Flow:

1.problem statement.
2.similarity
3.code variation
a.initialization
b.code
4.return type.

For 0/1 Knapsack problems use recursion and for lcs use bottom up table approach

A.1.Knapsack problems:

Remember to compare any problem with parent problems like 0/1 knapsack,lcs etc compare it by using following 3 parameters:
1.no. of inputs
2.name
3.return type
At Least if 2 parameters match then go ahead with the parent problem.

1.0/1 Knapsack:

Bruteforce approach:

A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets. Consider the only subsets whose total

weight is smaller than W. From all such subsets, pick the maximum value subset.

Optimal Sub-structure: To consider all subsets of items, there can be two cases for every item.

1. Case 1: The item is included in the optimal subset.
2. Case 2: The item is not included in the optimal set.

Therefore, the maximum value that can be obtained from ‘n’ items is the max of the following two values.

1. Maximum value obtained by n-1 items and W weight (excluding nth item).
2. Value of nth item plus maximum value obtained by n-1 items and W minus the weight of the nth item (including nth item).

If the weight of ‘nth’ item is greater than ‘W’, then the nth item cannot be included and Case 1 is the only possibility.

Complexity ,Time=O(2^n).
Space=O(1).

Dyanmic approach:

1.Memoized approach:

```
int knapSack(int W, int wt[], int val[], int n)
  {
     int **t;
```

```
        t=new int*[n+1];
        for(int i=0;i<=n;i++)t[i]=new int[W+1];
        for(int i=0;i<=n;i++)
          for(int j=0;j<W+1;j++)
             t[i][j]=-1;

        return knapSackRec(W,wt,val,n,t);
    }
    int knapSackRec(int W,int wt[],int val[],int n,int **t)
{
        if(n<0)return 0;
        if(t[n][W]!=-1)return t[n][W];
        else{
            if(wt[n-1]>W)return
t[n][W]=knapSackRec(W,wt,val,n-1,t);
            else
            return
t[n][W]=max(val[n-1]+knapSackRec(W-wt[n-1],wt,val,n-1,t),
                              knapSackRec(W,wt,val,n-1,t));
        }
    }
```

Complexity, Time=O(N*W)
Space=O(N*W).
2.Top down approach:

```
int knapSack(int W, int wt[], int val[], int n)
    {
        int t[n+1][W+1];
        for(int i=0;i<=n;i++)
        for(int j=0;j<=W;j++){
            if(i==0 || j==0)t[i][j]=0;
            else if(wt[i-1]<=j)
                t[i][j]=max(val[i-1]+t[i-1][j-wt[i-1]],t[i-1][j]);
            else t[i][j]=t[i-1][j];
        }
        return t[n][W];
    }
```

2.subset sum:

Method 1:Brute force

We need to generate all subsets and then check one by one with sum.

Complexity,time=O(2^n).
space=O(1).

Method 2:

Recursive approach:

```
bool isSubsetSum(int n, int arr[], int sum)
{
        int **t;
        t=new int*[n+1];
        for(int i=0;i<=n;i++)t[i]=new int[sum+1];
        for(int i=0;i<=n;i++)
```

```
          for(int j=0;j<sum+1;j++)
            t[i][j]=-1;
            return (isissubsetsum(n,arr,sum,t));
        }

    bool isissubsetsum(int n,int arr[],int sum,int **t){
        if(sum==0)return true;
        else if(n<=0)return false;
        else{
            if(t[n][sum]!=-1)return t[n][sum];
            if(arr[n-1]>sum)
            return t[n][sum]=isissubsetsum(n-1,arr,sum,t);
            else
            return t[n][sum]=isissubsetsum(n-1,arr,sum,t)
                             ||isissubsetsum(n-1,arr,sum-arr[n-1],t);
        }
    }
```

Top Bottom approach:

```
bool isSubsetSum(int n, int arr[], int sum)
{
      int t[n+1][sum+1];
      for(int i=0;i<n+1;i++)
        for(int j=0;j<sum+1;j++)
        {
            if(j==0)t[i][j]=true;
            else if(i==0)t[i][j]=false;
            else{
                if(arr[i-1]<=j)
                  t[i][j]=t[i-1][j-arr[i-1]] || t[i-1][j];
            else
            t[i][j]=t[i-1][j];

          }
        }
        return t[n][sum];
   }
```

Complexity,time=O(n*sum).
                space=O(n*sum).

3.Equal Sum:

Method 1:
We need to sum up all the elements and check whether sum is divisible by 2 or not,if yes then we can generate all subsets and check whether we get a sum equal to sum/2 if ye sthen true otherwise false.

Complexity ,time=O(2^n).
                  space=O(1).

