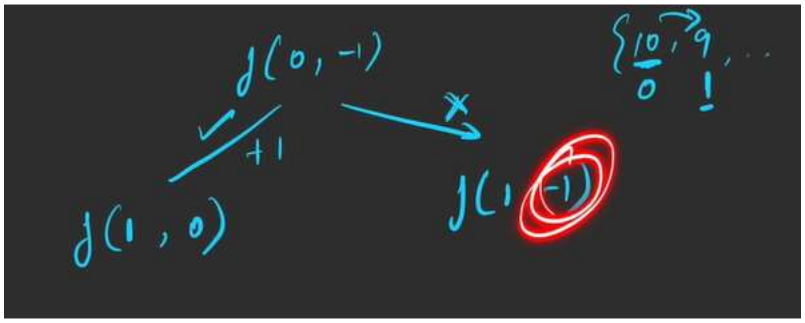
Optimised approach
We can write a recurrence relation
Express everything in terms of index
Explore all possiblities
Take max length of (pick,not pick)

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)

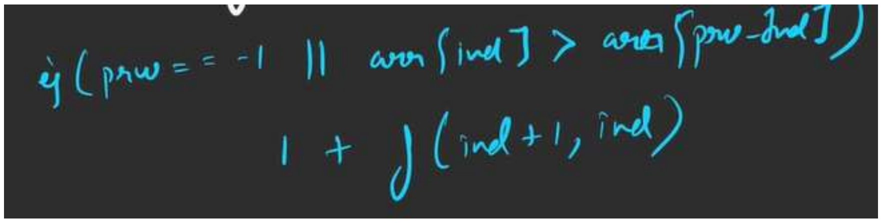
f(3,0) means length of LIS starting from index = 3 whose previous index is 0

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

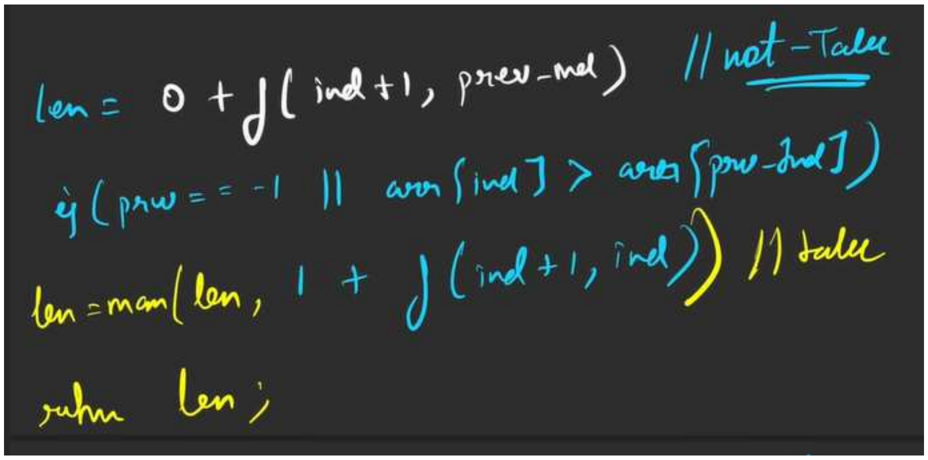


if we are **taking an index**, our previous changes and our length increases but keeping in mind the condition



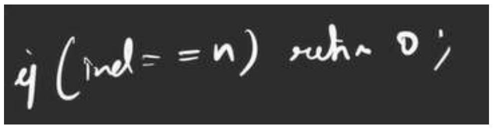
```
if (prev == -1 || arr[ind] > arr[prev_ind])
        1 + f(ind+1, ind)
```

So now max length is maximum of pick/not pick



```
len = 0 + f(ind+1, prev_ind)   // not-Take
  if (prev == -1 || arr[ind] > arr[prev_ind])
len = max(len, 1 + f(ind+1, ind))  // take
return len;
```

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



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

Complexity:

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

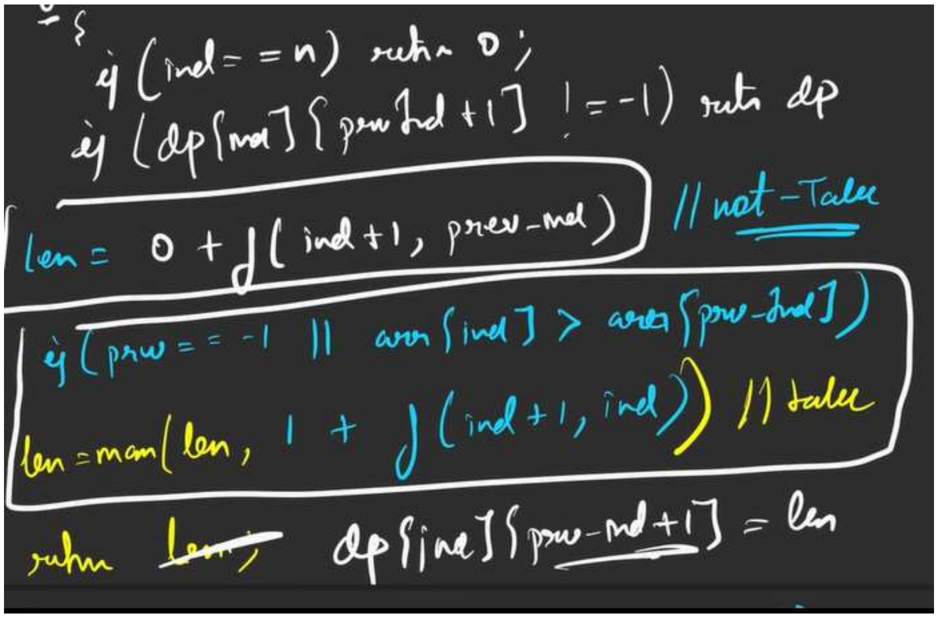
To optimise this, we look for over-lapping sub-problems

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?

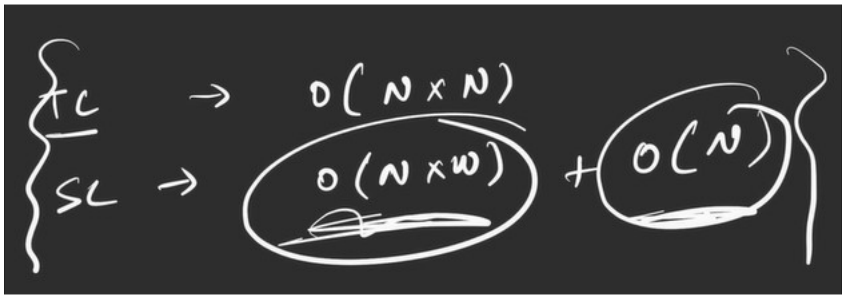
We will do coordinate shifting and shift
-1 -> 0
0 -> 1
1 -> 2
.
.
.
n-1 -> n
So we take an array of [N+1]

So now our code changes to:



```
-{
   if (ind == n) return 0;
   if (dp[ind][prevInd+1] != -1) return dp
len = 0 + f(ind+1, prev_ind)  // not-Take
if (prev == -1 || arr[ind] > arr[prev_ind])
len = max(len, 1 + f(ind+1, ind))  // take
return ~~len;~~  dp[ind][prev_ind+1] = len
```

Complexity changes to


$$\left\{\begin{array}{l} TC \rightarrow O(N \times N) \\ SC \rightarrow O(N \times N) + O(N) \end{array}\right\}$$

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);
}
```

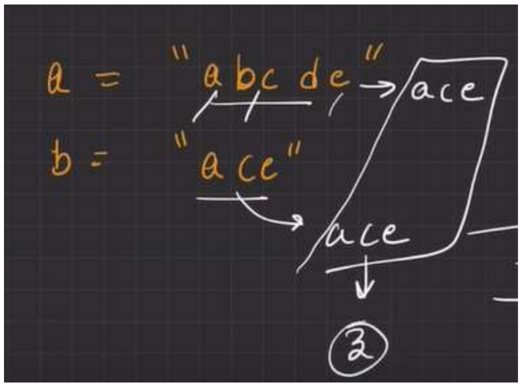
Memoization code

```
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);
}
```

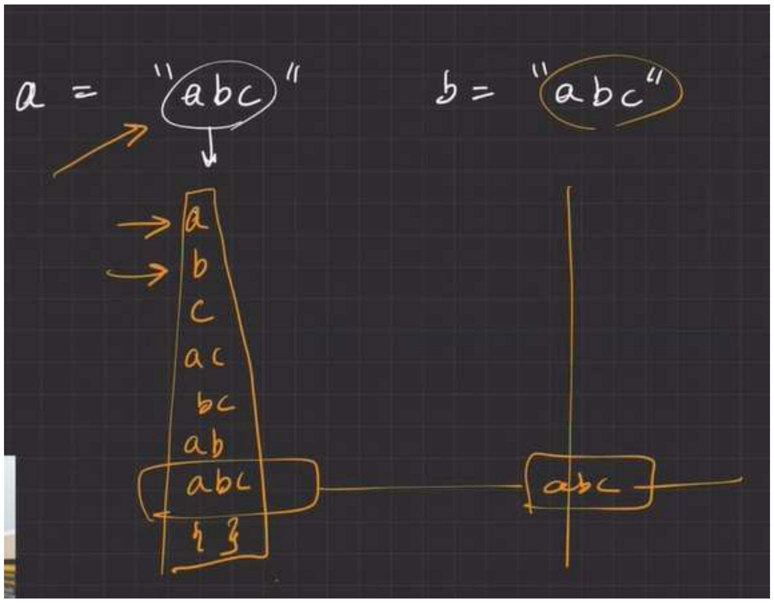
Longest Common Subsequence

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.

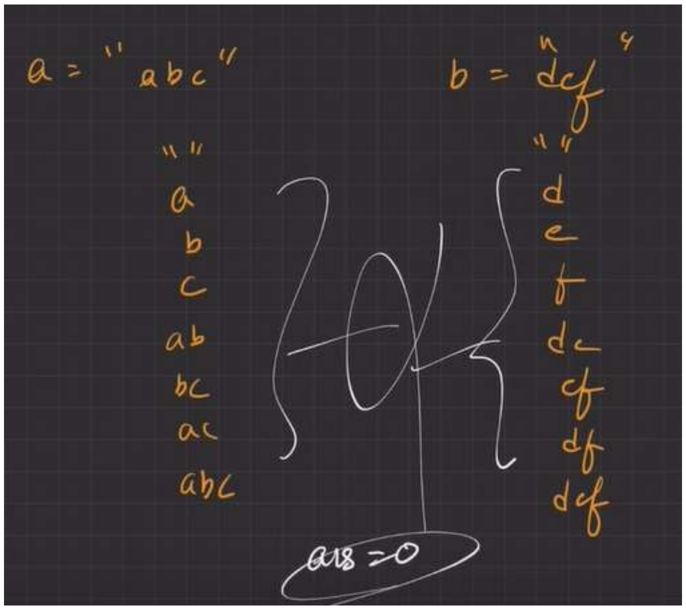
Like in below example:
 "ace" is common in both strings with same relative ordering
So we return length = 3



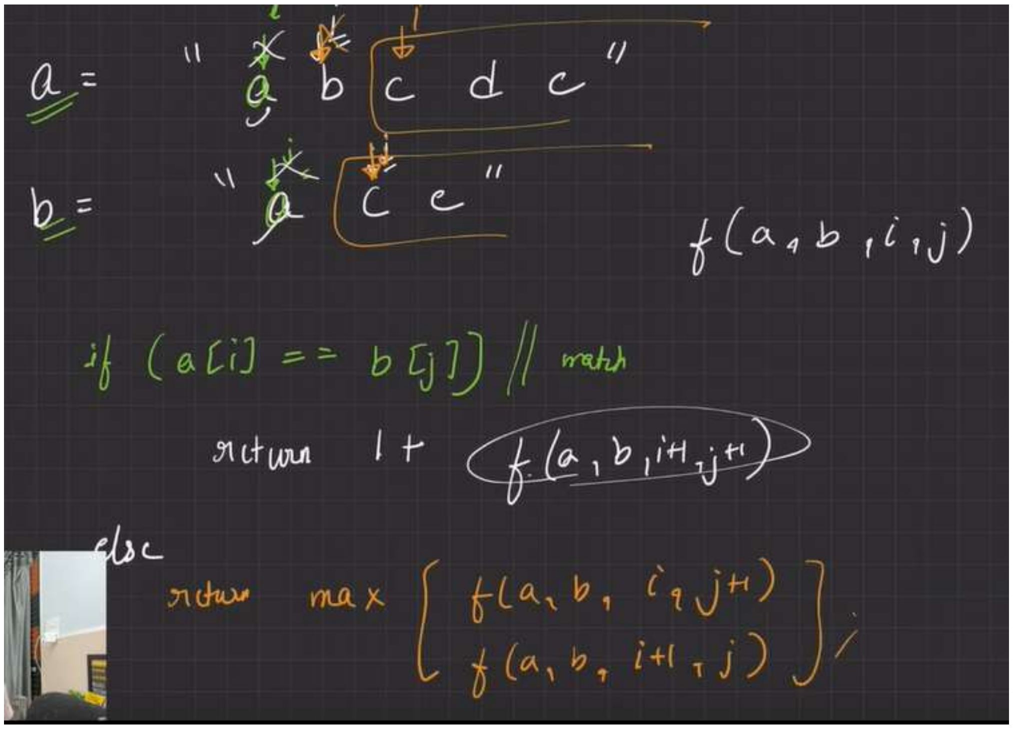
Another example:



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

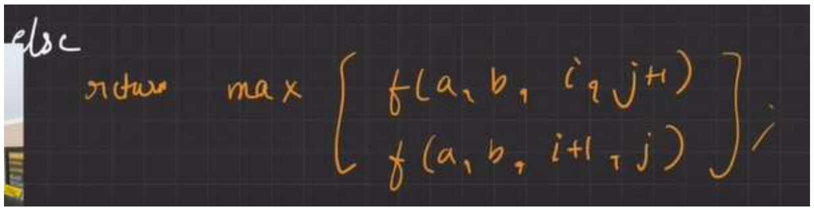


We will use recursion and put i and j in both string
Is arr1[i] == arr2[j] // match
Return 1 + f(arr1,arr2,i+1,j+1)



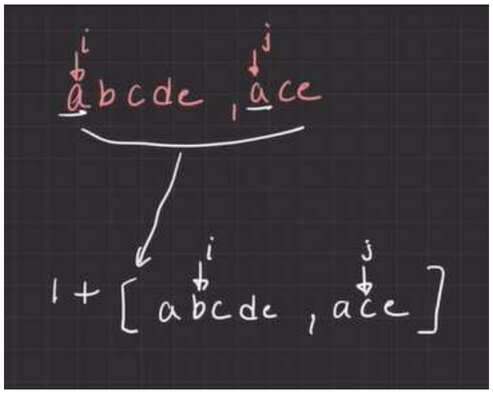
```
a = "a b c d e"
b = "a c e"
f(a, b, i, j)

if (a[i] == b[j]) // match
    return 1 + f(a, b, i+1, j+1)
else
    return max { f(a, b, i, j+1)
                 f(a, b, i+1, j) };
```

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

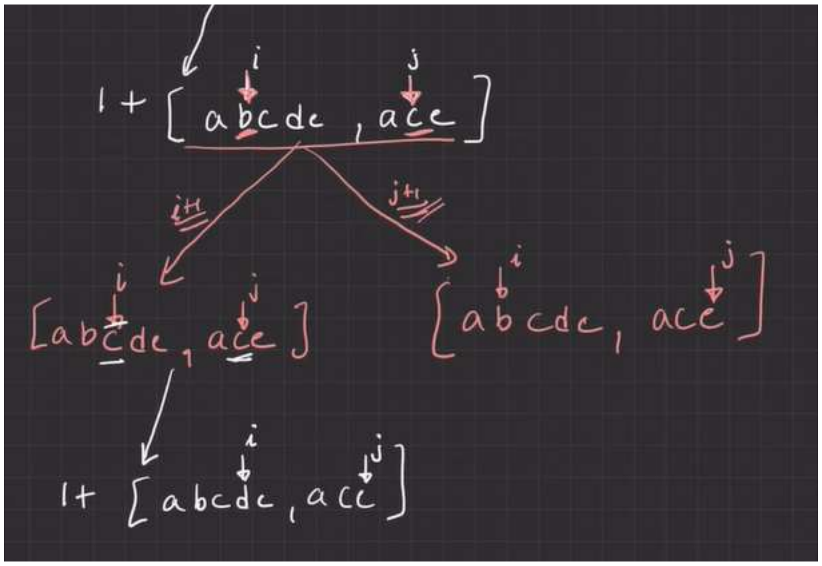


```
else
    return max { f(a, b, i, j+1)
                 f(a, b, i+1, j) };
```

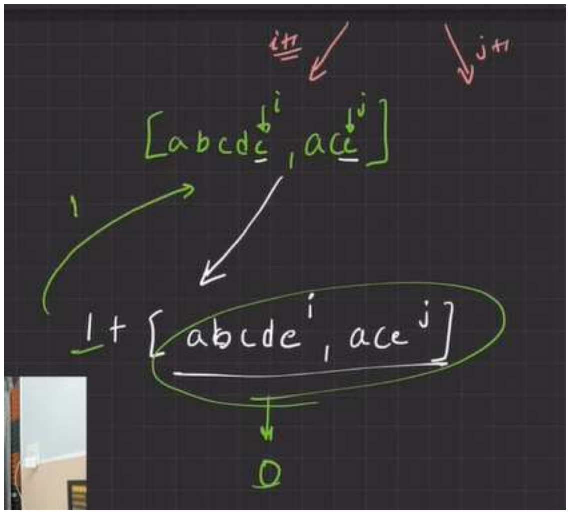
Working Tree



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);
}
```