Method 2:

Recursive approach:

```
bool equalPartition(int n, int arr[])
   {
       int sum=0;
```

```
    for(int i=0;i<n;i++)sum+=arr[i];
    if(sum%2==0)
    {
       int sum1=sum/2;
    int **t;
    t=new int*[n+1];
    for(int i=0;i<=n;i++)t[i]=new int[sum1+1];
    for(int i=0;i<=n;i++)
      for(int j=0;j<sum1+1;j++)
        t[i][j]=-1;
        return (isissubsetsum(n,arr,sum1,t));
    }
    else return false;

 }
 bool isissubsetsum(int n,int arr[],int sum,int **t){
    if(sum==0)return true;
    else if(n==0)return false;
    else{
       if(t[n][sum]!=-1)return t[n][sum];
       if(arr[n-1]>sum)
       return t[n][sum]=isissubsetsum(n-1,arr,sum,t);
       else
       return t[n][sum]=isissubsetsum(n-1,arr,sum,t)
                    ||isissubsetsum(n-1,arr,sum-arr[n-1],t);
    }
 }
```

Top down approach:

```
bool canPartition(vector<int>& nums) {
        int sum=0;
         int n=nums.size();
        for(int i=0;i<n;i++)sum+=nums[i];
        if(sum%2==0){
            int sum1=sum/2;
            int t[n+1][sum+1];
            for(int i=0;i<=n;i++)
                for(int j=0;j<=sum1;j++){
                    if(j==0)t[i][j]=true;
                    else if(i==0)t[i][j]=false;
                    else{
                        if(nums[i-1]<=j)t[i][j]=t[i-1][j-nums[i-1]] ||
                            t[i-1][j];
                        else
                            t[i][j]=t[i-1][j];

                    }

                }
            return t[n][sum1];
        }
            else return false; }
```

Complexity,time=O(n*sum1);space=O(n*sum1);

4.Count the subset sum:

Method 1:

Bruteforce will be same like previous ones.

Method 2:

Note:if(i==0){if(j==0return……..};

Recursive approach:

```
int perfectSum(int arr[], int n, int sum)
        {
         int **t;
        t=new int*[n+1];
        for(int i=0;i<=n;i++)t[i]=new int[sum+1];
        for(int i=0;i<=n;i++)
           for(int j=0;j<sum+1;j++)
             t[i][j]=-1;
             return (isissubsetsum(n,arr,sum,t));
        }

    int isissubsetsum(int n,int arr[],int sum,int **t)
{
        if(sum==0)return 1;
        else if(n<=0) return 0;
        else{
            if(t[n][sum]!=-1)return t[n][sum];
            if(arr[n-1]>sum)
            return t[n][sum]=isissubsetsum(n-1,arr,sum,t);
            else
            return t[n][sum]=isissubsetsum(n-1,arr,sum,t)
                              + isissubsetsum(n-1,arr,sum-arr[n-1],t);
        }
    }
```

Top down approach:

```
int perfectSum(int arr[], int n, int sum)
```

```
    {
        int t[n+1][sum+1];
        //memset(t,0,sizeof(t));
        for(int i=0;i<=n;i++)
            for(int j=0;j<=sum;j++)
            {
                if(i==0){
                if(j==0)t[i][j]=1;
                else t[i][j]=0;}
                else{
                if(arr[i-1]<=j)
                t[i][j]=t[i-1][j-arr[i-1]]+t[i-1][j];
                 else
                 t[i][j]=t[i-1][j];

                }
            }
            return t[n][sum];
    }
```

Complexity,time=O(n*sum).

5.Target sum:

Method1:Same as above.

Method 2:

Recursion:

```
int findTargetSumWays(vector<int>&A ,int target) {
        int total=0;
        int n=A.size();
        for(int i=0;i<A.size();i++)total+=A[i];
        if((total+target)%2)return 0;
        else{
            int sum=(target+total)/2;
             int **t;
        t=new int*[n+1];
        for(int i=0;i<=n;i++)t[i]=new int[sum+1];
        for(int i=0;i<=n;i++)
           for(int j=0;j<sum+1;j++)
             t[i][j]=-1;
             return (isissubsetsum(n,A,sum,t));
        }
    }
    int isissubsetsum(int n,vector<int>&A ,int sum,int **t)
{
        if(sum==0)return 1;
        else if(n<=0) return 0;
        else{
            if(t[n][sum]!=-1)return t[n][sum];
            if(A[n-1]>sum)
            return t[n][sum]=isissubsetsum(n-1,A,sum,t);
            else
            return t[n][sum]=isissubsetsum(n-1,A,sum,t)
                           + isissubsetsum(n-1,A,sum-A[n-1],t);
        }
    }
```