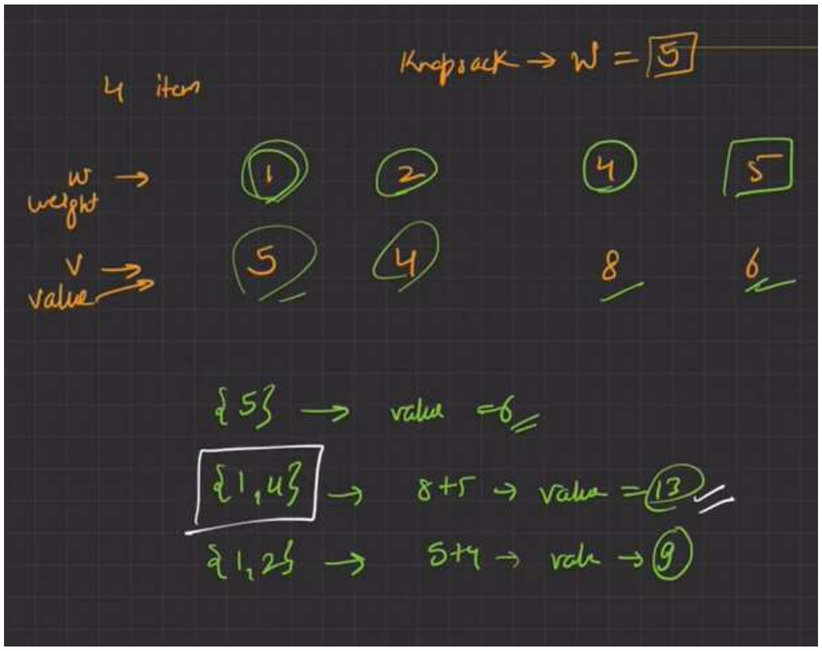
Bottom-up DP (Tabulation)

```
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

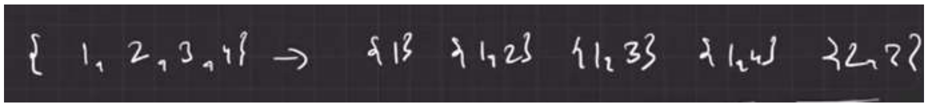
A thief has a bag which can carry only 'W' weight, he has to theft some items such that the weight is maximum and within "W".

Example:

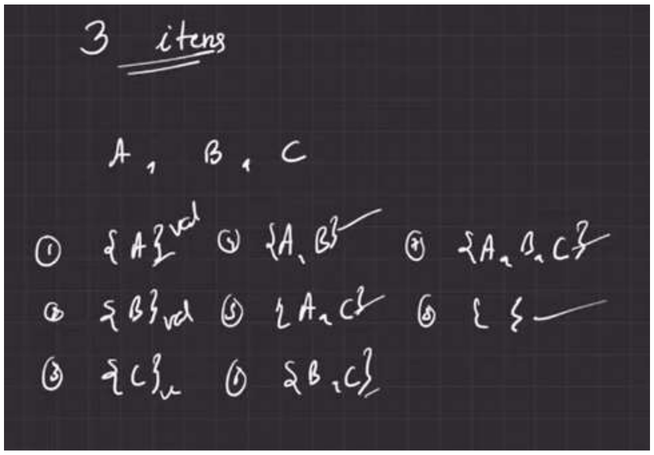


First approach (Brute force)

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

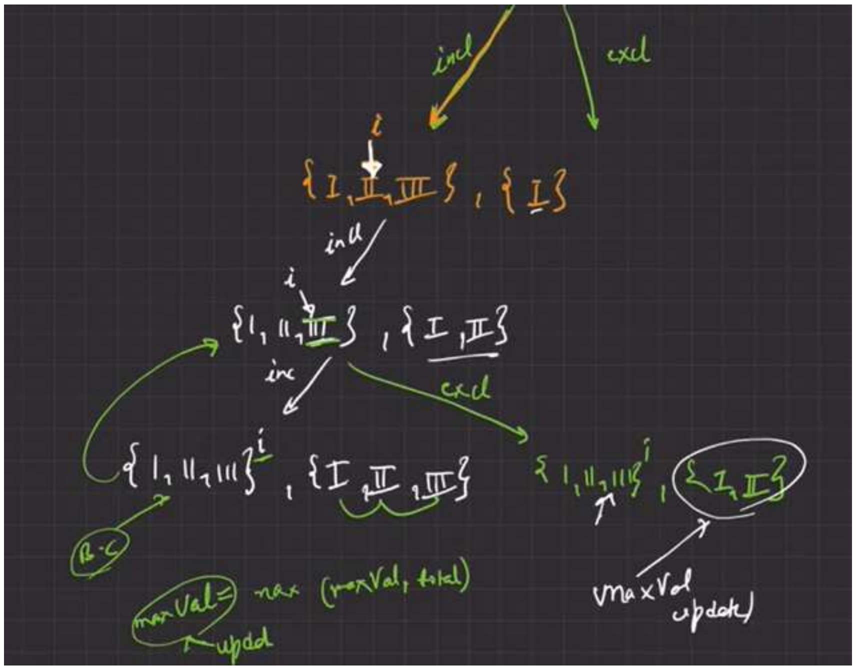


We take our combination of all weight/values and return maximum value out of it.
Let say we have 3 items A,B,C so I can have 8 combinations.



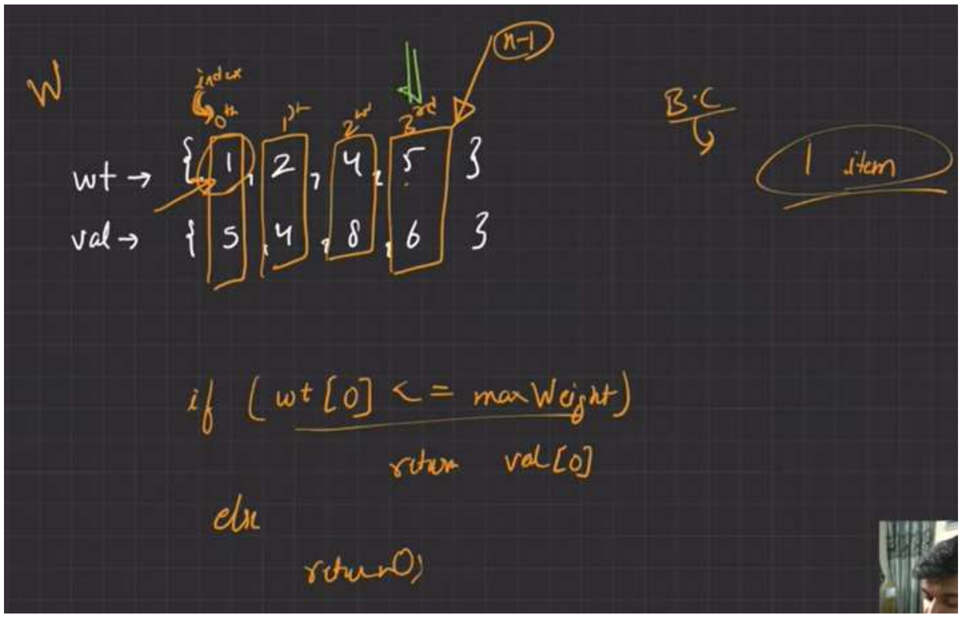
So we can use (include / not include) approach using recursion

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);
}
```

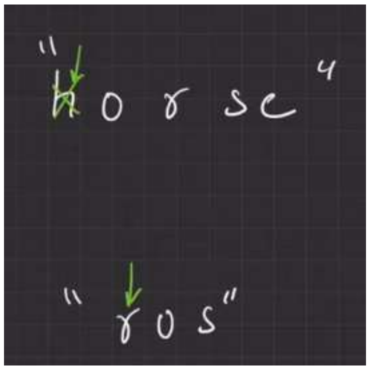
Edit Distance

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:

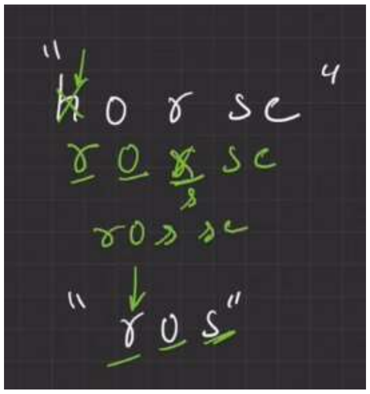
1. Insert a character
2. Delete a character
3. Replace a character

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"

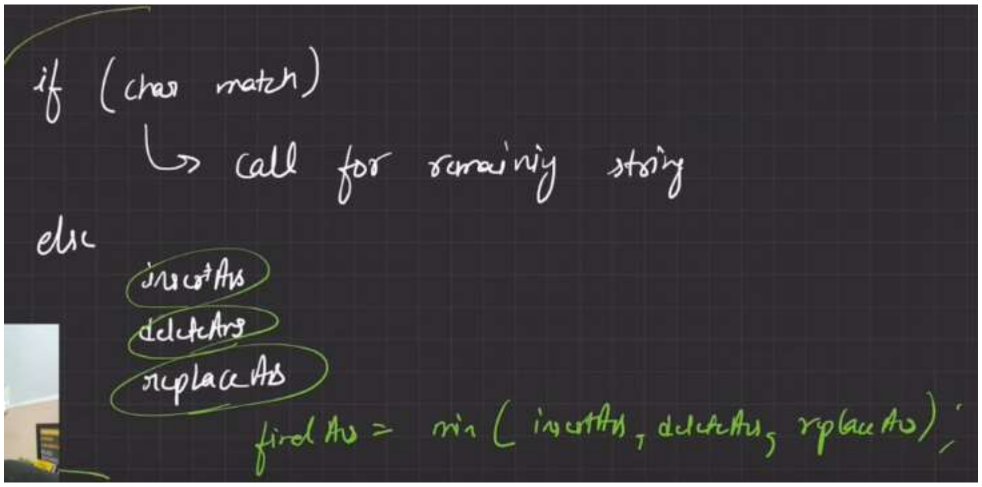


Now ros are mathced in both, delete 's' and 'e' from horse
So it took 4 operations replace,replace,delete,delete to comvert horse to ros

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)



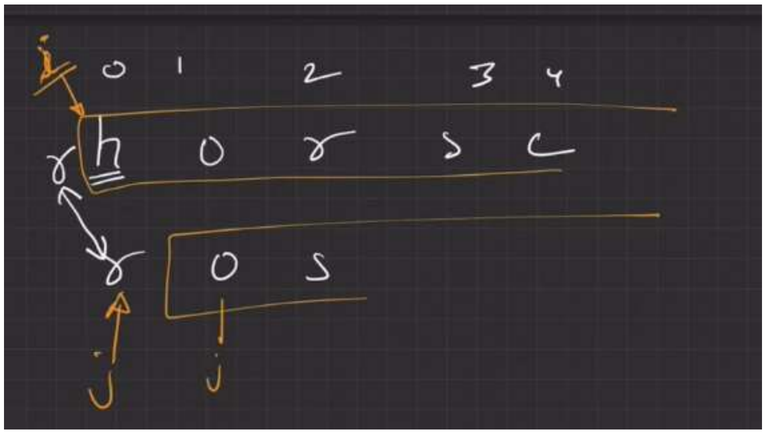
```
if (char match)
    ↳ call for remaining string
else
    insertAns
    deleteAns
    replaceAns
        finalAns = min(insertAns, deleteAns, replaceAns);
```

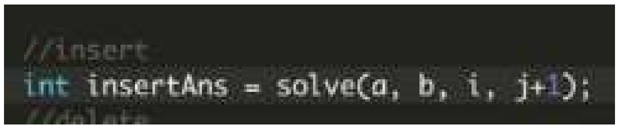
Base case:

 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)

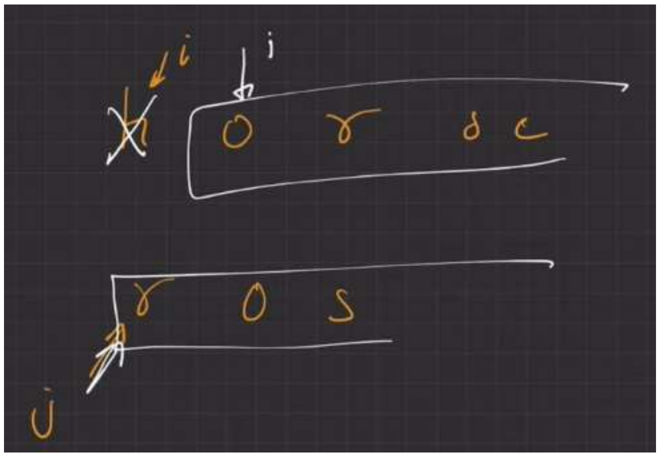
In our insert case, i will remain at its place as we are inserting string2[j] in string1 so i remains as it is, j becomes j + 1

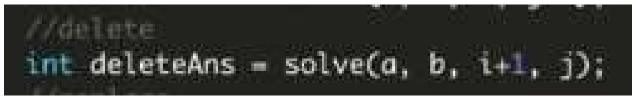




```
//insert
int insertAns = solve(a, b, i, j+1);
```

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

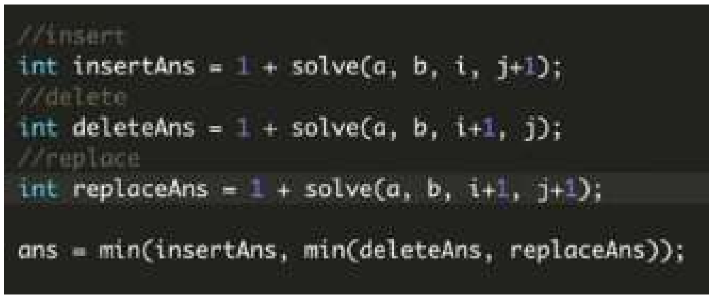




```
//delete
int deleteAns = solve(a, b, i+1, j);
```

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)

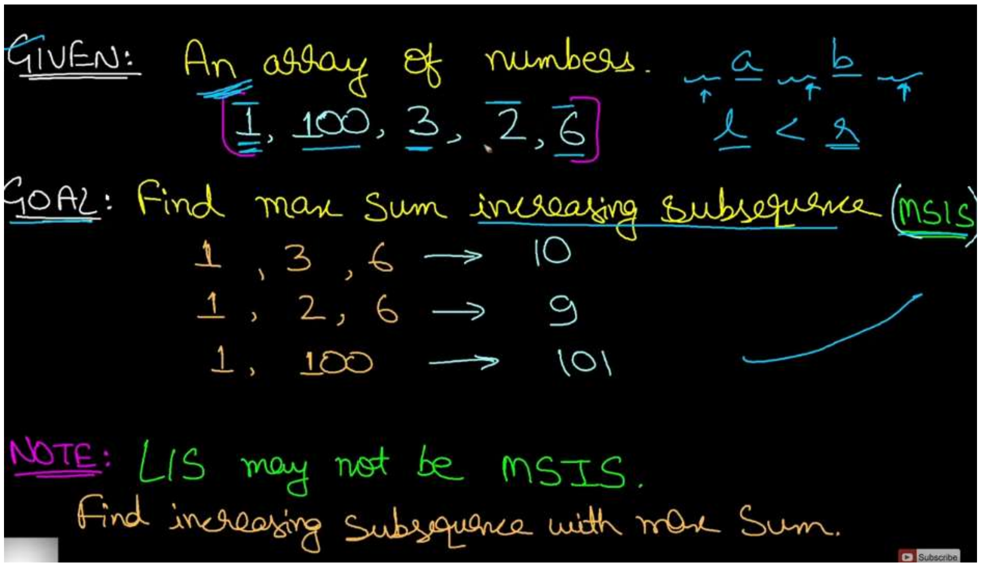
```
int solveTabulation(string a, string b)
    {
        // make dp array
        vector<vector<int>> dp(a.length()+
1, vector<int> (b.length()+1,0));

        // 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

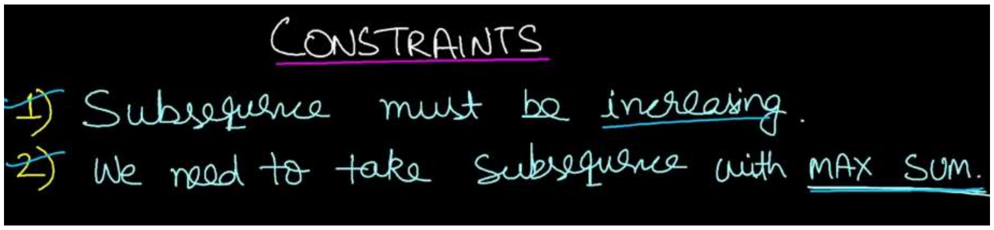


GIVEN: An array of numbers. a < b
[1, 100, 3, 2, 6]

GOAL: Find max Sum increasing subsequence (MSIS)

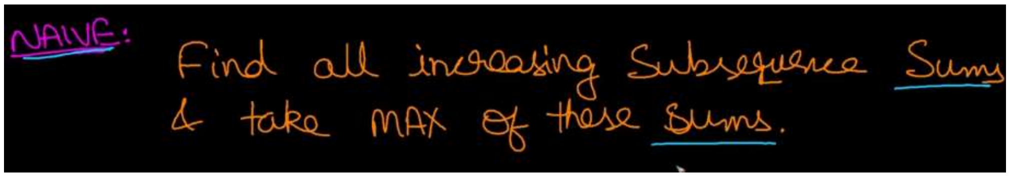
1 , 3 , 6 → 10
1 , 2 , 6 → 9
1 , 100 → 101

NOTE: LIS may not be MSIS.
Find increasing subsequence with max Sum.



CONSTRAINTS
1) Subsequence must be increasing.
2) We need to take subsequence with MAX SUM.

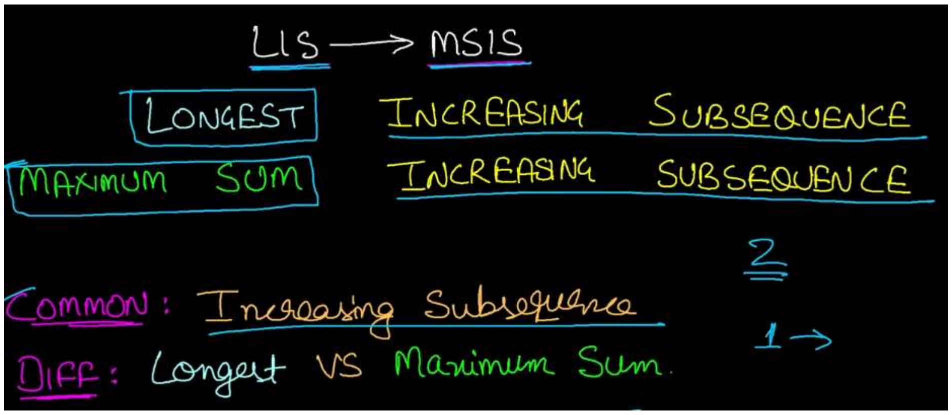
Naïve Approach gives TLE:



NAIVE: Find all increasing Subsequence Sums & take MAX of these sums.

Optimised Approach

Here also we need an increasing subsequence


LIS → MSIS

LONGEST INCREASING SUBSEQUENCE

MAXIMUM SUM INCREASING SUBSEQUENCE

2

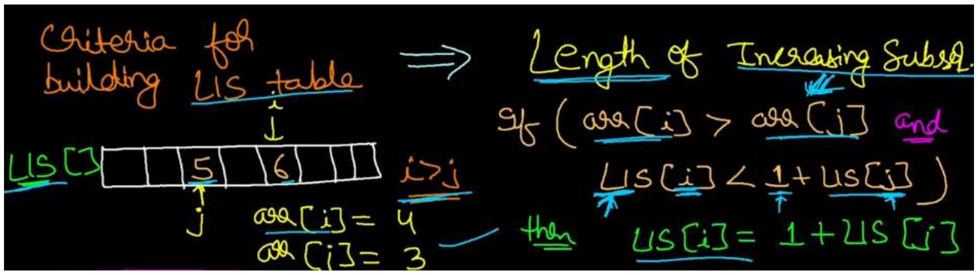
COMMON: Increasing Subsequence

DIFF: Longest VS Maximum Sum.

1 →

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

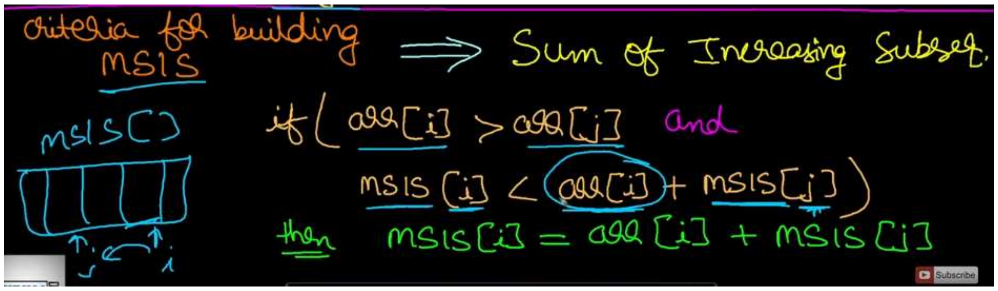
Criteria we followed for LIS:
Keeping in mind, i pointer is ahead j pointer
If arr[i] > arr[j] means increasing sequence &&
LIS[i] < 1 + LIS[j] means length is more than LIS[j], then we have one more longest increasing subsequence so store it.
We are doing +1 above because including an element means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


Criteria for building LIS table ⟹ Length of Increasing Subseq.

if (arr[i] > arr[j] and
LIS[i] < 1 + LIS[j])
then LIS[i] = 1 + LIS[j]

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


Criteria for building MSIS ⟹ Sum of Increasing Subseq.

if(arr[i] > arr[j] and
msis[i] < arr[i] + msis[j])
then msis[i] = arr[i] + msis[j]

TC: O(n^2)

Code:

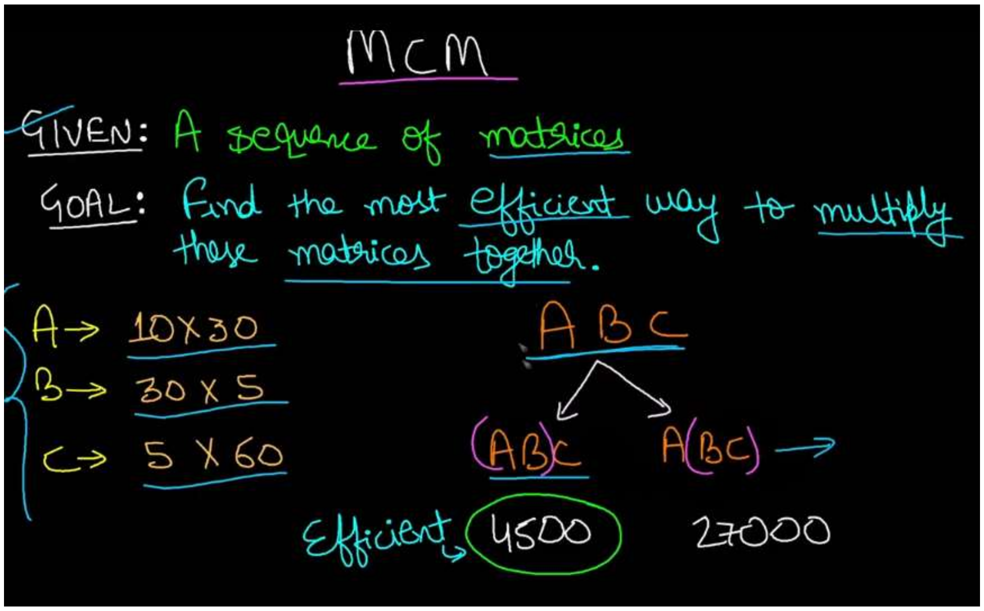
```
int solve(vector<int> &arr, int n)
{
    int maxi = 0;
    int MSIS[n];

    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);
}
```

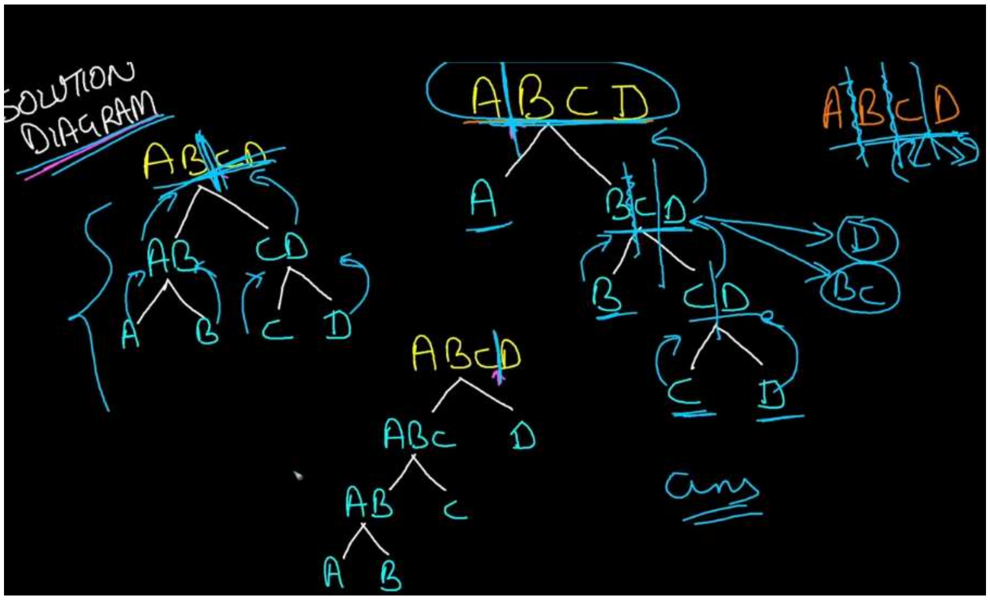
Matrix Chain Multiplication

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 **i**th matrix has the dimensions **(arr[i-1] x**

arr[i]).

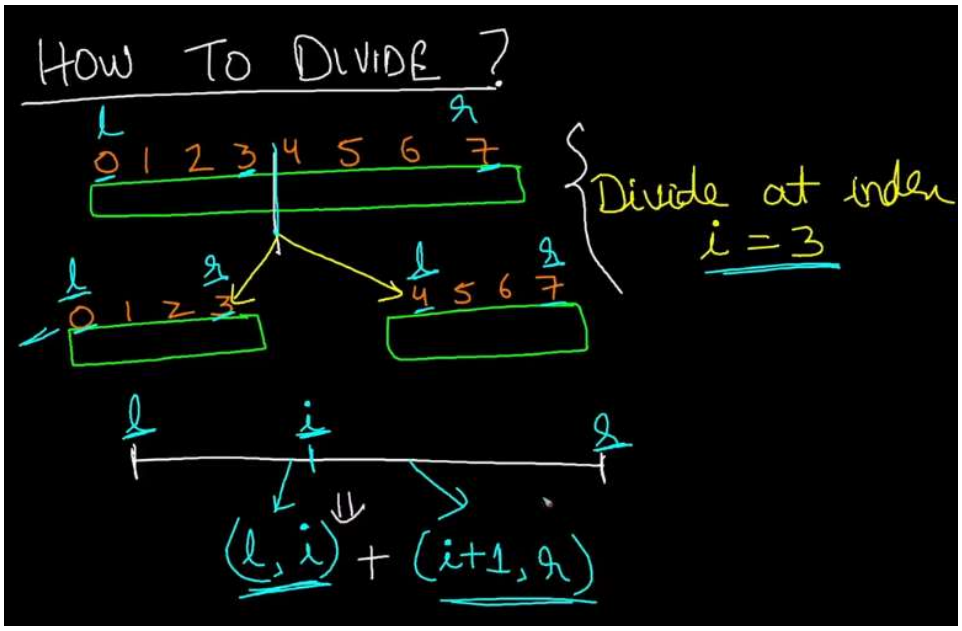
Approach to MCM Problems


MCM
GIVEN: A sequence of matrices
GOAL: Find the most efficient way to multiply these matrices together.
A → 10X30
B → 30 X 5
C → 5 X 60
A B C
(AB)C
A(BC)
Efficient
4500
27000

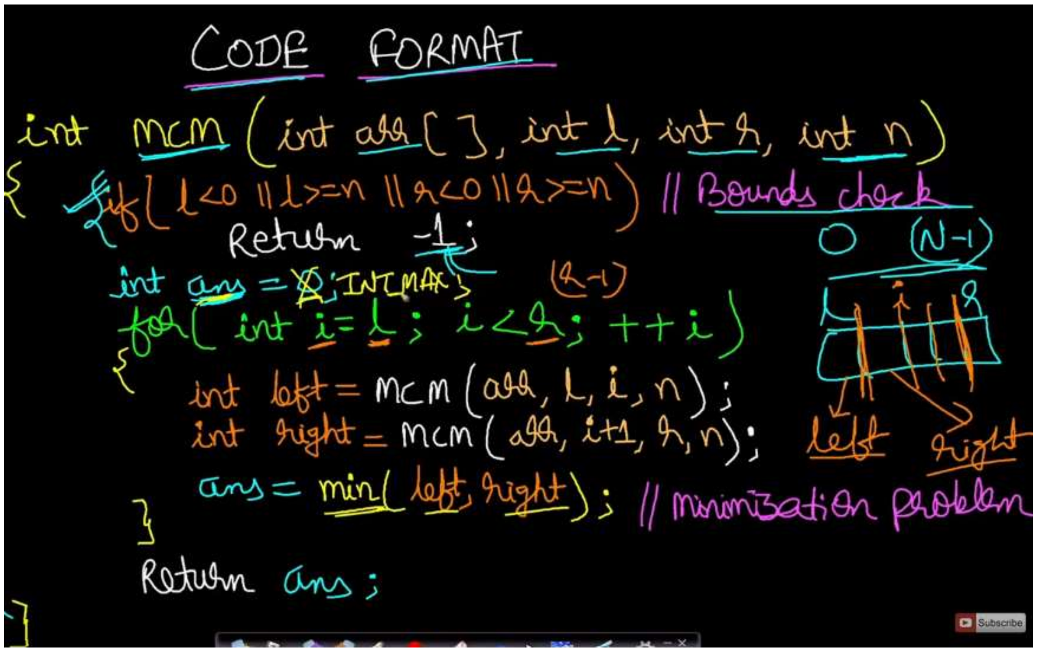


For partitioning there must be a left limit and right limit between which our division will take place.
We can solve each partition using recursion and then we can use max/min according to our answer and consider the max/min answer while returning in recursion based on our problem statement

What is the idea behind division?



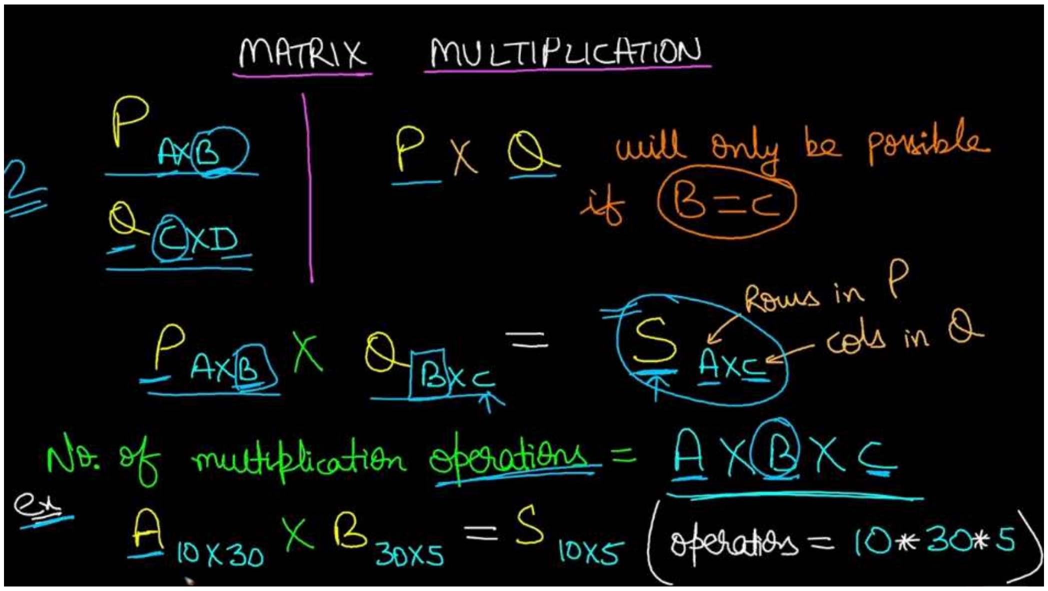
Code Format:
To solve the divided array we can use below code.
But, we need to do division from many parts to explore all the possiblities.


CODE FORMAT