Top down approach:

```
int isissubsetsum(int n,vector<int>&A ,int sum)
{
        int t[n+1][sum+1];
          //memset(t,0,sizeof(t));
          for(int i=0;i<=n;i++)
              for(int j=0;j<=sum;j++)
              {
                  if(j==0)t[i][j]=1;
                  else if(i<=0)t[i][j]=0;
                  else{
                  if(A[i-1]<=j)
                  t[i][j]=t[i-1][j-A[i-1]]+t[i-1][j];
                   else
                   t[i][j]=t[i-1][j];

                  }
              }
              return t[n][sum];

   }
```

6.Count the number of subsets with given difference:

Note:same as target sum by making a small difference as

```
      s1-s2=difference;
      s1+s2=total;
```

Add these two…
We will get 2s1=total+difference

s1=(total+diffrence)/2

```
      Int sum=(total+difference)/2;
```

7.Minimum subset sum difference:
1.method 1

```
int minDifference(int arr[], int n)
    {
          int sum=0;
          for(int i=0;i<n;i++)sum+=arr[i];
          int **t;
          t=new int*[n+1];
          for(int i=0;i<n+1;i++)t[i]=new int[sum+1];
          for(int i=0;i<=n;i++)
              for(int j=0;j<=sum;j++)
                  t[i][j]=-1;

           recminDifference(arr,n,0,sum,t);
       }
int recminDifference(int arr[],int n,int s1,int sum,int **t){
          if(n==0)return (abs(sum-2*s1));

          if(t[n][s1]!=-1)return t[n][s1];
          else {
              return
t[n][s1]=min(recminDifference(arr,n-1,s1+arr[n-1],sum,t),
              recminDifference(arr,n-1,s1,sum,t));
          }
       }
```

Method 2:

Get the timetable from the subset sum and then

```
    int mini = INT_MAX;
    for(int i = sum/2 ; i>=0 ; i--){
        if(dp[n][i]){
            mini = min(mini , sum - 2*i);
            break;
        }
    }
    return mini;
```

B.Unbonded knapsack

Here we can process an element more than twice if we have already processed but if we leave an element then we can’t visit it again.

i-1—>>>i

1.Rod cut:

Method 1:Naive approach

A naive solution to this problem is to generate all configurations of different pieces and find the highest-priced configuration. This solution is exponential in terms of time complexity.
If array size is n and length of rod is l,then

Complexity:time=O(l^n)

Method 2:

Recursive approach:

```
int cutRod(int price[], int n) {
        int n1[n];
        for(int i=0;i<n;i++)n1[i]=i+1;
        int **t;
        t=new int*[n+1];
        for(int i=0;i<=n;i++)t[i]=new int[n+1];
        for(int i=0;i<=n;i++)
          for(int j=0;j<=n;j++)
             t[i][j]=-1;

        return cutRodRec(n,price,n1,n,t);

    }

    int cutRodRec(int n,int price[],int l[],int l1,int **t)
    {
        if(l1==0)return 0;
        else if(n<=0)return 0;
        else{
            if(t[n][l1]!=-1)return t[n][l1];
            else{
```

```
            if(l[n-1]<=l1)
            return
t[n][l1]=max(price[n-1]+cutRodRec(n,price,l,l1-l[n-1],t),
            cutRodRec(n-1,price,l,l1,t));
            else
            return t[n][l1]=cutRodRec(n-1,price,l,l1,t);
          }
        }
    }
```

2.Coin change:Max number of ways

Method1:
Same as above.

Method2:

Recursive approach:

```
 int change(int m, vector<int>& arr) {

        int **t;
        int n=arr.size();
        t=new int*[n+1];
        for(int i=0;i<=n;i++)t[i]=new int[m+1];
        for(int i=0;i<=n;i++)
           for(int j=0;j<m+1;j++)
             t[i][j]=-1;
             return (isissubsetsum(n,arr,m,t));
```

```
   }
   int isissubsetsum(int n,vector<int>& arr,int sum,int **t)
{
       if(sum==0)return 1;
       else if(n<=0) return 0;
       else{
          if(t[n][sum]!=-1)return t[n][sum];
          if(arr[n-1]>sum)
          return t[n][sum]=isissubsetsum(n-1,arr,sum,t);
          else
          return t[n][sum]=isissubsetsum(n-1,arr,sum,t)
                            + isissubsetsum(n,arr,sum-arr[n-1],t);
       }
   }
```

3.Minimum number of coins in coin change:

Method1:Nave approach->Same as above

Method2:

Recursive approach:

```
int coinChange(vector<int>& arr, int m) {
       int **t;
       int count=0;
       int n=arr.size();
       t=new int*[n+1];
       for(int i=0;i<=n;i++)t[i]=new int[m+1];
```

```
        for(int i=0;i<=n;i++)
           for(int j=0;j<m+1;j++)
             t[i][j]=-1;
         int a=coinchange1(coins,amount,n,t);
        if(a==INT_MAX-1)return -1;
        else return a;


    }
    int isissubsetsum(int n,vector<int>& arr,int sum,int **t)
{
        if(sum==0)return 0;
        else if(n<=0) return INT_MAX-1;
        else{
            if(t[n][sum]!=-1)return t[n][sum];
            if(arr[n-1]>sum)
            return t[n][sum]=isissubsetsum(n-1,arr,sum,t);
            else
            return t[n][sum]=min(isissubsetsum(n-1,arr,sum,t),
                          1+ isissubsetsum(n,arr,sum-arr[n-1],t));
        }
    }
```

Top down approach:

```
int coinChange(vector<int>& coins, int V) {

       int table[V+1];
       int m=coins.size();
       table[0] = 0;
```

```
    for (int i=1; i<=V; i++)
        table[i] = INT_MAX;

      for (int i=1; i<=V; i++)
    {
        for (int j=0; j<m; j++)
          if (coins[j] <= i)
          {
              int sub_res = table[i-coins[j]];
              if (sub_res != INT_MAX && sub_res + 1 < table[i])
                  table[i] = sub_res + 1;
          }
    }

      if(table[V]==INT_MAX)
        return -1;

    return table[V];
}
```

Complexity,time=O(V*m).
space=O(V).

C.1.LCS:

Method 1:Naive approach
In order to find out the complexity of brute force approach, we need to first know the number of possible different subsequences of a string with length n, i.e., find the number of subsequences with lengths ranging from 1,2,..n-1. Recall

from theory of permutation and combination that number of combinations with 1 element are nC1. Number of combinations with 2 elements are nC2 and so forth and so on. We know that nC0 + nC1 + nC2 + … nCn = 2^n. So a string of length n has 2^n-1 different possible subsequences since we do not consider the subsequence with length 0. This implies that the time complexity of the brute force approach will be O(n * 2^n). Note that it takes O(n) time to check if a subsequence is common to both the strings.

Method 2:

Recursive approach:

```
int lcs(int x, int y, string s1, string s2)
   {
        int **t;
       t=new int*[x+1];
       for(int i=0;i<=x;i++)t[i]=new int[y+1];
       for(int i=0;i<=x;i++)
          for(int j=0;j<=y;j++)
            t[i][j]=-1;
            return (lcsrec(x,y,s1,s2,t));
   }
   int lcsrec(int x,int y,string& s1,string& s2,int **t){
       if(x==0 || y==0)return 0;
       if(t[x][y]!=-1)return t[x][y];
       else{
          if(s1[x-1]==s2[y-1])
          return t[x][y]=1+lcsrec(x-1,y-1,s1,s2,t);
          else
          return
t[x][y]=max(lcsrec(x,y-1,s1,s2,t),lcsrec(x-1,y,s1,s2,t));
       }
    }
```

Top down approach:

```
int lcs(int x, int y, string s1, string s2)
    {
         int **t;
        t=new int*[x+1];
        for(int i=0;i<=x;i++)t[i]=new int[y+1];
        for(int i=0;i<=x;i++)
           for(int j=0;j<=y;j++)
             t[i][j]=-1;
             return (lcsrec(x,y,s1,s2,t));
    }

    int lcsrec(int x,int y,string s1,string s2,int **t){
        for(int i=0;i<=x;i++)
           for(int j=0;j<=y;j++)
               {
                   if(i==0 || j==0)t[i][j]=0;
                   else if(s1[i-1]==s2[j-1])
                   t[i][j]=1+t[i-1][j-1];
                   else
                   t[i][j]=max(t[i-1][j],t[i][j-1]);
               }
               return t[x][y];
    }
```

Complexity,time=O(x*y)=space.

2.Longest common substring:

Method1:Brute force approach

Let m and n be the lengths of the first and second strings respectively.

A simple solution is to one by one consider all substrings of the first string and for every substring check if it is a substring in the second string. Keep track of the maximum length substring. There will be O(m^2) substrings and we can find whether a string is substring on another string in O(n) time . So overall time complexity of this method would be O(n * m2)

Method2:Dynamic programming

Note:keeping track of maximum value by using r.