```
int mcm(int arr[], int l, int r, int n)
{
  if(l<0 || l>=n || r<0 || r>=n) // Bounds check
      Return -1;
  int ans = ~~0~~ INTMAX;
  for( int i=l; i<r; ++i)
  {
      int left = mcm(arr, l, i, n);
      int right = mcm(arr, i+1, r, n);
      ans = min(left, right); // minimization problem
  }
  Return ans;
}
```

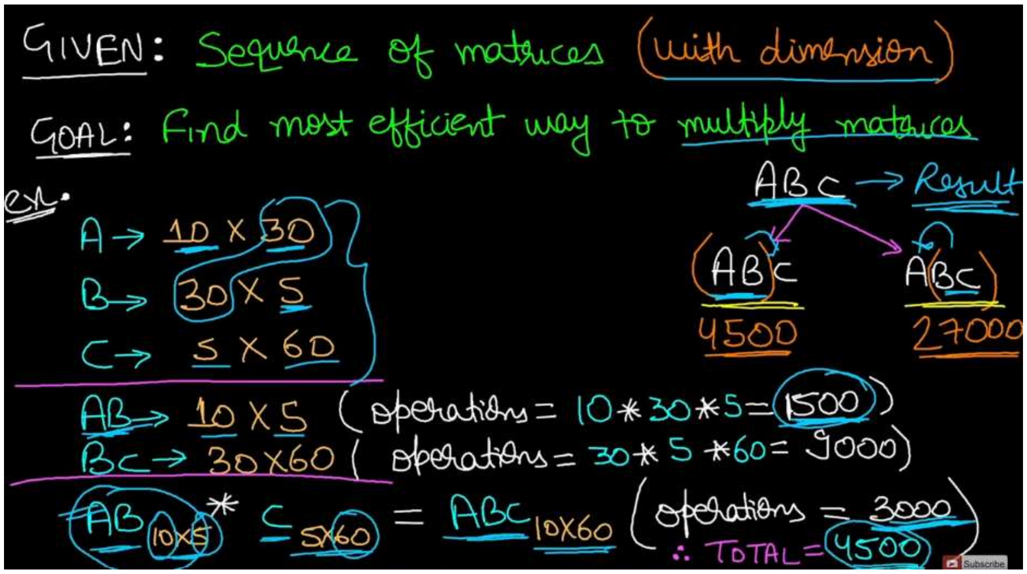
Actual Solution starts from here

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


MATRIX MULTIPLICATION
$P_{A \times B}$
$Q_{C \times D}$
P X Q will only be possible if B=C
$P_{A \times B} \times Q_{B \times C} = S_{A \times C}$
Rows in P
cols in Q
No. of multiplication operations = A X B X C
ex
$A_{10 \times 30} \times B_{30 \times 5} = S_{10 \times 5}$ (operations = 10*30*5)

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

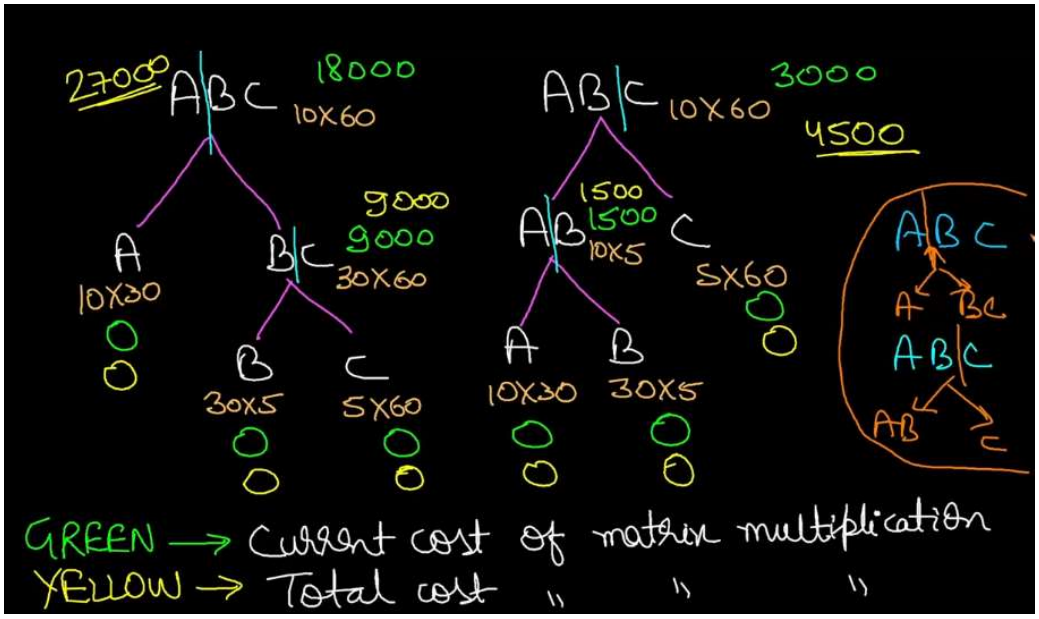

GIVEN: Sequence of matrices (with dimension)
GOAL: Find most efficient way to multiply matrices
ex.
A → 10 X 30
B → 30 X 5
C → 5 X 60
ABC → Result
(AB)C
4500
A(BC)
27000
AB → 10 X 5 (operations = 10*30*5 = 1500)
BC → 30 X 60 (operations = 30*5*60 = 9000)
$AB_{10 \times 5} * C_{5 \times 60} = ABC_{10 \times 60}$ (operations = 3000
∴ TOTAL = 4500)

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

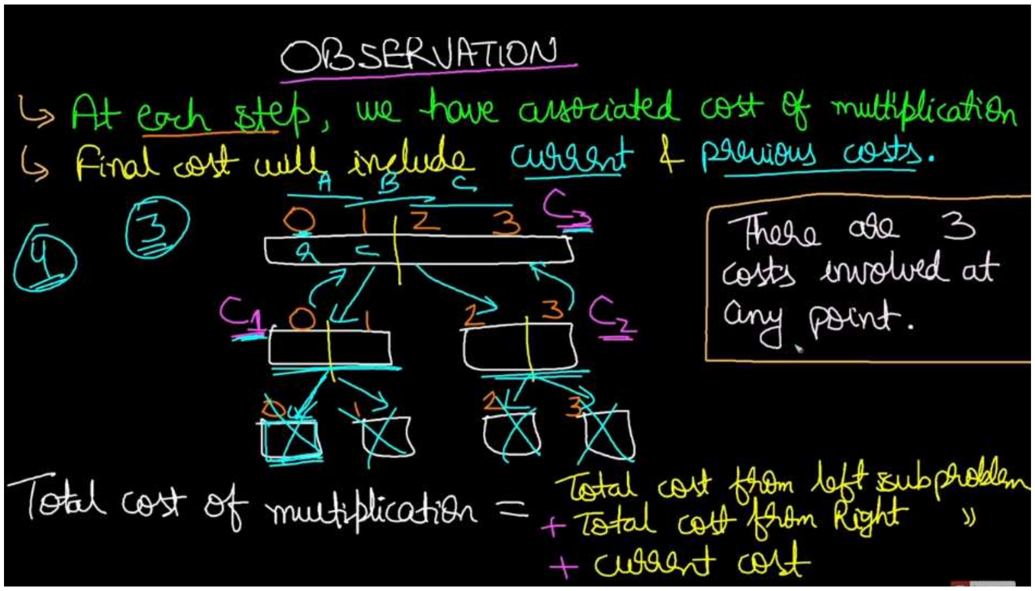
Approach

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

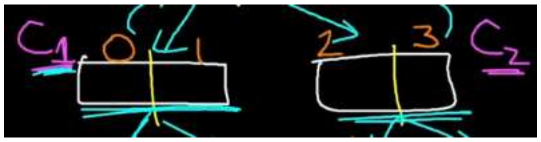


Observations

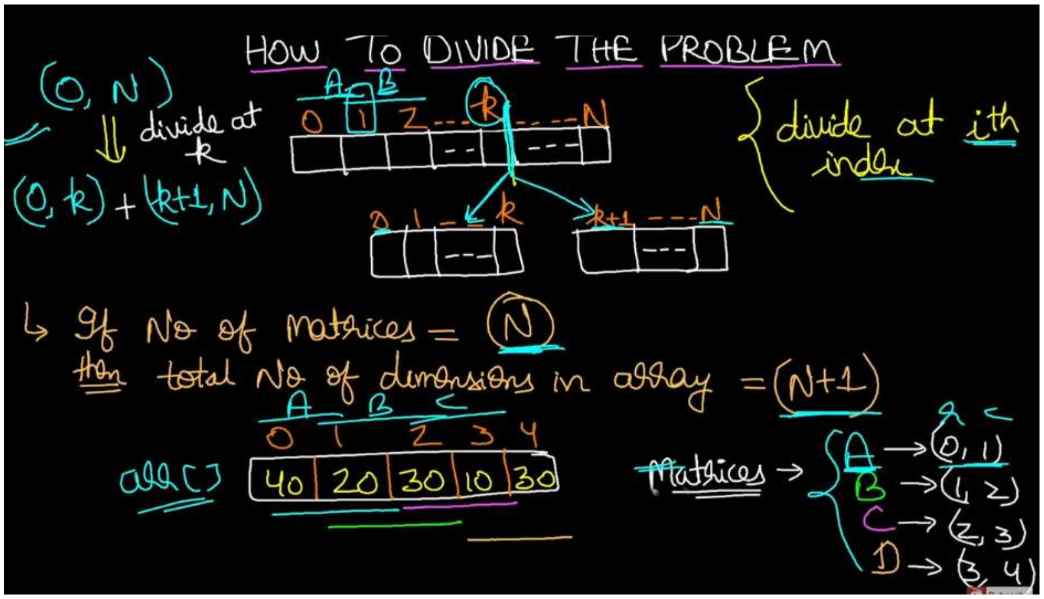


OBSERVATION
↳ At each step, we have associated cost of multiplication
↳ Final cost will include current & previous costs.
There are 3 costs involved at any point.
C3 C1 C2
0 1 2 3
Total cost of multiplication = Total cost from left subproblem + Total cost from Right " + current cost

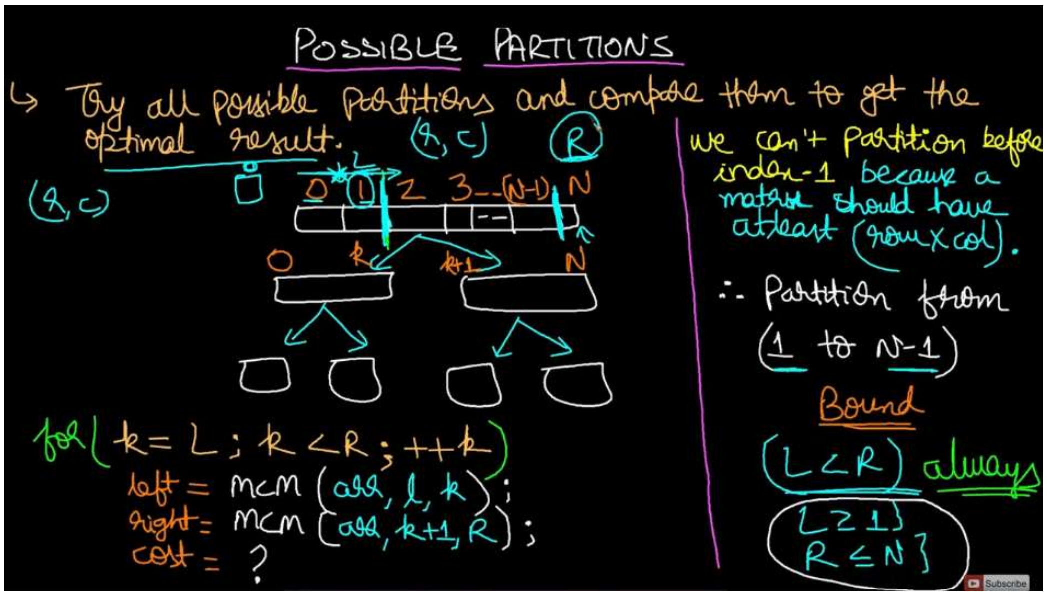
Our Leaf node will look like



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

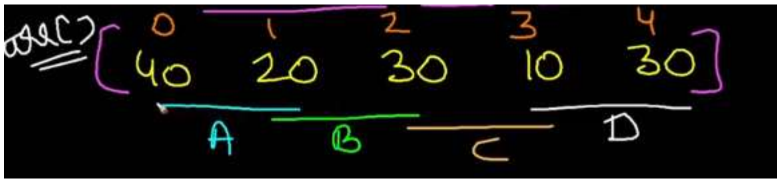

HOW TO DIVIDE THE PROBLEM
(0, N) ⇓ divide at k (0, k) + (k+1, N)
0 1 2 --- k --- N
divide at ith index
↳ If No of matrices = N then total No of dimensions in array = (N+1)
A B C
0 1 2 3 4
arr[] 40 20 30 10 30
Matrices → A → (0, 1) B → (1, 2) C → (2, 3) D → (3, 4)

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

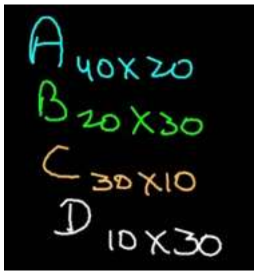

POSSIBLE PARTITIONS
↳ Try all possible partitions and compare them to get the optimal result.
(r, c) R
0 1 2 3 --(N-1) N
0 k k+1 N
for(k = L ; k < R ; ++k)
left = mcm (arr, l, k);
right = mcm (arr, k+1, R);
cost = ?
we can't partition before index-1 because a matrix should have atleast (row x col).
∴ Partition from (1 to N-1)
Bound
(L < R) always
{L ≥ 1, 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

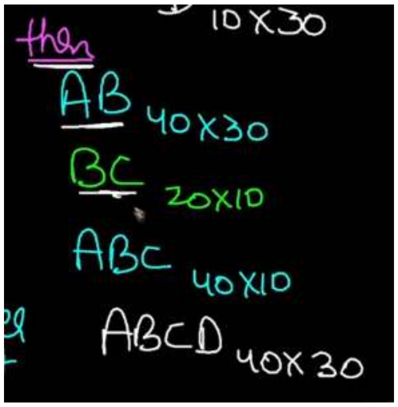
A, B, C, D, are our 4 matrices. Whose dimensaions are shown at indexes

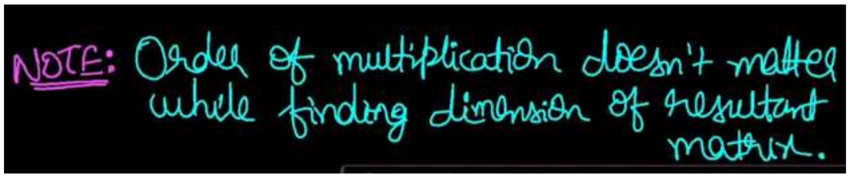


Our matrices are like:

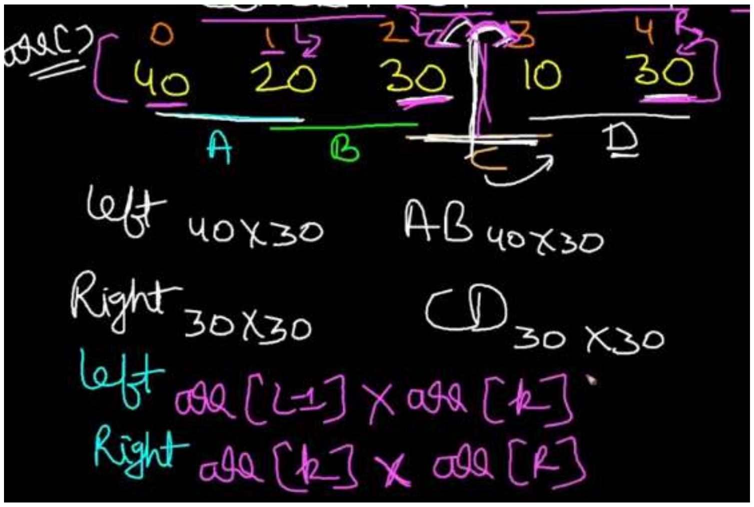


We can only multiply 2 matrices if there column number is same.
For ABC, we check number of column of AB and number of column of C, if they are same. Then, **ABC has number of rows = number of rows of AB and number of column = number of column of C**

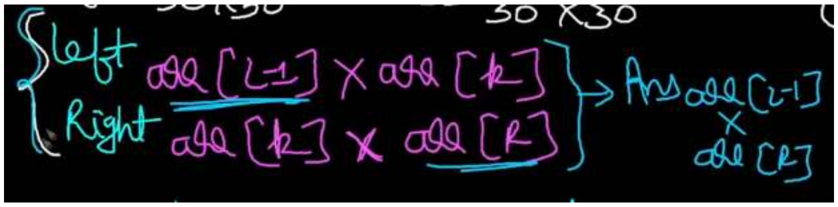



NOTE: Order of multiplication doesn't matter while finding dimension of resultant matrix.

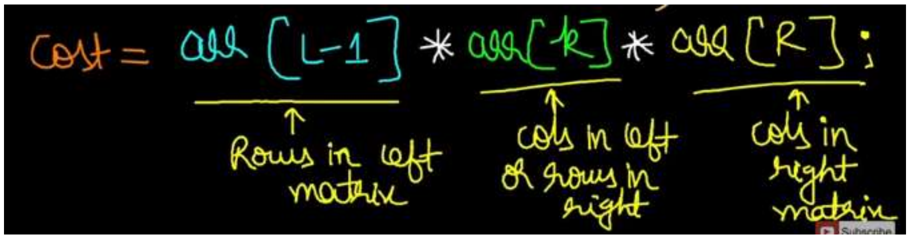
So, Dimensions of left partition result will be arr[L-1] X arr[K]
Dimensions of right partition result will be arr[K] X arr[R]



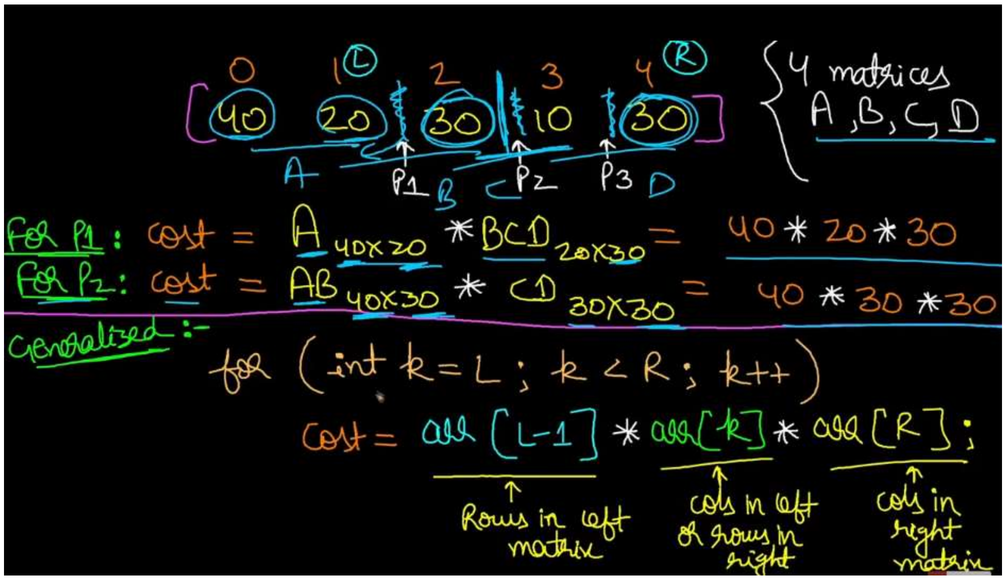
Resultans Answer will have dimension arr[L-1] X arr[R]



Cost will be:

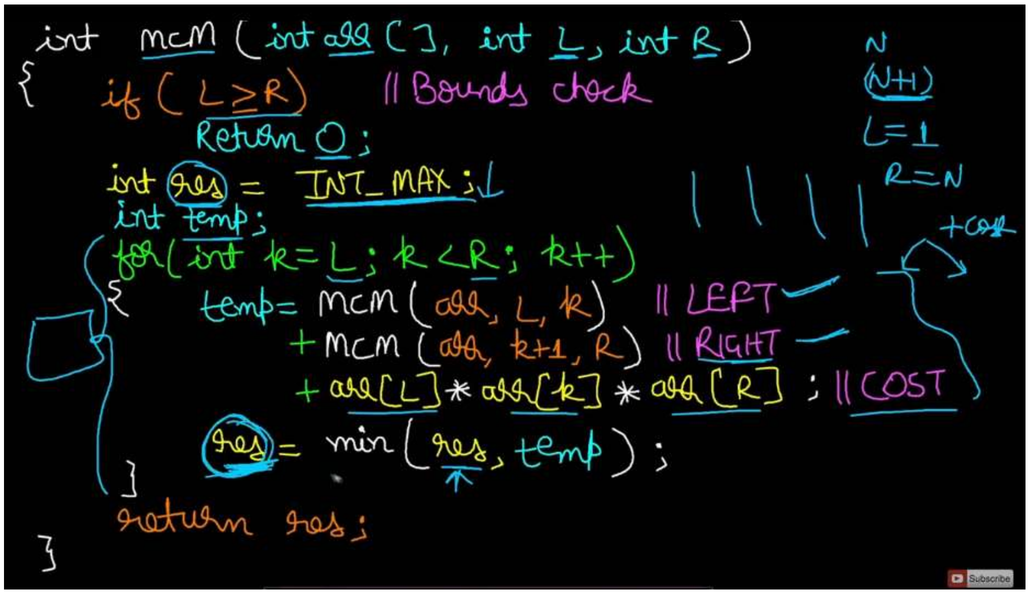

Cost = arr[L-1] * arr[k] * arr[R];
Rows in left matrix
cols in left or rows in right
cols in right matrix

Pseudo-Code:


0 1 L 2 3 4 R
[40 20 30 10 30]
A P1 B P2 C P3 D
4 matrices A, B, C, D
For P1 : cost = A 40x20 * BCD 20x30 = 40 * 20 * 30
For P2 : cost = AB 40x30 * CD 30x30 = 40 * 30 * 30
Generalized :-
for (int k = L; k < R; k++)
Cost = arr[L-1] * arr[k] * arr[R];
Rows in left matrix
cols in left or rows in right
cols in right matrix

Code:

L = 1, R = N initially



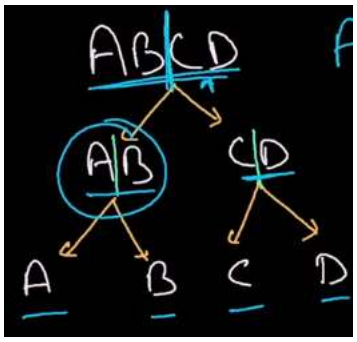
```
int mcm(int arr[], int L, int R)
{   if(L>=R)    // Bounds check
        Return 0;
    int res = INT_MAX;
    int temp;
    for(int k=L; k<R; k++)
    {   temp= mcm(arr, L, k)   // LEFT
            + mcm(arr, k+1, R)  // RIGHT
            + arr[L]*arr[k]*arr[R]; // COST
        res = min(res, temp);
    }
    return res;
}
```

Code (Recursive) (gives TLE):

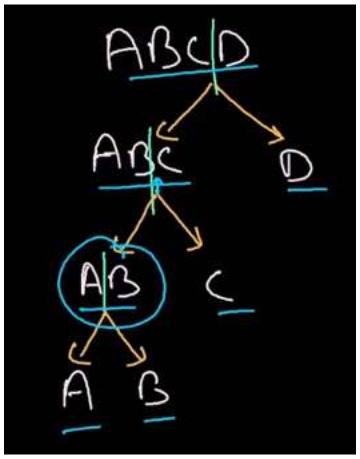
```
int solve(int N, int arr[], int left, int right)
{
    if(left>=right)
    {
        return 0;
    }

    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

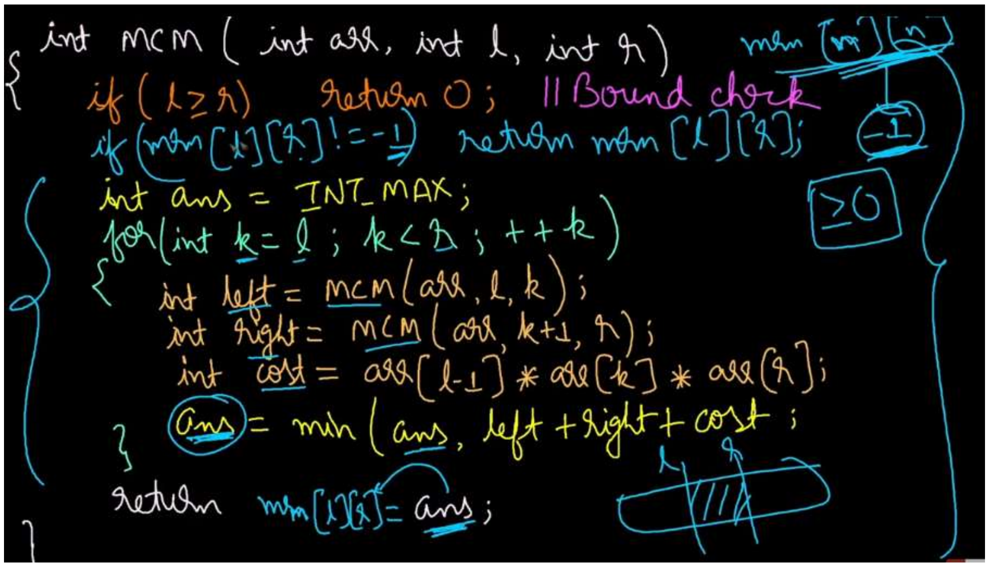


If we partition from ABC | D



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 MCM ( int arr, int l, int r)
{
    if (l ≥ r)   return 0;   // Bound check
    if (mem[l][r] != -1)   return mem[l][r];
    int ans = INT_MAX;
    for(int k = l ; k < r ; ++k)
    {
        int left = MCM(arr, l, k);
        int right = MCM(arr, k+1, r);
        int cost = arr[l-1] * arr[k] * arr(r];
        ans = min(ans, left + right + cost ;
    }
    return mem[l][r] = ans;
}
```

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

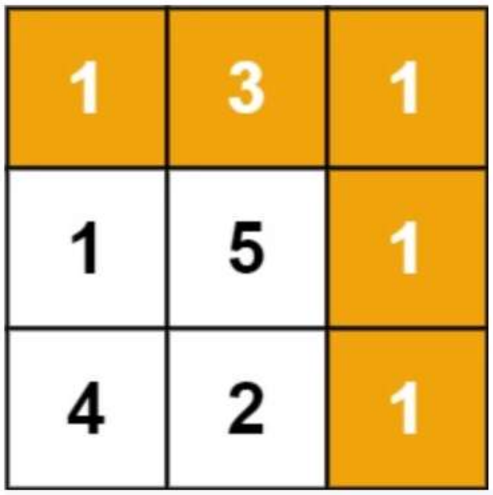
```
{
    if(left>=right)
    {
        return 0;
    }

    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);
}
```

Minimum Path Sum

Given a n x m grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can **only move either down or right** at any point in time.

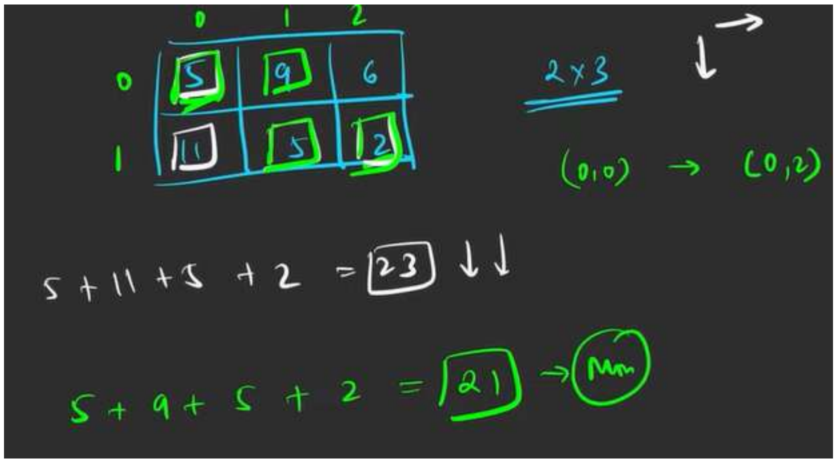
Example 1:



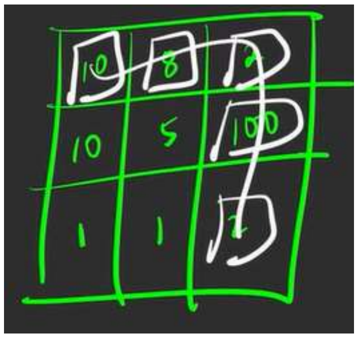
1	3	1
1	5	1
4	2	1

```
Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.
```

Approach



Why can't we follow Greedy?
In below Grid greedy would follow this path



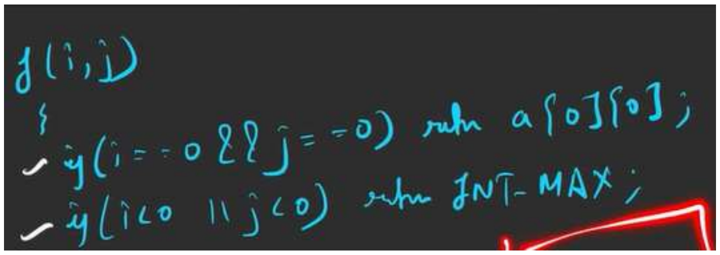
Whereas there are other paths also which are less expensive, so we cannot use Greedy
Values are not uniform, they are not only increasing or only decreasing
When there is no uniformity we should not apply greedy

Brute force
Tell me all paths, return the one with minimum sum
Steps to write a recurrence relation

1. Express in terms of i , j
2. Explore all paths
3. Take the minimum path

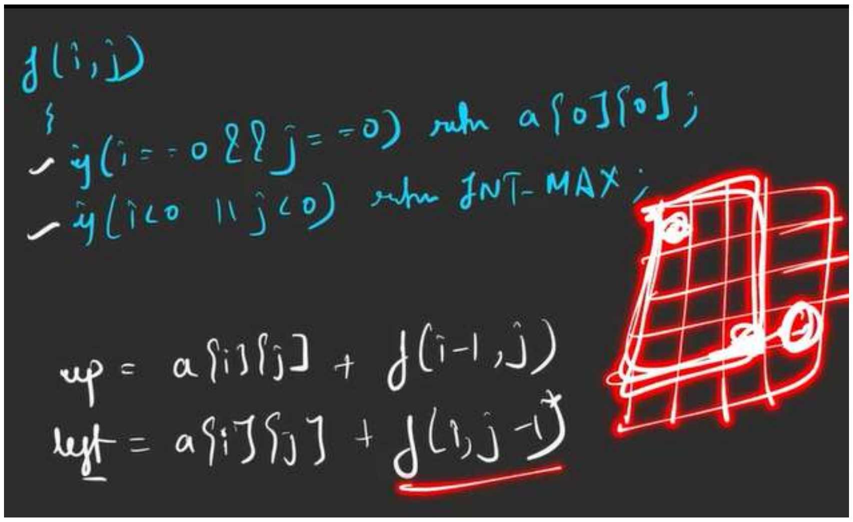
Our recursion should give us minimum cost to reach (0,0) -> (i,j)
We will start from f(n-1, m-1)

Base case
If we reach our destination, we return the element
If we reach out of bound, we return INT_MAX as we need minimum value and if there is any out of bound index, we do not need to take it.



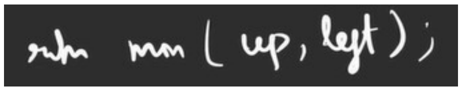
```
f(i,j)
{
✓ if(i==0 && j==0) return a[0][0];
✓ if(i<0 || j<0) return INT_MAX;
```

Now we go up or left



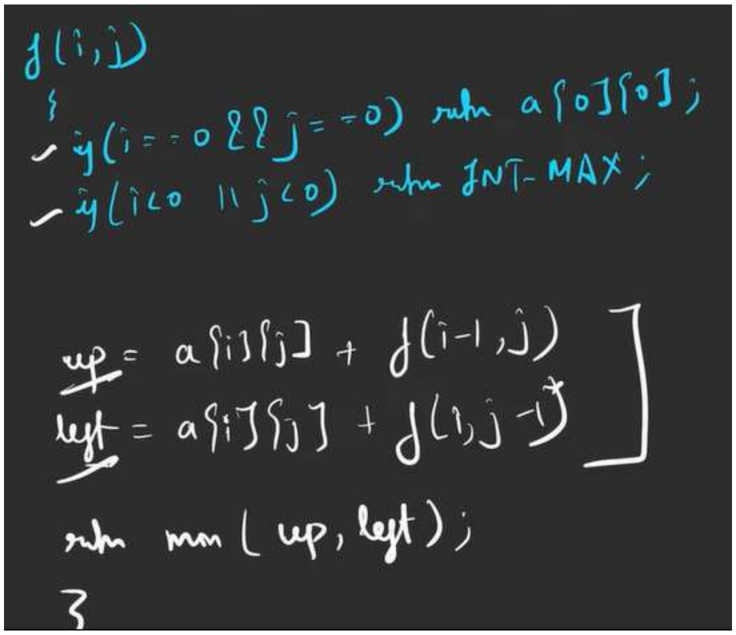
```
f(i,j)
{
✓ if(i==0 && j==0) return a[0][0];
✓ if(i<0 || j<0) return INT_MAX;

  up = a[i][j] + f(i-1,j)
  left = a[i][j] + f(i,j-1)
```

Take minimal of both of them.



```
return min(up, left);
```

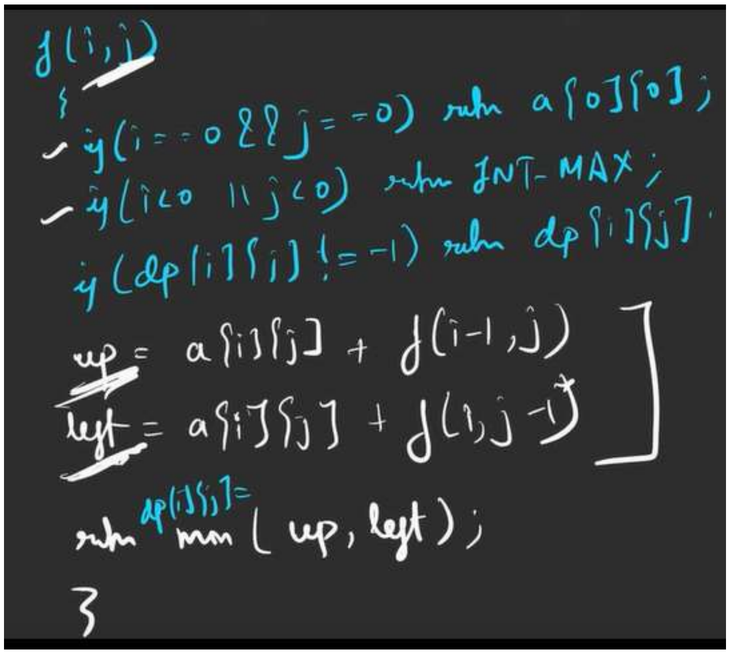
Our Peudocode looks like:



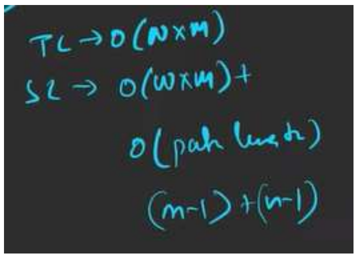
```
f(i,j)
{
  if(i==0 && j==0) return a[0][0];
  if(i<0 || j<0) return INT_MAX;

  up = a[i][j] + f(i-1,j)
  left = a[i][j] + f(i,j-1)

  return min(up, left);
}
```

There will be some Overlapping sub-problems
So we need to apply **Memoization**

Declare a dp of [n-1][m-1] with initialised by -1



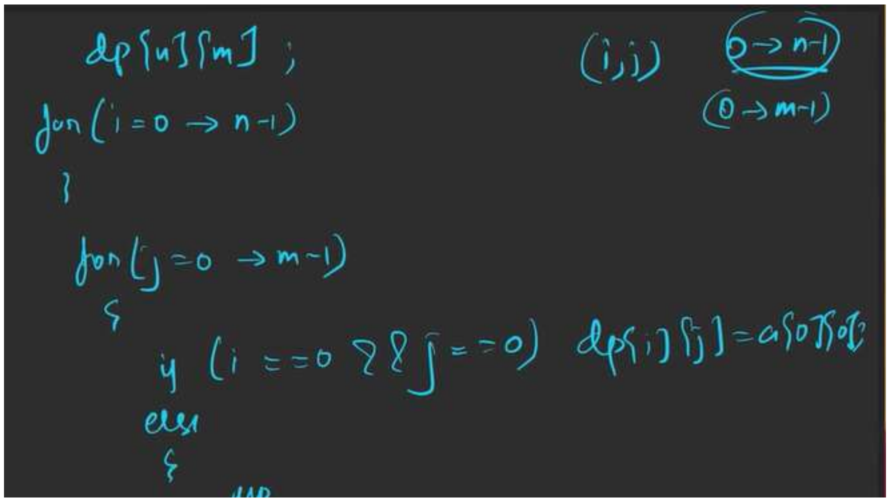
```
f(i,j)
{
  if(i==0 && j==0) return a[0][0];
  if(i<0 || j<0) return INT_MAX;
  if(dp[i][j] != -1) return dp[i][j];
  up = a[i][j] + f(i-1,j)
  left = a[i][j] + f(i,j-1)
  return dp[i][j] = min(up, left);
}
```



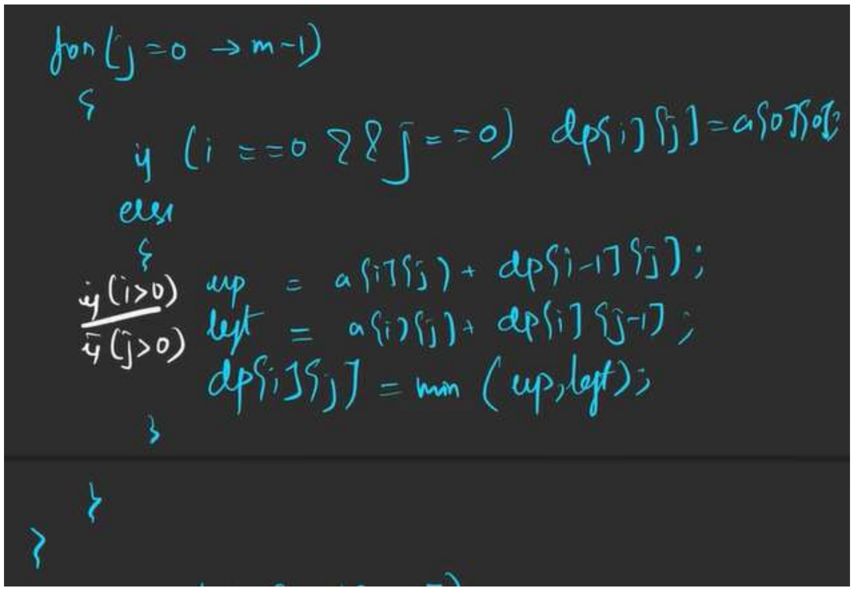
```
TC -> O(NxM)
SC -> O(NxM) +
        O(path length)
         (m-1)+(n-1)
```

Tabulation

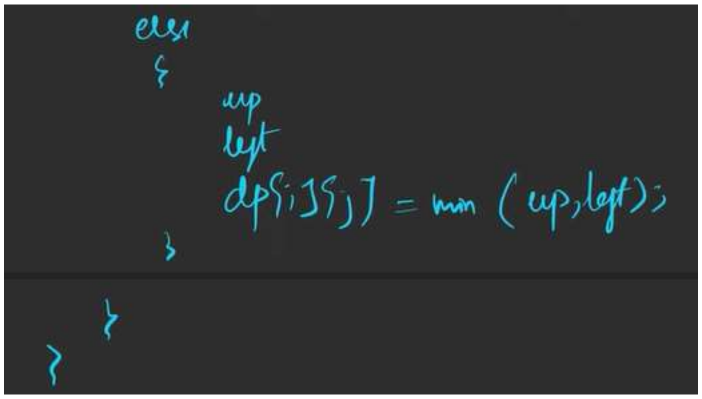
We have 2 changing variables i and j
I goes from 0 to n-1
J goes from 0 to m-1
We take dp[n][m]



```
dp[n][m];                              (i,j)   (0 -> n-1)
for (i=0 -> n-1)                               (0 -> m-1)
 {
  for (j=0 -> m-1)
   {
     if (i==0 && j==0) dp[i][j] = a[0][0];
     else
     {
```



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



```
   else
   {
      up
      left
      dp[i][j] = min (up,left);
   }
 }
}
```

At the end we return

print (dp[n-1][m-1]) ;

Recursion Code

```
int solve(vector<vector<int>> &grid, int i, int j)
{
    if(i==0 && j==0) return grid[i][j];
    if(i<0 || j<0) return 1e9; // we do not need to take this value
so we return INT_MAX as min(left,up) will ignore INT_MAX automatical
ly
    int up = grid[i][j] + solve(grid,i-1,j);
    int left = grid[i][j] + solve(grid,i,j-1);
    return min(left,up);
}

int minSumPath(vector<vector<int>> &grid) {
    int n = grid.size();
    int m = grid[0].size();
    return solve(grid,n-1,m-1);
}
```

Memo Code

```
int solve(vector<vector<int>> &grid, int i, int j,vector<vector<int>
> &dp)
{
    if(i==0 && j==0) return grid[i][j];
    if(i<0 || j<0) return 1e9; // we do not need to take this value
so we return INT_MAX as min(left,up) will ignore INT_MAX automatical
ly
    if(dp[i][j] != -1) return dp[i][j];
    int up = grid[i][j] + solve(grid,i-1,j,dp);
    int left = grid[i][j] + solve(grid,i,j-1,dp);
    return dp[i][j] = min(left,up);
}

int minSumPath(vector<vector<int>> &grid) {
    int n = grid.size();
    int m = grid[0].size();
    vector<vector<int>> dp(n, vector<int>(m,-1));
    return solve(grid,n-1,m-1,dp);
}
```

Tabulation Code

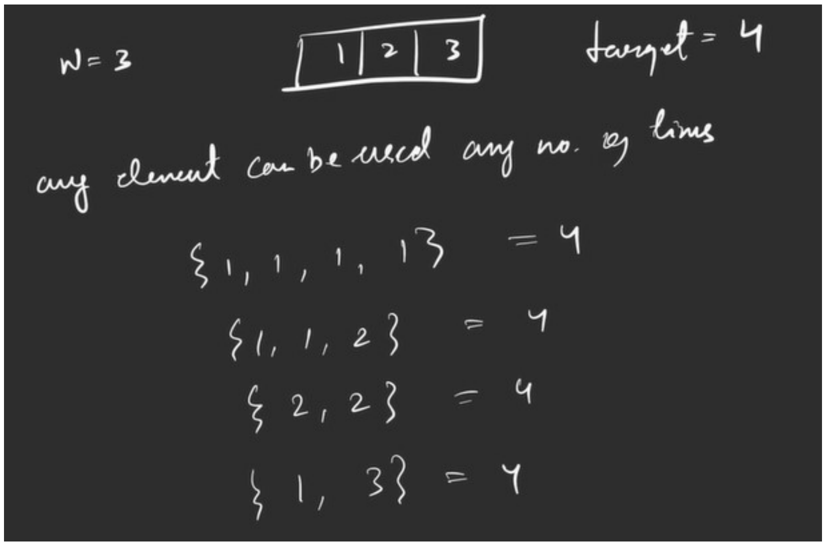
```
int Tabulation(vector<vector<int>> &grid, int n, int m)
{
    vector<vector<int>> dp(n, vector<int>(m,0));

    for(int i = 0;i<n;i++)
    {
        for(int j = 0;j<m;j++)
        {
            if(i==0 && j==0)  dp[i][j] = grid[0][0];
            else{
                int up = grid[i][j];
                if(i>0)
                {
                    up = grid[i][j] + dp[i-1][j];
                }
                else{
                    up += 1e9;
                }

                int left = grid[i][j];

                if(j>0)
                {
                    left += dp[i][j-1];
                }
                else{
                    left += 1e9;
                }

                dp[i][j] = min(up,left);
            }
        }
    }

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

int minSumPath(vector<vector<int>> &grid) {
    int n = grid.size();
    int m = grid[0].size();
    return Tabulation(grid,n,m);
}
```

Coin Change

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.
Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return -1.
You may assume that you have an infinite number of each kind of coin.

Example 1:
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
Example 2:
Input: coins = [2], amount = 3
Output: -1
Example 3:
Input: coins = [1], amount = 0

Output: 0
We can take any number of coins any number of times



We try all ways and return the minimum answer

In total number of ways questions

In base case, we return 1 if destination is reached, we return 0 if destination is not reached

Otherwise we make function call 1
Fucntion call 2

Return call 1 + call 2;

1. Express in term of index,target
2. Explore all possiblities
3. Sum all possiblity and return

F(index, target) means till index = index, how many number of ways are there to make the target = target

Either we can take a element
Or we do not take a element

When we take it, we do target - element and **we do not move to index -1 because we can pick same element any number of time so we remain stand at that index** and if(element < target) then only we can take it.

When we do not take it, we do not change the target and move index - 1

Base case

If we start recursion from last index, we move from n-1 -> 0

Say if we have {3} as element and 4 as target, will we be able to make target from element?

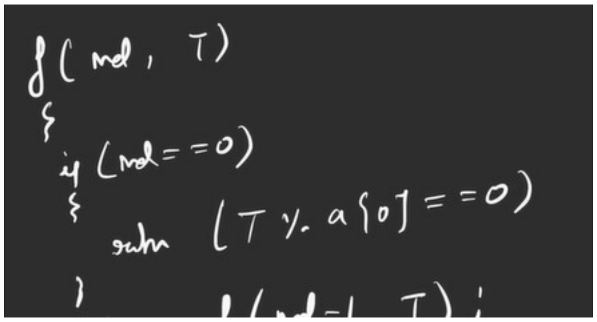
No

If element = {2} and target = 4 then?
Yes {2,2} = 4

If element = {1} and target = 4 then?
Yes {1,1,1,1} = 4

So if (Target % (element) == 0) then we can make target from element? Yes

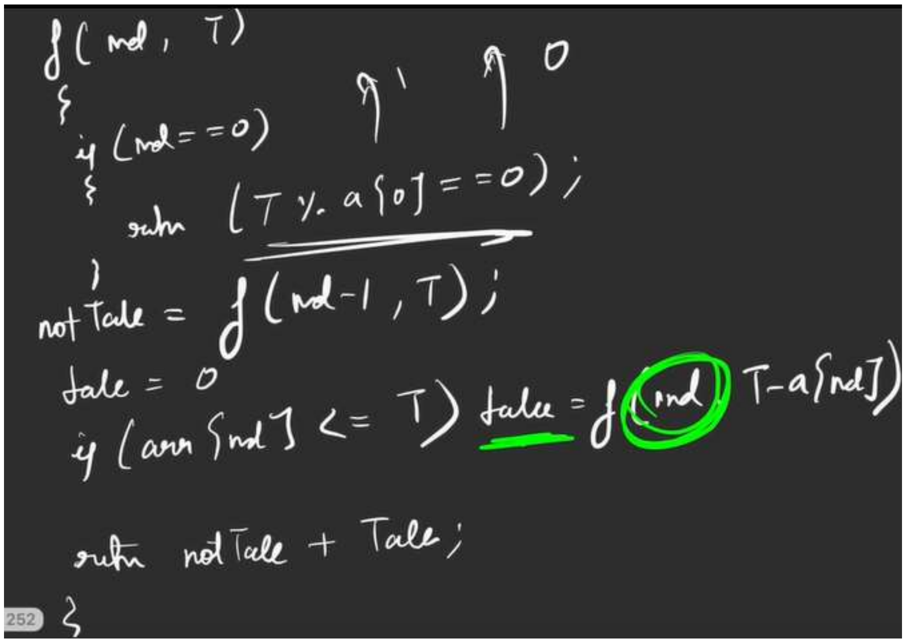
So in base case we will reach index = 0, and if we can use arr[0] for target or not, we see it by doing



```
f( ind, T)
{
  if (ind==0)
  {
     return (T % a[0] == 0)
  }
          f(ind-1, T);
```

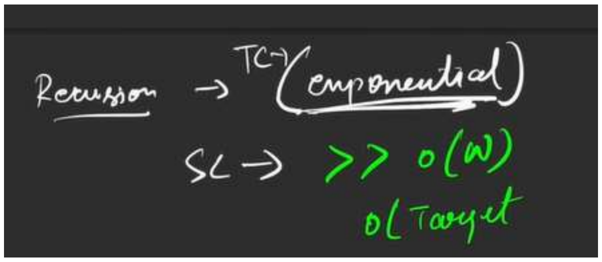
So if it can be included, it return 1, else it return 0

Our **Recursion code** looks like:



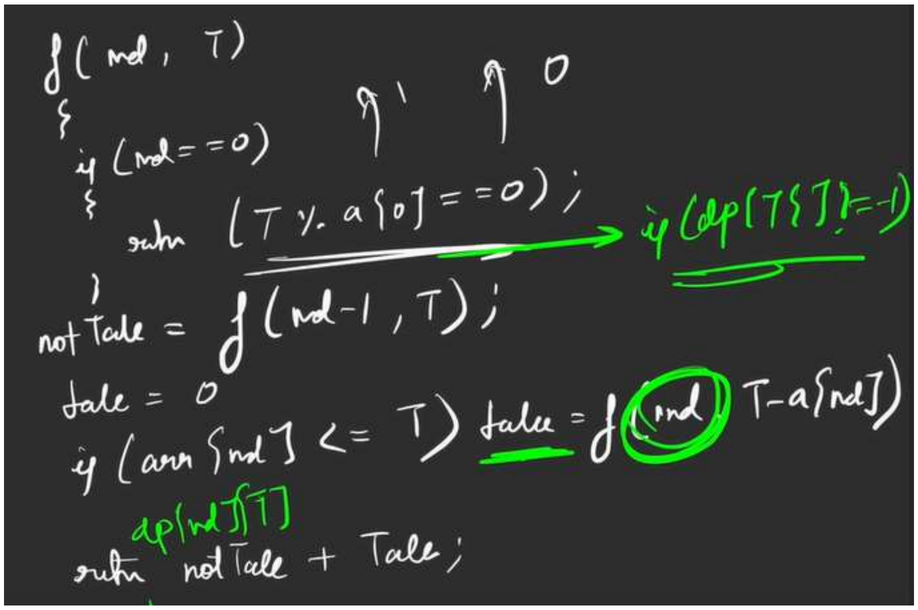
```
f( ind, T)
{
  if (ind==0)
  {
     return (T % a[0] == 0);
  }
notTake = f(ind-1, T);
 take = 0
 if (arr[ind] <= T) take = f(ind, T-a[ind])

 return notTake + Take;
}
```

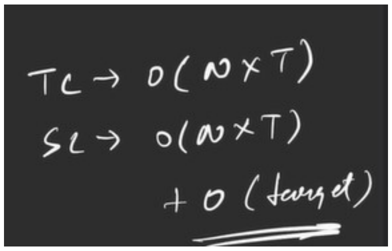
Complexity:


Recursion → TC → (exponential)
SC → >> O(N)
O(Target)

Memo Code

We will have overlapping sub-problems
We will use 2D array for (index, target) as these both are changing.