Top down approach:

```
int longestCommonSubstr (string s1, string s2, int x, int y)
    {
    int r=0;
    for(int i=0;i<=x;i++)
      for(int j=0;j<=y;j++)
      {
         if(i==0 || j==0)t[i][j]=0;
         else{
             if(s1[i-1]==s2[j-1]){
           t[i][j]=1+t[i-1][j-1];
           r=max(r,t[i][j]);}
             else
             t[i][j]=0;
         }
      }
    return r;}
```

Complexity:Time Complexity: O(x*y)

Auxiliary Space: O min(x*y)

3.Print LCS

Method 1:Same as LCS

Method 2:DP

Top down approach:

```
int t[100][100];
string s1,s2;
  void lcsprint(int x,int y){
      stack<char> s;
 for(int i=0;i<=x;i++)
           for(int j=0;j<=y;j++)
               {
                   if(i==0 || j==0)t[i][j]=0;
                   else if(s1[i-1]==s2[j-1])
                   t[i][j]=1+t[i-1][j-1];
                   else
                   t[i][j]=max(t[i-1][j],t[i][j-1]);
               }
         for(int i=x;i>=0;)
           for(int j=y;j>=0;)   {
               if(s1[i-1]==s2[j-1]){s.push(s1[i-1]);i--;j--;}
               else if(t[i-1][j]>t[i][j-1])i--;
               else j--;
           }
           while(!s.empty()){
               char c=s.top();
                           s.pop();
```

```
                        cout<<c;
            }
              }

int main() {
        cin>>s1>>s2;
        lcsprint(s1.length(),s2.length());
        return 0;
}
```

Recursive approach:Here we r storing lcs in stack.

```
void  lcsprint1(int x,int y){
        if(x==0 || y==0)return;
        else{
                if(s1[x-1]==s2[y-1]){
                        s.push(s1[x-1]);
                        return lcsprint1(x-1,y-1);
                }
                else if(t[x-1][y]>t[x][y-1])return lcsprint1(x-1,y);
                else
```

```
            return lcsprint1(x,y-1);
        }
}
```

Complexity,time=O(x*y)=space.

4.Length smallest supersequence:

Given two strings str1 and str2, the task is to find the length of the shortest string that has both str1 and str2 as subsequences.

E,g. A:abss

B:assd

Here we will compare both strings if the char are same then we will add that char once otherwise add both.

Supersequence:abssd

Here we realize that longest common subsequence is same both in A & B.

So Length of small supersequence=length(A+B)-length(LCS).

Method 1:Bruteforce approach

Naive approach will be same as LCS

Complexity:time=O(n*2^n).

Method 2:Dynamic programming

```
int shortestCommonSupersequence(string s1, string s2, int
x, int y)
   {
         int **t;
       t=new int*[x+1];
       for(int i=0;i<=x;i++)t[i]=new int[y+1];
            int lcs=lcsrec(x,y,s1,s2,t);
            return(s1.length()+s2.length()-lcs);



   }
   int lcsrec(int x,int y,string s1,string s2,int **t){
       for(int i=0;i<=x;i++)
          for(int j=0;j<=y;j++)
              {
                  if(i==0 || j==0)t[i][j]=0;
                  else if(s1[i-1]==s2[j-1])
                  t[i][j]=1+t[i-1][j-1];
                  else
                  t[i][j]=max(t[i-1][j],t[i][j-1]);
              }
            return t[x][y];
    }
```

Complexity,time=O(x*y)=space.

5.Print smallest common supersequence:

```
string shortestCommonSupersequence(string s1, string s2) {
        string ans="",temp="";
        int x=s1.size();
        int y=s2.size();
        int **t;
        t=new int*[x+1];
        for(int i=0;i<=x;i++)t[i]=new int[y+1];
        for(int i=0;i<=x;i++)
           for(int j=0;j<=y;j++)
             t[i][j]=-1;
             int l=(lcsrec(x,y,s1,s2,t));
        int i=x,j=y;
    while(i>0 && j>0)
            {
               if(s1[i-1]==s2[j-1]){
                   ans+=s1[i-1];
```

```
                i--;j--;
            }
           else if(t[i-1][j]>t[i][j-1]){ans+=s1[i-1];i--;}
            else {ans+=s2[j-1];j--;}

        }
     while(j>0){ans+=s2[j-1];j--;}
     while(i>0){ans+=s1[i-1];i--;}
      reverse(ans.begin(),ans.end());
      return ans;

  }
  int lcsrec(int x,int y,string& s1,string& s2,int **t){
      if(x==0 || y==0)return 0;
      if(t[x][y]!=-1)return t[x][y];
      else{
          if(s1[x-1]==s2[y-1])
          return t[x][y]=1+lcsrec(x-1,y-1,s1,s2,t);
          else
          return
t[x][y]=max(lcsrec(x,y-1,s1,s2,t),lcsrec(x-1,y,s1,s2,t));
      }
```

```
    }
```

5.Min no. of deletions and insertions to convert a string from one form to other:

Method 1:Brute force

A simple approach is to consider all subsequences of str1 and for each subsequence calculate minimum deletions and insertions so as to transform it into str2. A very complex method and the time complexity of this solution is exponential

A:sea

B:eat

E,g: sea—>eat,Here we need to delete ‘s’ and add ‘t’,so total operations=2.First find length(lcs) and then

no. of delete operations= length(A)-length(lcs).

no. of insertions operations=length(B)-length(lcs).