```
f(ind, T)
{
  if (ind==0)
  {
    return (T % a[0] == 0);      → if (dp[T][T] != -1)
  }
  notTake = f(ind-1, T);
  take = 0
  if (arr[ind] <= T) take = f(ind, T-a[ind])
  return dp[ind][T] = notTake + Take;
```

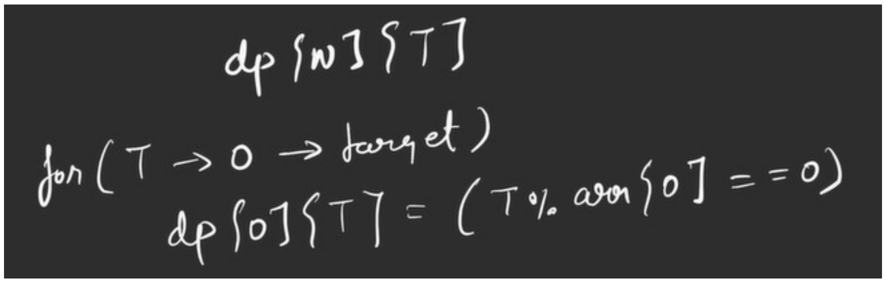

TC → O(N×T)
SC → O(N×T)
+ O(target)

Tabulation code (Bottom up approach)

Steps to write Tabulation

1. Base case
2. Check index, T range
3. Copy the Recurrence

Base case only execute at index = 0
We take dp[N][T]


dp [N] [T]
for (T -> 0 -> target)
dp [0] [T] = (T % arr [0] == 0)

Index goes from n-1 -> 0 in our recurrence
For tabulation it will go from 1 -> n-1 as for index = 0 we have already written the base case
Target move from 0 -> T

Recursion Code

```
long solve(int *deno, int index, int T)
{
    if(index==0)
    {
        return T % deno[0] == 0;
    }
    int notTake = solve(deno,index-1,T);
    int take = 0;
    if(deno[index] <= T)
    {
        take = solve(deno,index,T - deno[index]);
    }
    return take + notTake;
}

long countWaysToMakeChange(int *denominations, int n, int value)
{
    return solve(denominations,n-1,value);
}
```

Memo Code

```
long solve(int *deno, int index, int T, vector<vector<int>> &dp)
{
    if(index==0)
    {
        return T % deno[0] == 0;
    }
    if(dp[index][T] != -1)
    {
        return dp[index][T];
    }
    long notTake = solve(deno,index-1,T,dp);
    long take = 0;
    if(deno[index] <= T)
    {
        take = solve(deno,index,T - deno[index],dp);
    }
    return dp[index][T] = take + notTake;
}

long countWaysToMakeChange(int *denominations, int n, int value)
{
    vector<vector<long>> dp(n, vector<long> (value+1,-1));
    return solve(denominations,n-1,value,dp);
}
```

Tabulation Code

```
long Tabulation(int *deno, int value, int n)
{
    vector<vector<long>> dp(n, vector<long> (value+1,0));
    for(int T = 0;T<=value; T++)
    {
        dp[0][T] = (T % deno[0] == 0);
    }
    for(int index = 1; index < n;index++)
    {
        for(int T = 0; T<=value; T++)
        {
            long notTake = dp[index-1][T];
            long take = 0;
            if(deno[index] <= T)
            {
                take = dp[index][T - deno[index]];
            }
            dp[index][T] = take + notTake;
        }
    }
    return dp[n-1][value];
}

long countWaysToMakeChange(int *denominations, int n, int value)
{
    return Tabulation(denominations,value,n);
}
```

Subset Sum Equal To K

You are given an array/list ‘ARR’ of ‘N’ positive integers and an integer ‘K’. Your task is to check if there exists a subset in ‘ARR’ with a sum equal to ‘K’.

Note: Return true if there exists a subset with sum equal to ‘K’. Otherwise, return false.

```
For Example :
If 'ARR' is {1,2,3,4} and 'K' = 4, then there exists 2
subsets with sum = 4. These are {1,3} and {4}. Hence,
return true.
```

DP on Subsequences/ Subsets

Any contiguous / non-contiguous part of array is called as subsequence/ subset

Subsets need not to be in a sequence

Brute force

Generate all subsequences & check if any of them gives sum of K
But we just need any one subset and return true

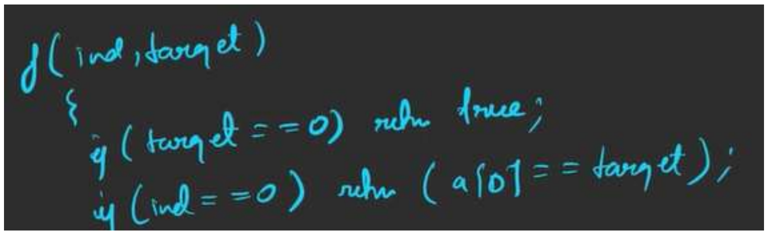
1. Express everything in term of index, target
2. Explore possiblities of that index, possiblities can be A[ind] is part of subsequence or its not part of subsequence
3. Return true / false

We start from index = n-1

F(n -1, target) means In the array till index = n-1, Is there exists a target = target?

What are the base cases?

We might have achieved the target, our target has become zero.
We might have reached index = 0 and we have a target = T left to achieve. Now Can we achieve target T using element at index = 0.
If Arr[0] = target then only we can achieve that T target using index = 0



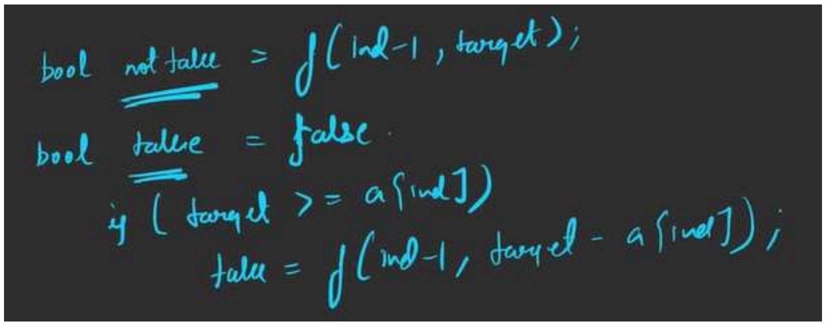
```
f(ind, target)
{
    if (target == 0) return true;
    if (ind == 0) return (a[0] == target);
```

Explore all possiblities now

We either take element or not take it

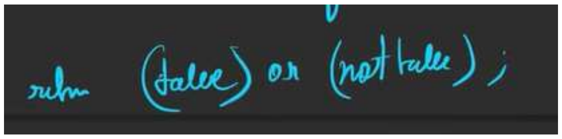
Bool Not-take = we move index - 1 and target remains same
bool take = false;

We can only take an element if(element <= target)
Take = we move index - 1 and target = target - arr[index]



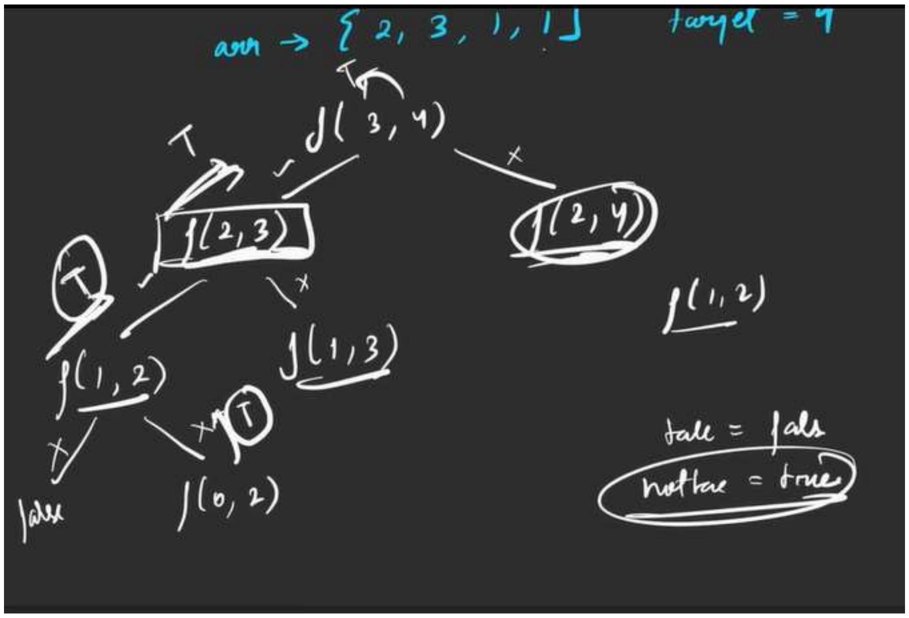
```
bool not take = f(ind-1, target);
bool take = false.
    if ( target >= a[ind])
        take = f(ind-1, target - a[ind]);
```

Now we are looking for possiblities so it take gives us answer, we are OK
If not-take gives us the answer, we are still OK
So return take || not-take



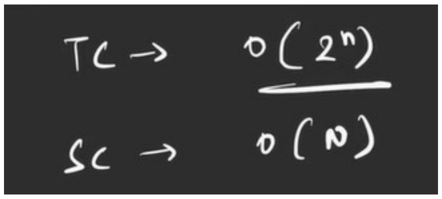
```
return (take) or (not take);
```

Recursion Tree



We see we have some over-lapping subproblems

Recursion TC:

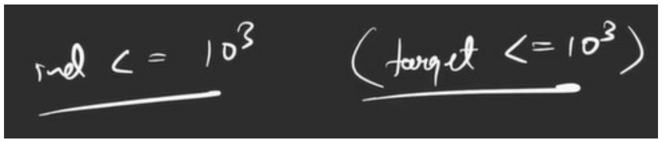


$TC \rightarrow O(2^n)$

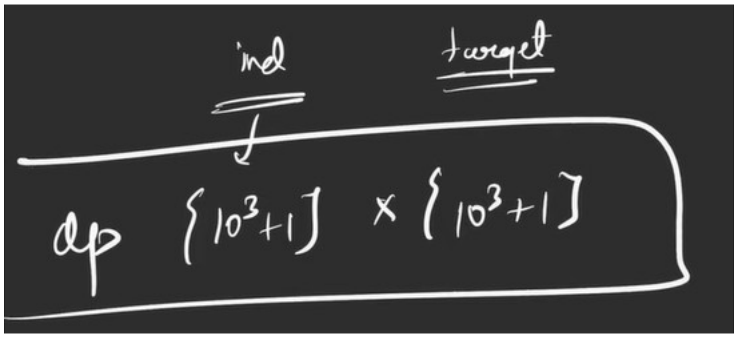
$SC \rightarrow O(N)$

Memo Approach

Changing terms are index and target
Let say our constraints say our index and target are lesser than 10^3

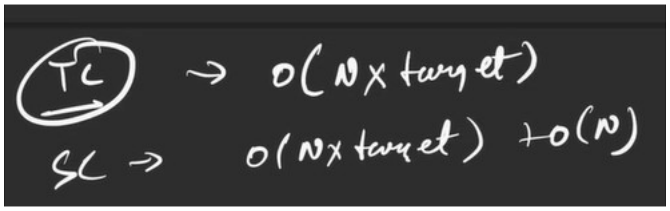

ind <= 10^3 (target <= 10^3)

Then we need 2D Dp of type int of size 10^3 + 1


ind target

dp $[10^3+1] \times [10^3+1]$

Complexity

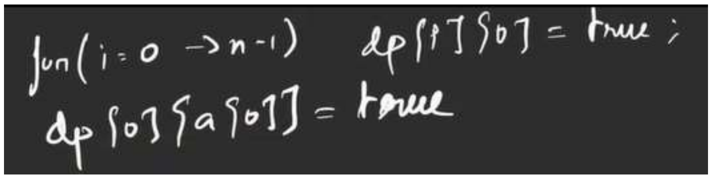

TC → O(N x target)

SC → O(N x target) + O(N)

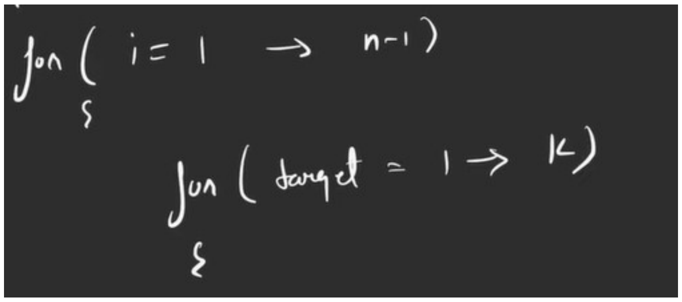
We see we use O(n) as Auxiliary Space, To reduce this space we use Tabulation

Tabulation (Bottom up) bottom up does not means write from last stage to first stage, it means write ulta of whatever was the base case in recurrence

1. Declare DP array of same size
2. See base cases in recurrence and think of indexes it can work upon

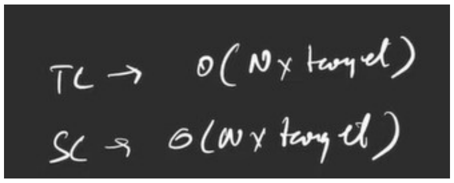

```
for(i=0 -> n-1)  dp[i][0] = true;
dp[0][a[0]] = true
```

3. Form the nexted loops (the number of states = number of nested loops), we have 2 states index and target
   In recurrence we went from index = n-1 to 0, target = target -> 0
   In tabulation we go ulta so
   Index is 1 to n-1 as we have already covered index = 0 in base case
   Target is 1 to value, 0 is already covered in base case


for (i = 1 → n-1)
{
for (target = 1 → k)
{

4. Copy paste the recurrence relation and store it in DP, at the end return dp[n-1][k]

Complexity


TC → O(N x target)
SC → O(N x target)

Recursion Code

```
bool solve(int index, int target, vector<int> arr)
{
    if(target==0) return true;
    if(index==0) return arr[0]==target;
    bool notTake = solve(index-1,target,arr);
    bool take = false;
    if(target >= arr[index])
    {
        take = solve(index-1,target - arr[index] ,arr);
    }
    return take | notTake;
}

bool subsetSumToK(int n, int k, vector<int> &arr) {
    return solve(n-1,k,arr);
}
```

Memo Code

```
bool solve(int index, int target, vector<int> arr,vector<vector<int>
> &dp)
{
    if(target==0) return true;
    if(index==0) return arr[0]==target;
    if(dp[index][target] != -1)
    {
        return dp[index][target];
    }
    bool notTake = solve(index-1,target,arr,dp);
    bool take = false;
    if(target >= arr[index])
    {
        take = solve(index-1,target - arr[index] ,arr,dp);
    }
    return dp[index][target] = take | notTake;
}

bool subsetSumToK(int n, int k, vector<int> &arr) {
    vector<vector<int>> dp(n, vector<int> (k+1,-1));
    return solve(n-1,k,arr,dp);
}
```

Tabulation Code

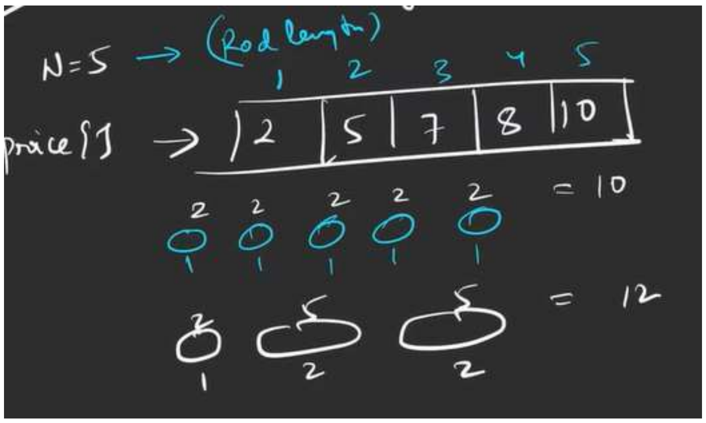
```
bool Tabulation(int n, int k, vector<int> arr)
{
    vector<vector<bool>> dp(n, vector<bool> (k+1,0));
    for(int index = 0; index<n; index++)
    {
        dp[index][0] = true;
    }
    dp[0][arr[0]] = true;
    for(int index = 1;index < n;index++)
    {
        for(int target = 1;target <= k; target++)
        {
            bool notTake = dp[index-1][target];
            bool take = false;
            if(target >= arr[index])
            {
                take = dp[index-1][target - arr[index]];
            }
            dp[index][target] = take | notTake;
        }
    }
    // we are going from 0 to n-1 here in index and 0 to k in target
so dp[n-1][k] has our answer
    return dp[n-1][k];
}

bool subsetSumToK(int n, int k, vector<int> &arr) {
    return Tabulation(n,k,arr);
}
```

Rod cutting problem

Given a rod of length ‘N’ units. The rod can be cut
into different sizes and each size has a cost
associated with it. Determine the maximum cost obtained

```
by cutting the rod and selling its pieces.