Total operations=delete +insert.

Method 1:Brute force approach

Same as LCS.Complexity is also same.

Method 2:DP

```
int minDistance(string s1, string s2) {
        int x=s1.length();
        int y=s2.length();
         int **t;
        t=new int*[x+1];
        for(int i=0;i<=x;i++)t[i]=new int[y+1];
        for(int i=0;i<=x;i++)
           for(int j=0;j<=y;j++)
              t[i][j]=-1;
              int lcs=lcsrec(x,y,s1,s2,t);
        return(x+y-2*lcs); }              Complexity=same as above.
```

6.Print smallest supersequence:

Method1:Running on letcode

```
 string shortestCommonSupersequence(string s1, string s2)
{
        string ans="",temp="";
        int x=s1.size();
        int y=s2.size();
        int **t;
        t=new int*[x+1];
        for(int i=0;i<=x;i++)t[i]=new int[y+1];
        for(int i=0;i<=x;i++)
           for(int j=0;j<=y;j++)
```

```
         t[i][j]=-1;
         int l=(lcsrec(x,y,s1,s2,t));
    int i=x,j=y;
    while(i>0 && j>0)
       {
          if(s1[i-1]==s2[j-1]){
              ans+=s1[i-1];
               i--;j--;
          }
          else if(t[i-1][j]>t[i][j-1]){ans+=s1[i-1];i--;}
          else {ans+=s2[j-1];j--;}
      }

   while(j>0){ans+=s2[j-1];j--;}
   while(i>0){ans+=s1[i-1];i--;}
    reverse(ans.begin(),ans.end());
    return ans;

}
int lcsrec(int x,int y,string& s1,string& s2,int **t){
    if(x==0 || y==0)return 0;
    if(t[x][y]!=-1)return t[x][y];
```

```
        else{
            if(s1[x-1]==s2[y-1])
            return t[x][y]=1+lcsrec(x-1,y-1,s1,s2,t);
            else
            return
t[x][y]=max(lcsrec(x,y-1,s1,s2,t),lcsrec(x-1,y,s1,s2,t));
        }
    }
```

7.Length of longest palindromic subsequence:

Method 1:Naive approach

The naive solution for this problem is to generate all subsequences of the given sequence and find the longest palindromic subsequence. This solution is exponential in terms of time complexity.

Complexity will be O(n*2^n)

Method 2:DP

Since in this problem number of input strings is 1 but in LCS there r 2 input strings,we can generate 2nd one by reversing the 1st one, rest will be same.

e,g:A=aagbaa

B=reverse(A)=aabgaa,then apply LCS

Complexity will be same as LCS.

8.Minimum deletions in a string too make it a palindrome:

Method 1:same as LCS—>naive approach

Method 2:DP

Since there is only 1 input string to get another one we need reverse the 1st string,then get the LCS of 2 strings.

No. of deletions=length(any string)-length(lcs);

Complexity same as LCS.

9.Length of longest repeating subsequence:

Method 1:same as LCS

Method 2:DP

Here only 1 string is given getting 2nd one we r using again first string ,means 1 and 2 string is same now getting longest common repeating subsequence only one change in the code of LCS.

if(s1[i-1]==s2[j-1] && i!=j)----->here we r finding 2 occurrences of an element,then repeating is possible.

```
int LongestRepeatingSubsequence(string s1){
        int **t;
        int x=s1.length();
        t=new int*[x+1];
        for(int i=0;i<=x;i++)t[i]=new int[x+1];
        for(int i=0;i<=x;i++)
           for(int j=0;j<=x;j++)
             t[i][j]=-1;
             return (lcsrec(x,x,s1,t));
    }
    int lcsrec(int x,int y,string s1,int **t){
        if(x==0 || y==0)return 0;
        if(t[x][y]!=-1)return t[x][y];
        else{
            if((x-1)!=(y-1) && s1[x-1]==s1[y-1] )
            return t[x][y]=1+lcsrec(x-1,y-1,s1,t);
            else
            return
t[x][y]=max(lcsrec(x,y-1,s1,t),lcsrec(x-1,y,s1,t));}}
```

10.Pattern matching subsequence:

Here we need show whether 1 complete string is a subsequence of another one or not.