Note:

1. The sizes will range from 1 to ‘N’ and will be
integers.
2. The sum of the pieces cut should be equal to ‘N’.
3. Consider 1-based indexing.
```

We are given an Array of N size where value of each index shows price of cutting rod into ith piece

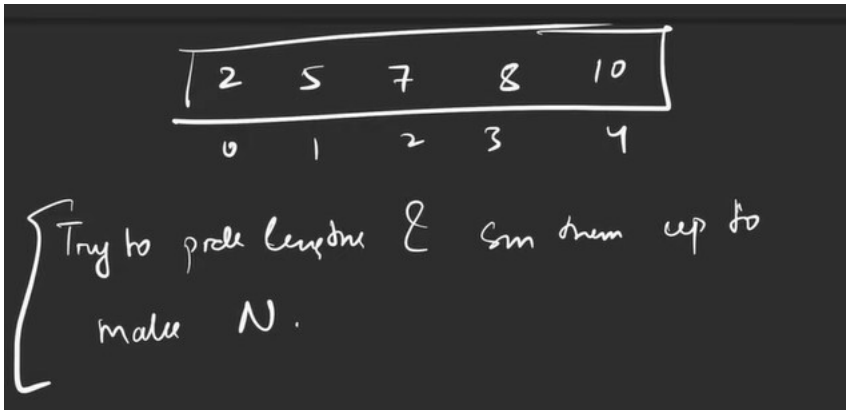


We need to return maximum such cost.

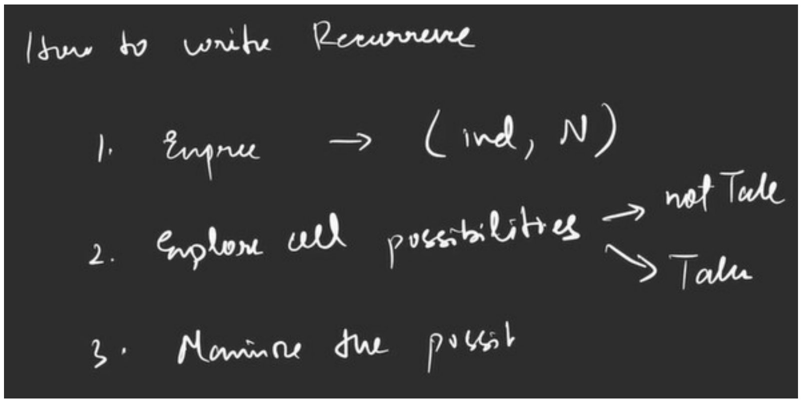
Instead of returning max cost in cutting rod into N pieces we can say Return maximum price for rod length collected to make N

If array is of 0-based indexing, then arr[0] means price of cutting rod into 1 piece
Arr[1] means cost of cutting rod into 2 piece and so on

We try to pick lengths in all possible ways and sum them up to make N



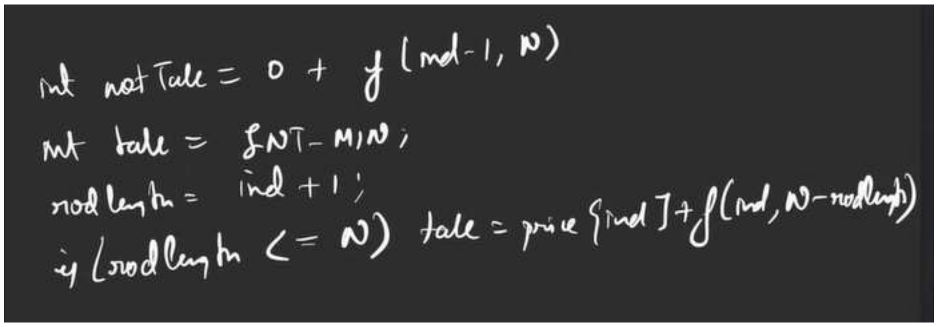
To try all possible ways, we make an recurrence relation


How to write Recurrence

1. Express → (ind, N)
2. Explore all possibilities → not Take
   → Take
3. Maximise the possib

We start from f(index, N) which means till index = index, what is maximum price we can make

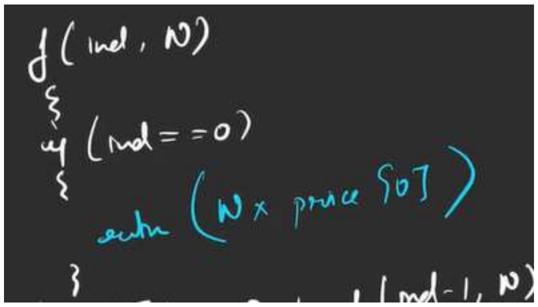
We can pick a index as many times as we want
We make a case of Take / not-Take
We initialise take with INT_MIN as we need maximum price

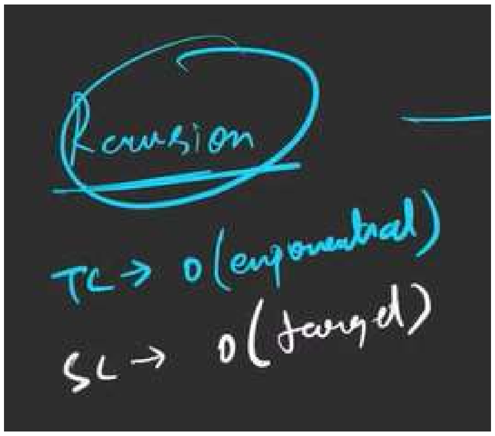

```
int notTake = 0 + f(ind-1, N)
int take = INT-MIN;
rodLength = ind + 1;
if (rodLength <= N) take = price[ind] + f(ind, N-rodLength)
```

We move from n-1 to 0

What will be base case?

If we reach index = 0, We can pick it with price = N * price[0] because say, we have only N part left and we have only index = 0 part which we can use so we need index = 0, N times to fulfill N part so price becomes N * arr[0]


```
f(ind, N)
{
if (ind==0)
{
    return (N × price[0])
  }
            (ind-1, N)
```

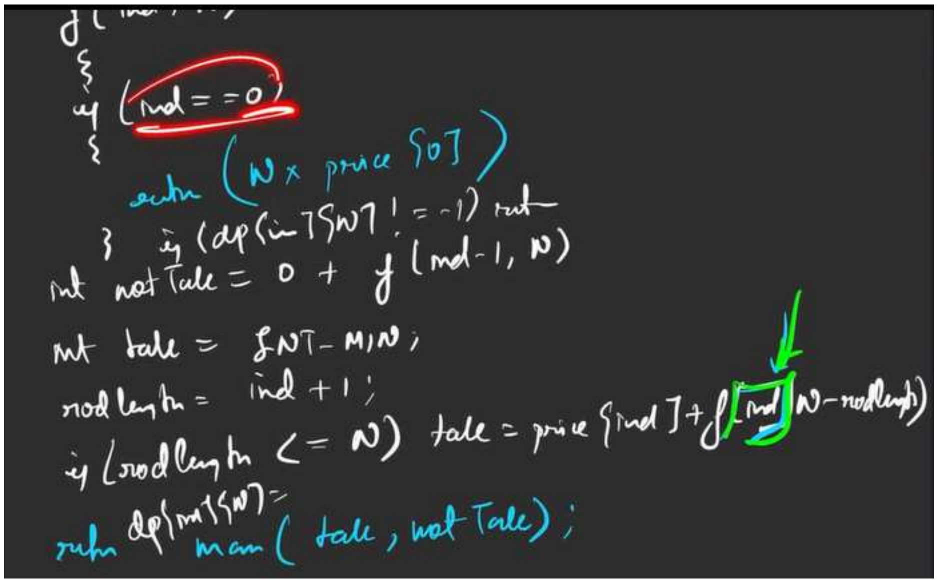


Recursion

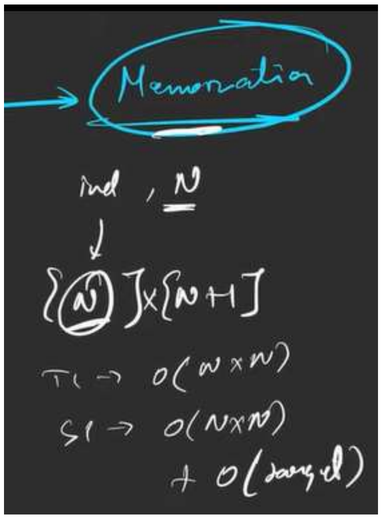
TC → O(exponential)

SC → O(target)

Memo Code



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



Memorization

ind , N

[N] x [N+1]

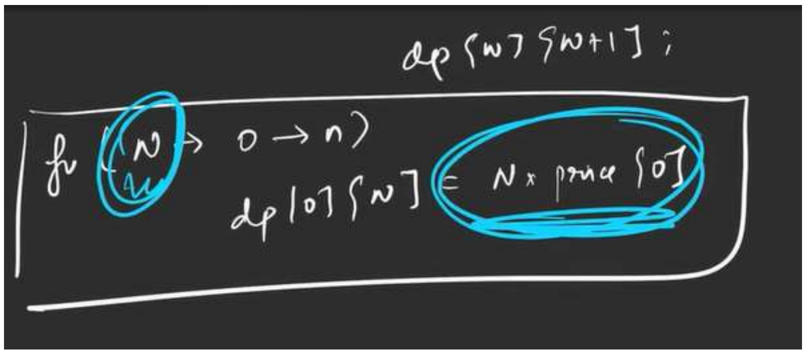
TC → O(N x N)

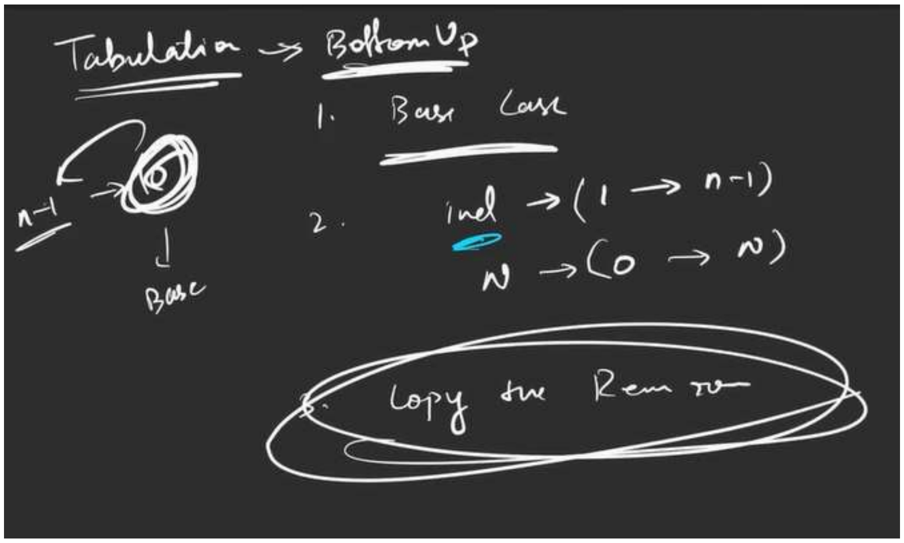
SC → O(N x N)
+ O(target)

Tabulation (Bottom Up)

Our N can go from 0 to n
Our index can go from 0 to n

Base case:





Recursion Code

```
int solve(vector<int> &price, int index, int n)
{
    if(index == 0)
    {
        return n * price[0];
    }
    int notTake = 0 + solve(price,index-1,n);
    int take = INT_MIN;
    int rodLength = index + 1;
    if(rodLength<=n)
    {
        take = price[index] + solve(price, index, n - rodLength);
    }
    return max(take, notTake);
}

int cutRod(vector<int> &price, int n)
{
    return solve(price,n-1,n);
}
```

Memo Code

```
int solve(vector<int> &price, int index, int n, vector<vector<int>>
&dp)
{
    if(index == 0)
    {
        return n * price[0];
    }
    if(dp[index][n] != -1)
    {
        return dp[index][n];
    }
    int notTake = 0 + solve(price,index-1,n,dp);
    int take = INT_MIN;
    int rodLength = index + 1;
    if(rodLength<=n)
    {
        take = price[index] + solve(price, index, n - rodLength,dp);
    }
    return dp[index][n] = max(take, notTake);
}

int cutRod(vector<int> &price, int n)
{
    vector<vector<int>> dp(n, vector<int> (n+1,-1));
    return solve(price,n-1,n,dp);
}
```

Tabulation Code

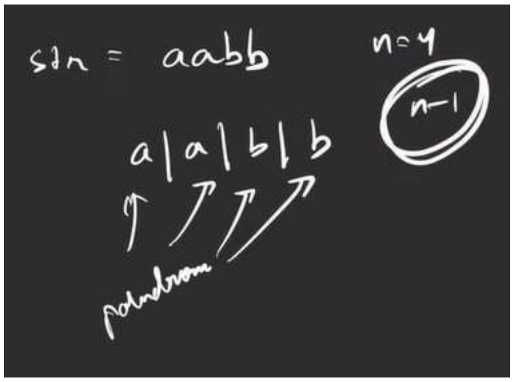
```
int Tabulation(vector<int> &price, int n)
{
    vector<vector<int>> dp(n, vector<int> (n+1,0));
    for(int N = 0;N<=n; N++)
    {
        dp[0][N] = N*price[0];
    }
    for(int index = 1;index<n; index++)
    {
        for(int N = 0; N<=n; N++)
        {
            int notTake = 0 + dp[index-1][N];
            int take = INT_MIN;
            int rodLength = index + 1;
            if(rodLength<=N)
            {
                take = price[index] + dp[index][N - rodLength];
            }
            dp[index][N] = max(take, notTake);
        }
    }
    return dp[n-1][n];
}

int cutRod(vector<int> &price, int n)
```

```
{
    return Tabulation(price,n);
}
```

Palindrome Partitioning 2

```
You are given a string 'str' of length 'n'
Find the minimum number of partitions in the string so
that no partition is empty and every partitioned
substring is a palindrome.
Example:
Input: 'str' = "aaccb"
Output: 2
Explanation: We can make a valid partition like aa | cc
| b.
```

We will have n-1 partitions



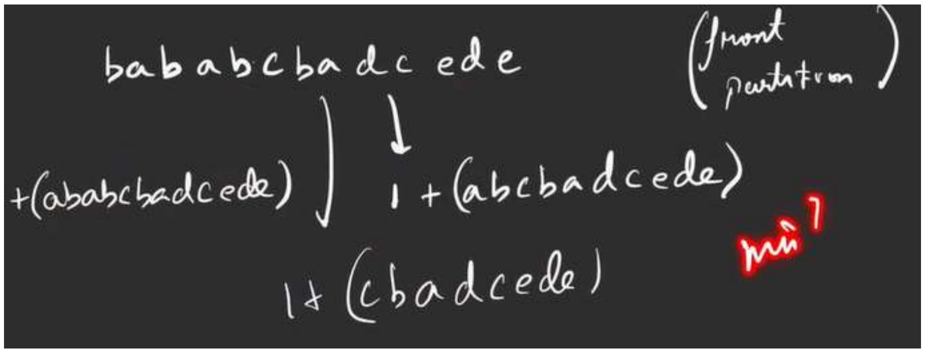
We need to find minimum number of paritions which can give us string where every partition is palindrome and not empty.

Front Partition Approach

We start from front index and check if we can make a partition after first index, if yes
We do 1 + (rest of string)
Now we work upon rest of the string

When we do partition everytime, we check if its previous whole string is a palindrome or not

For the given string, we can do partition at 3 points and each partition gives us some result, we need to take the minimum result



Recurrence Relation

We are trying all partitions
We start initial index from i = 0 as we are doing front partition.
Express all possibllities
Take min of all possiblities
Write base case

Base case

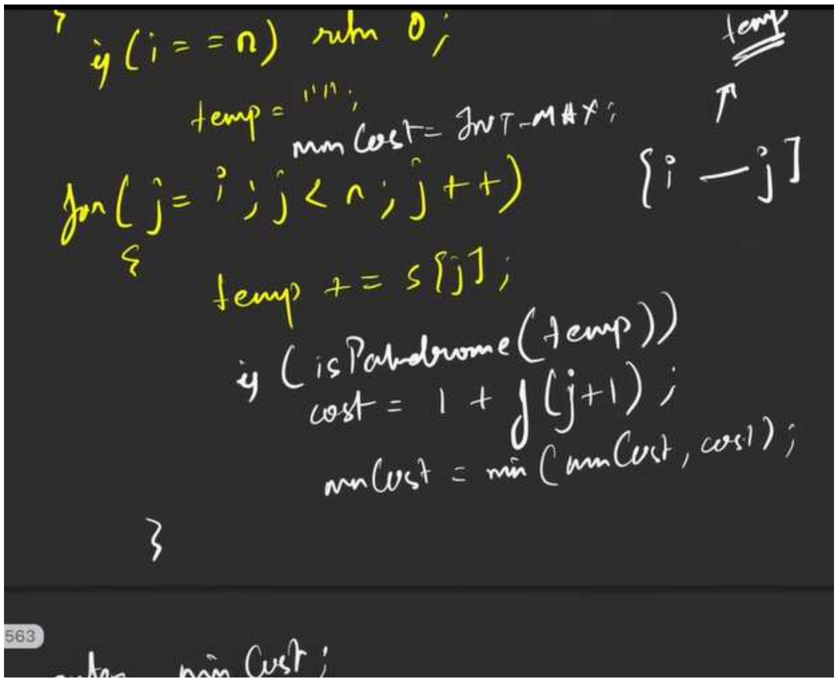
When we reach end of string, we return 0

We take a variable Min_cost = INT_MAX to store cost

We take another index j, to store whole string between i and j we make a variable temp, and check if its palindrome, if its palindrome then only we can make a partition
So we add + 1 to cost and call for j+1 i.e next index

Min_cost = min(cost, Min_cost)

Return Min_cost



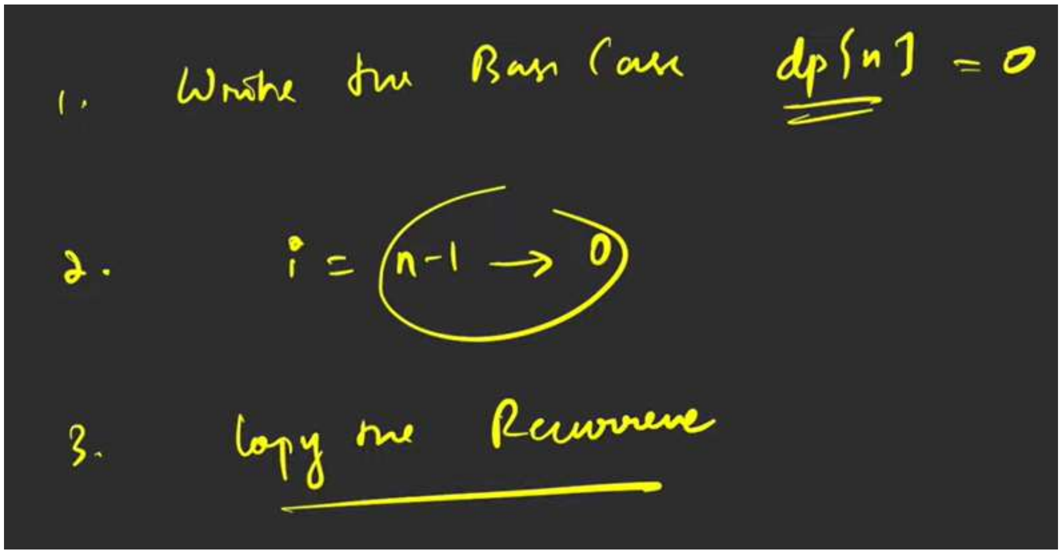
```
if (i == n) return 0;
    temp = "";
    minCost = INT_MAX;
for (j = i; j < n; j++)
{
    temp += s[j];
    if (isPalindrome(temp))
        cost = 1 + f(j+1);
        minCost = min(minCost, cost);
}
return minCost;
```

Complexity of Recurrence is Exponential so we apply **Memoisation**

Here only one parameter is changing i.e index i, so we take dp[i]

Tabulation

When index == n, return 0 means dp[n] = 0
In recurrence we move i from 0 to n-1 so in Tabulation we move i from n-1 to 0


1. Write the Base Case dp[n] = 0
2. i = n-1 → 0
3. Copy the Recurrence

Recursion Code

```
bool isPalindrome(int i, int j, string str)
{
    while(i<j)
    {
        if(str[i] != str[j])
        {
            return false;
        }
        i++;
        j--;
    }
    return true;
}

int solve(string str, int index,int n)
{
    if(index == n)
    {
        return 0;
    }
    int Min_cost = INT_MAX;
    for(int j = index; j<n ; j++)
    {
        if(isPalindrome(index,j,str))
        {
            int cost = 1 + solve(str,j+1,n);
            Min_cost = min(Min_cost, cost);
        }
    }
    return Min_cost;
}

int palindromePartitioning(string str)
{
    int n = str.size();
    //we need to do -1 in the answer because our code run till i ==
n means it puts a partition after last index also
    // which is not needed so we do -1 below
    return solve(str,0,n) -1;
}
```

Memo Code

```
bool isPalindrome(int i, int j, string str)
{
    while(i<j)
    {
        if(str[i] != str[j])
        {
            return false;
        }
        i++;
        j--;
    }
    return true;
}

int solve(string str, int index,int n,vector<int> &dp)
{
    if(index == n)
    {
        return 0;
    }
    if(dp[index] != -1)
    {
        return dp[index];
    }
    int Min_cost = INT_MAX;
    for(int j = index; j<n ; j++)
    {
        if(isPalindrome(index,j,str))
        {
            int cost = 1 + solve(str,j+1,n,dp);
            Min_cost = min(Min_cost, cost);
        }
    }
    return dp[index] = Min_cost;
}

int palindromePartitioning(string str)
{
    int n = str.size();
    //we need to do -1 in the answer because our code run till i ==
n means it puts a partition after last index also
    // which is not needed so we do -1 below
    vector<int> dp(n,-1);
    return solve(str,0,n,dp) -1;
}
```

Tabulation Code