1.get LCS

2.return(LCS==min(s1.length(),s2.length());

11.Minimum number of insertions to make a string palindrome

—>>>>Same as no.of deletions.

D.1.Matrix chain multiplication:

Before approaching any problem in mcm :

1.select the value of i & j;

2.Base condition;

3.K loop

Format for MCM like problems:

i=1,j=n-1;

```
Int solve(int arr[],int i,int j){
if(i>j)return 0;
Else
for(int k=i;k<j;k++){
temp_ans=solve(arr,i,k)
            +           —-->here it may be anything
                solve(arr,k+1,j)


}
Return ans;
}
```

Method1:Brute force approach

We need to find all the combinations which is similar like to find all possible parentheses for an expression.Just for finding all parentheses we need O(n*2^n) according to katlons no.,then we need to compare all so

Complexity,time=O(n*2^n)

Method2:DP

```
class Solution{
public:
    int t[1001][1001];
    int matrixMultiplication(int N, int arr[])
    {
```

```
    for(int i=0;i<1001;i++)
        for(int j=0;j<1001;j++)
           t[i][j]=-1;

    int i=1,j=N-1;
    return(matrixMultiplicationRec(arr,i,j,t));
}

int matrixMultiplicationRec(int arr[],int i,int j,int t[][1001]){
    int m=INT_MAX;
    if(i>=j)return 0;
    else{
        if(t[i][j]!=-1)return t[i][j];
        else{
            int temp;
            for(int k=i;k<j;k++)
            {
                temp =matrixMultiplicationRec(arr,i,k,t)

+matrixMultiplicationRec(arr,k+1,j,t)+arr[i-1]*arr[k]*arr[j];

                m=min(m,temp);
            }
             return t[i][j]=m;
        }
    }
    return m;
}
};
```

Complexity,time=O(n^3),space.=O(n^2)

2.Palindrome partitioning:

Method1:Bruteforce approach

Same like with MCM with addition of checking whether a string is palindrome or not.

Complexity,time=O(n^2*2^n)

Method 2:DP

```
class Solution{
public:
    int t[501][501];
    int palindromicPartition(string& str)
    {
        int N=str.length();
```

```
for(int i=0;i<501;i++)
        for(int j=0;j<N+1;j++)
           t[i][j]=-1;

    int i=0,j=N-1;
    return(matrixMultiplicationRec(str,i,j,t));


}
int matrixMultiplicationRec(string& arr,int i,int j,int t[][]){
    int m=INT_MAX;
    if(i>=j)return 0;
    else if(palindromeCheck(arr,i,j))return 0;
    else{
        if(t[i][j]!=-1)return t[i][j];
        else{
            int temp=0;
            for(int k=i;k<j;k++)
            {
                temp =matrixMultiplicationRec(arr,i,k,t)
                +matrixMultiplicationRec(arr,k+1,j,t)+1;

                m=min(m,temp);
            }
            return t[i][j]=m;
        }
    }
}
bool palindromeCheck(string& arr,int i,int j){
    while(i<=j){
        if(arr[i]!=arr[j])return false;
        i++;j--;
    }
    return true;
}}; Complexity,time=O(n^3).
```

Method3:Hash map

```
int palindromicPartition(string str)
    {
        int N=str.length();
        unordered_map<string, int> m;

        int i=0,j=N-1;
        return(palindromicPartitionRec(str,i,j,m));
    }
    string convert(int i,int j){
        return to_string(i)+""+to_string(j);
    }
    bool palindrome(string s,int i,int j){
        while(i<=j){
            if(s[i]!=s[j])return false;
            j--;i++;
        }
        return true;
    }
    int palindromicPartitionRec(string s,int i,int
j,unordered_map<string,int> &m){
        if(i>j)return 0;
        string ij=convert(i,j);

        if(m.find(ij)!=m.end())
        return m[ij];
        if(i==j){
            m[ij]=0;
            return m[ij];
```

```
}
if(palindrome(s,i,j)){
     m[ij]=0;
    return m[ij];
}
int minimum=INT_MAX;
for(int k=i;k<j;k++){
    int left_min=INT_MAX;
     int right_min=INT_MAX;
     string left=convert(i,k);
     string right=convert(k+1,j);

     if(m.find(left)!=m.end())
     left_min=m[left];
     if(m.find(right)!=m.end())
     right_min=m[right];

     if(left_min==INT_MAX)
     left_min=palindromicPartitionRec(s,i,k,m);
     if(right_min==INT_MAX)
     right_min=palindromicPartitionRec(s,k+1,j,m);
```

```
            minimum=min(minimum,left_min+right_min+1);
        }
        m[ij]=minimum;
        return m[ij];

    }
```
Complexity,time=O(n^2).