```
bool isPalindrome(int i, int j, string str)
{
    while(i<j)
    {
        if(str[i] != str[j])
        {
            return false;
        }
        i++;
        j--;
    }
    return true;
}

int palindromePartitioning(string str)
{
    // Tabulation code
    int n = str.size();
    //we need to do -1 in the answer because our code run till i ==
n means it puts a partition after last index also
    // which is not needed so we do -1 below
    vector<int> dp(n+1,0);
    dp[n] = 0;
    for(int i = n-1; i>=0; i--)
    {
        int Min_cost = INT_MAX;
        for(int j = i; j<n ; j++)
        {
            if(isPalindrome(i,j,str))
            {
                int cost = 1 + dp[j+1];
                Min_cost = min(Min_cost, cost);
            }
        }
        dp[i] = Min_cost;
    }
    return dp[0]-1;
}
```

Egg Dropping Puzzle

There exists a floor **f** where **0** <= **f** <= **K** such that any egg dropped from a floor higher than **f** will break, and any egg dropped **from or below** floor **f** will **not break**.

There are few rules given below.

- An egg that survives a fall can be used again.
- A broken egg must be discarded.
- The effect of a fall is the same for all eggs.
- If the egg doesn't break at a certain floor, it will not break at any floor below.
- If the eggs breaks at a certain floor, it will break at any floor above.

Return the minimum number of moves that you need to determine with certainty what the value of f is.

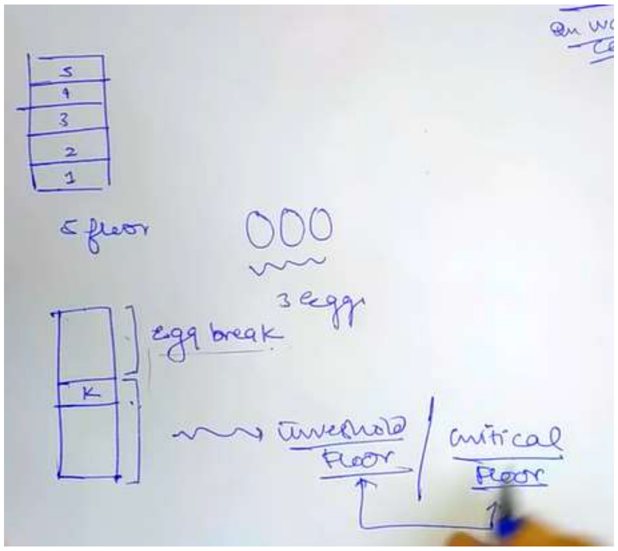
Example 1:
Input:
N = 1**, K** = 2
Output: 2
Explanation:
1. Drop the egg from floor 1. If it
   breaks, we know that f = 0.
2. Otherwise, drop the egg from floor 2.
   If it breaks, we know that f = 1.
3. If it does not break, then we know f = 2.
4. Hence, we need at minimum 2 moves to
   determine with certainty what the value of f is.

Example 2:
Input:
N = 2, K = 10
Output: 4

Number of eggs and number of floors are given

We need to find critical / threshold floor
What is critical / threshold floor?

A floor "k" including it and above which if we throw egg from any floor, the egg will break and below which if we use any floor to throw egg, the egg will not break.



We need the minimum number of attemps in finding the critical floor.

We will go to every floor and and throw egg
If egg break, go to lower floors so that chances are egg will not break at some floor

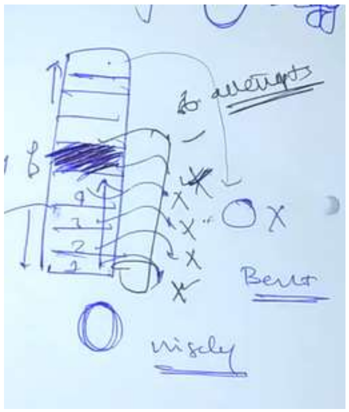
We need to use number of eggs given very wisely. Our technique should be best.

Let say we have number of eggs = 3
We have number of floors = 50

We throw egg from floor = 50, egg break, now we have 2 eggs, we go to lower floor
We throw egg from floor = 49, egg break, now we have 1 eggs, go to lower floor
We throw egg from floor = 48, egg break, now we have 0 eggs

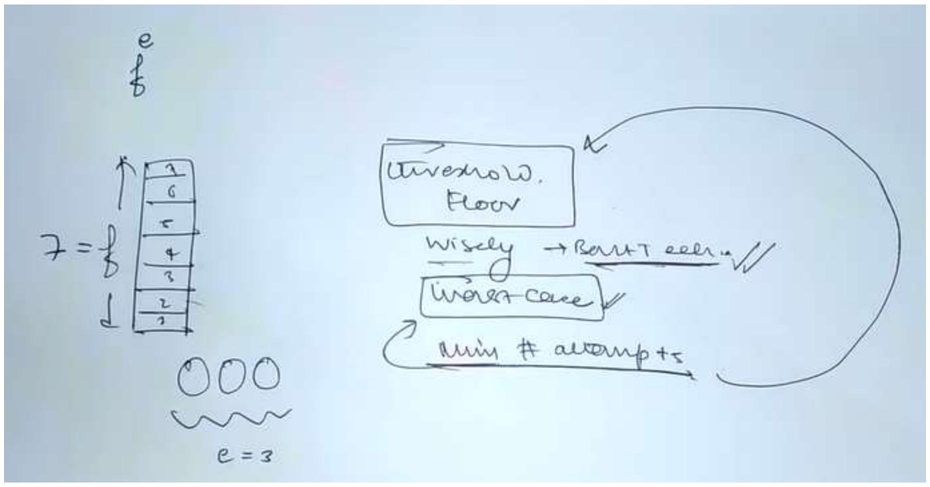
Let say our critical floor was = 12, but now we don't have eggs to check so we need to use our eggs wisely

If we would have started from floor = 1, egg would not have broken and we could have been able to re-use it.
So best technique is to start dropping from lowest floor

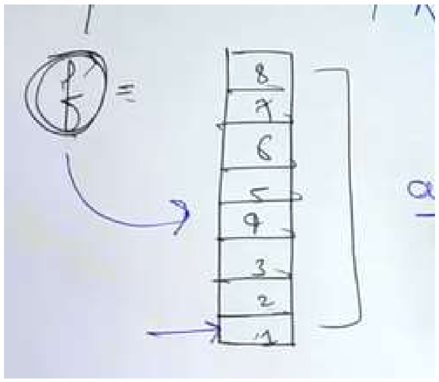


In worst case, we will reach top floor and it’s the critical floor then it take us f attempts

In worst case, we need to find minimum number of attempts to find threshold floor, using best technique.



f represents number of floors in a building, which starts from 1 to f
This can be analogous to an array only,



MCM Variation

We take an pointer i at 1st floor and j pointer at last floor
Now how to take k?
We will check for each value so k can go from 1st floor to last floor

Base case
Think of the smallest valid input
Egg = 0, this is invalid condition according to problem statement
Egg = 1, then we need to go to f floors, return f
Floor = 0, return 0 as there is no floor
Floor = 1, return 1 as there is only 1 floor

Possiblities

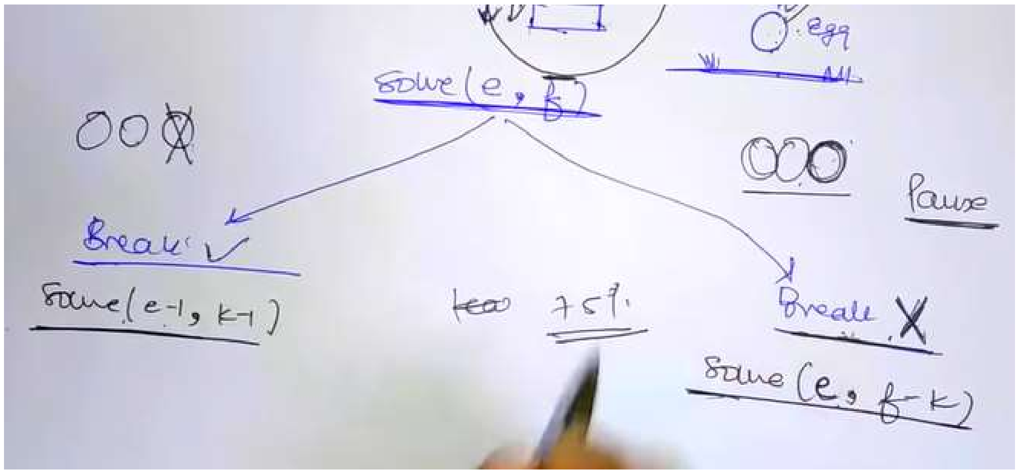
If we drop an egg there are **2 possiblities**

We begin from function(egg, floor)

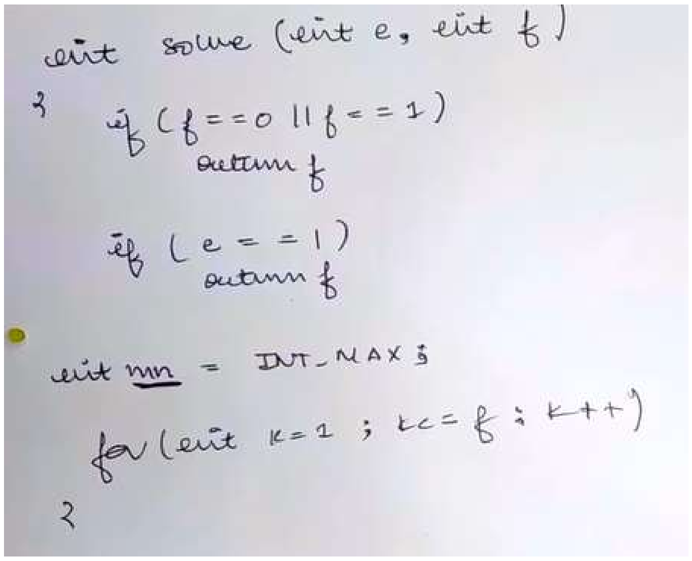
1. **It will break**, say its kth floor so egg will break for all floors > k
   And our threshold floor is the one jiske upar break krta h and jiske neeche nhi break krta so threshold floor in below k, so we go k-1 and egg broke so egg =

egg-1
Function (egg-1, k-1);

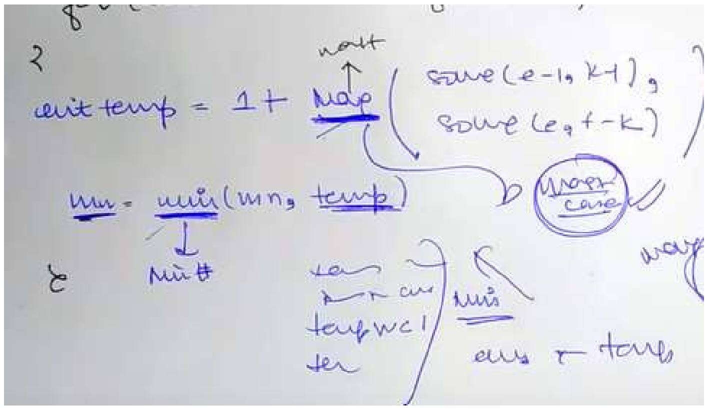
2. **It will not break**, means we need to check above k as for all below k, egg will not break. We need to check (f - k) floors now, egg will not break means It can be re-used so egg = egg only
   Function(egg, F - K);



Code:



```
int solve (int e, int f)
{
    if (f==0 || f==1)
        return f

    if (e == 1)
        return f

  int mn = INT_MAX;

   for (int k=1 ; k<=f ; k++)
   {
```



```
{
int temp = 1 + max( solve(e-1, k-1),
                    solve(e, f-k) )

mn = min(mn, temp)

}
```
worst
worst case
Min#

```
int solve(int egg, int floor)
{
    // if there is only 0 or 1 floor left
    if(floor==0 || floor==1) return floor;

    // if there is only 1 egg, then the floor we are at is the criti
cal/threshold floor
    if(egg==1) return floor;

    int mini = INT_MAX;

    // we run loop for each floor and apply break/ not break conditi
on
    for(int k = 1; k<= floor; k++)
    {
        // we need to solve for the worst case so we use max() to ge
t the worst case answer from break and not break conidition for each
floor
        int temp = 1 + max(solve(egg-1,floor-1), solve(egg, floor -
k));

        // get the minimum number of attemps from that worst case an
swer to fing threshold floor
        mini = min(mini,temp);
    }
    return mini;
}

int eggDrop(int n, int k)
{
    return solve(n,k);
}
```

Memoization

Why Memoization is needed?

If we draw a complete Recursive Tree, we observe there are some **overlapping sub problems.**

We will make a matrix, we see there are 2 changing parameters in our recurrence, egg and floor so we make 2D DP Array initialised with -1.

What will be size and dimension of Matrix?
It depends on number of changing variables.

```
int solve(int egg, int floor,vector<vector<int>> &dp)
{
    // if there is only 0 or 1 floor left
    if(floor==0 || floor==1) return floor;

    // if there is only 1 egg, then the floor we are at is the criti
cal/threshold floor
    if(egg==1) return floor;
    if(egg==0) return 0;

    if(dp[egg][floor] != -1) return dp[egg][floor];

    int mini = INT_MAX;

    // we run loop for each floor and apply break/ not break conditi
on
    for(int k = 1; k<= floor; k++)
    {
        // we need to solve for the worst case so we use max() to ge
t the worst case answer from break and not break coniditon for each
floor
        int temp = 1 + max(solve(egg-1,floor-1,dp), solve(egg, floor
 - k,dp));

        // get the minimum number of attemps from that worst case an
swer to fing threshold floor
        mini = min(mini,temp);
    }
    return dp[egg][floor] = mini;
}

int eggDrop(int n, int k)
{
    vector<vector<int>> dp(n+1, vector<int>(k+1,-1));
    return solve(n,k,dp);
}
```

Word Break

Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:
Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".
Example 2:
Input: s = "applepenapple", wordDict = ["apple","pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
Note that you are allowed to reuse a dictionary word.
Example 3:
Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
Output: false

```
bool solve(string s, int index, set<string> &st)
{
    if(index == s.size()) return true;
    // make an temp string
    string temp = "";

    for(int j = index; j< s.size(); j++)
    {
        // store character by character inside the string temp and c
heck if it matches the dict
        temp += s[j];
        if(st.find(temp) != st.end())
        {
            // check for next index of j
            if(solve(s,j+1,st))
            {
                return true;
            }
        }
    }
    return false;
}

bool wordBreak(string s, vector<string>& wordDict) {
    // What we do is we will pick one character and try to match it 
in the      dictionary
    // starting from index = 0, we run a loop j = i to  j < s.size()
    // we will store dict in set so that its easy to compare
    // gives TLE
    set<string> st;
    for(auto it: wordDict)
    {
        st.insert(it);
    }

    return solve(s,0,st);
}
```

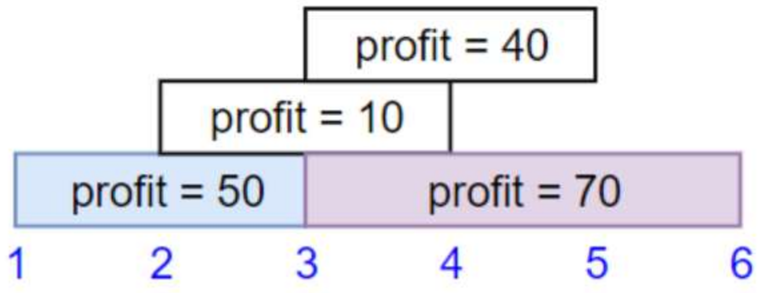
Memoized Code:

```
bool solve(string s, int index, set<string> &st,vector<int> &dp)
{
    if(index == s.size()) return true;

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

    // make an temp string
    string temp = "";

    for(int j = index; j< s.size(); j++)
    {
        // store character by character inside the string temp and c
heck if it matches the dict
        temp += s[j];
        if(st.find(temp) != st.end())
        {
            // check for next index of j
            if(solve(s,j+1,st,dp))
            {
                return dp[index] = true;
            }
        }
    }
    return dp[index] = false;
}

bool wordBreak(string s, vector<string>& wordDict) {
    // What we do is we will pick one character and try to match it 
in the  dictionary
    // starting from index = 0, we run a loop j = i to  j < s.size()
    // we will store dict in set so that its easy to compare
    // gives TLE
    // let us make an DP array, we have only one changing variable i
.e index so we make 1D DP array
         // Instead of calculating for index i again and again we pass 
         DP array and store result in it.
    set<string> st;
    vector<int> dp(s.size(),-1);
    for(auto it: wordDict)
    {
        st.insert(it);
    }

    return solve(s,0,st,dp);
}
```

Maximum Profit in Job Scheduling

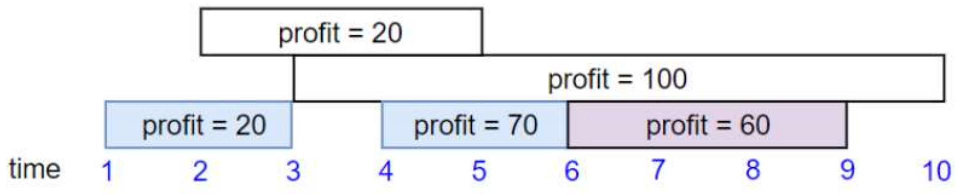
We have n jobs, where every job is scheduled to be done from startTime[i] to endTime[i], obtaining a profit of profit[i].
You're given the startTime, endTime and profit arrays, return the maximum profit you can take such that there are no two jobs in the subset with overlapping time range.
If you choose a job that ends at time X you will be able to start another job that starts at time X.

Example 1:



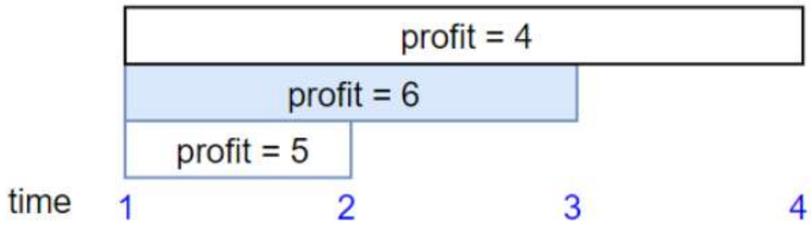
```
Input: startTime = [1,2,3,3], endTime = [3,4,5,6], profit = 
[50,10,40,70]
Output: 120
Explanation: The subset chosen is the first and fourth job. 
Time range [1-3]+[3-6] , we get profit of 120 = 50 + 70.
```

Example 2:



```
Input: startTime = [1,2,3,4,6], endTime = [3,5,10,6,9], profit =
[20,20,100,70,60]
Output: 150
Explanation: The subset chosen is the first, fourth and fifth job.
Profit obtained 150 = 20 + 70 + 60.
```

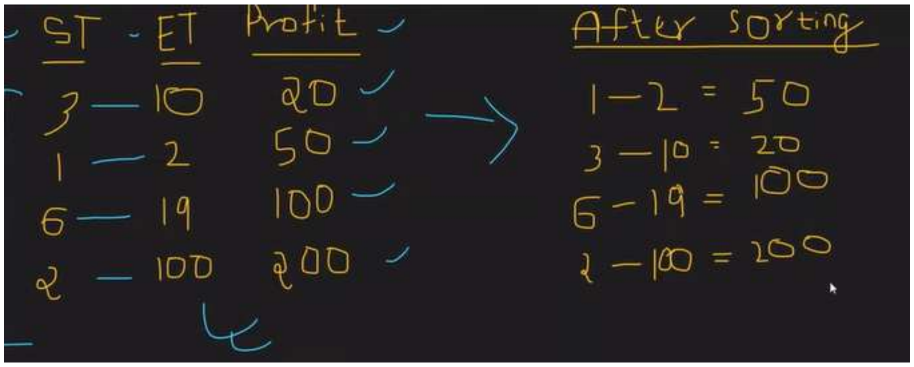
Example 3:



```
Input: startTime = [1,1,1], endTime = [2,3,4], profit = [5,6,4]
Output: 6
```

Starting and ending time of 2 jobs should not overlap and we need maximum profit

We can do sorting for ease in solving



ST	ET	Profit
3	10	20
1	2	50
6	19	100
2	100	200

After sorting

1 — 2 = 50
3 — 10 = 20
6 — 19 = 100
2 — 100 = 200

We cannot try greedy here, we need to take each case and find out profit for it
We will try pick / not pick approach
While picking we check that it does not overlap with our previous job
We store maximum in the answer