3.Egg Dropping problem:

```
int t[201][201];
    int eggDrop(int n, int k)
    {
        memset(t,-1,sizeof(t));
        return eD(n,k);
    }
    int eD(int e,int f){
        if(e==1)return f;
        if(f==0 || f==1)return f;
        if(t[e][f]!=-1)return t[e][f];
        int mn=INT_MAX;int temp=0;
        for(int k=1;k<=f;k++){
          int temp=1+max(eD(e-1,k-1),eD(e,f-k));
        mn=min(mn,temp);
        }
        return t[e][f]=mn;

    }
```

4.Minimum Cost to Cut the Stick:

```
#include<bits/stdc++.h>

int mincost(int i,int j,vector<int> &cuts,vector<vector<int>>&
dp)
{
    if(i>j)return 0;
    if(dp[i][j]!=-1)return dp[i][j];
    int mini=INT_MAX;
    for(int k=i;k<=j;k++)
    {
        int cost=mincost(i,k-1,cuts,dp) +
mincost(k+1,j,cuts,dp)+cuts[j+1]-cuts[i-1];
```

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


int cost(int n, int c, vector<int> &cuts){
    cuts.push_back(n);
    cuts.insert(cuts.begin(),0);
    sort(cuts.begin(),cuts.end());
    vector<vector<int>> dp(c+1,vector<int>(c+1,-1));
    return mincost(1,c,cuts,dp);
}
```

5.Mining Diamonds:

```
#include<bits/stdc++.h>
int maxcoins(int i,int j,vector<int>& a,vector<vector<int>>&
dp)
{
    if(i>j)return 0;
    if(dp[i][j]!=-1)return dp[i][j];
    int maxi=INT_MIN;
    for(int k=i;k<=j;k++)
    {
        int
cost=a[i-1]*a[k]*a[j+1]+maxcoins(i,k-1,a,dp)+maxcoins(k+1,j,a
,dp);
        maxi=max(maxi,cost);
    }
    return dp[i][j]=maxi;
}



int maxCoins(vector<int>& a)
{
    int n=a.size();
```

```
        a.push_back(1);
    a.insert(a.begin(),1);
    vector<vector<int>> dp(n+1,vector<int>(n+1,-1));
    return maxcoins(1,n,a,dp);
}
```

6.Boolean Expression true:

```
#include<bits/stdc++.h>
int evaluate(int i,int j,bool istrue,string &
exp,vector<vector<vector<int>>> &dp)
{
    if(i>j)return false;
    if(dp[i][j][istrue]!=-1)return dp[i][j][istrue];
    if(i==j)
    {
        if(istrue)return exp[i]=='T';
        else return exp[i]=='F';
    }
    int ways=0;
    for(int k=i+1;k<=j-1;k=k+2)
```

```
{
    long int lt=evaluate(i,k-1,1,exp,dp);
    long int lf=evaluate(i,k-1,0,exp,dp);
    long int rt=evaluate(k+1,j,1,exp,dp);
    long int rf=evaluate(k+1,j,0,exp,dp);

    if(exp[k]=='&')
    {
        if(istrue)ways+=rt*lt;
        else ways+=rt*lf+rf*lt+rf*lf;
    }
    else if(exp[k]=='|')
    {
        if(istrue)ways+=rt*lt+rt*lf+rf*lt;
        else ways+=rf*lf;
    }
    else {
        if(istrue)ways+=rt*lf+rf*lt;
        else ways+=rt*lt+lf*rf;
    }
}
return dp[i][j][istrue]=ways;
```

```
}
int evaluateExp(string & exp) {
    int n=exp.size();
    vector<vector<vector<int>>>
dp(n,vector<vector<int>>(n,vector<int>(2,-1)));
    return evaluate(0,n-1,1,exp,dp);
}
```

7.palindrome partitioning :

```
#include<bits/stdc++.h>
bool ispalindrome(string temp,int i,int j)
{
    //int i=0,j=temp.size()-1;
    while(i<j)
        if(temp[i++]!=temp[j--])return false;
    return true;
}
int palindrome_min_cuts(int i,int n,vector<int> &d,string
&str)
```

```
{
    if(i==n)return 0;
    if(d[i]!=-1)return d[i];
    string temp="";
    int mincost=INT_MAX;
    for(int k=i;k<n;k++)
    {
    //temp+=d[i];
    if(ispalindrome(str,i,k))
        {
        int cost=1+palindrome_min_cuts(k+1,n,d,str);
        mincost=min(mincost,cost);
        }
    }
    return d[i]=mincost;
}

int palindromePartitioning(string str) {
   int n=str.size();
   vector<int> d(n+1,-1);
    return palindrome_min_cuts(0,n,d,str)-1;
}
```