Index - 6

Dynamic Programming

1) Max Sum without taking Adjacents

```
int findMaxSum(int a[], int n) {
      int dp[] = new int[n+1];
      Arrays.fill(dp,-1);

      return stealHouse(a,0 ,n ,dp);
   }
   int stealHouse(int a[],int curIndex, int n, int dp[]){
      if(curIndex>=n)
         return 0;
      if(dp[curIndex]!=-1)
         return dp[curIndex];

      int stealCur = a[curIndex] + stealHouse(a, curIndex+2, n, dp);
      int skipCur = stealHouse(a, curIndex+1,n, dp);
      dp[curIndex] = Math.max(stealCur, skipCur);
      return dp[curIndex];
   }
```

T.C = O(n)
S.C = O(n)

2) Edit Distance

```
public int editDistance(String s, String t) {
      int dp[][] = new int[s.length()][t.length()];
      for(int r[]:dp)
         Arrays.fill(r,-1);

      return findOperations(s ,t, 0, 0, dp);
   }
   int findOperations(String s1,String s2, int i1 ,int i2 ,int dp[][]){
      if(i1==s1.length())
         return s2.length() - i2;
      else if(i2==s2.length())
         return s1.length() - i1;

      if(dp[i1][i2]!=-1)
         return dp[i1][i2];
      if(s1.charAt(i1)==s2.charAt(i2))
         return dp[i1][i2] = findOperations(s1,s2,i1+1,i2+1, dp);

      int insert = findOperations(s1,s2,i1+1,i2, dp);
      int delete = findOperations(s1,s2,i1,i2+1, dp);
      int replace = findOperations(s1,s2,i1+1,i2+1, dp);

      dp[i1][i2] = 1 + Math.min(insert,Math.min(delete, replace));
      return dp[i1][i2];
   }
```

T.C = O(s1.length() * s2.length())
S.C = O(s1.length() * s2.length())

3) Longest Common Subsequence

```
static int lcs(int p, int q, String s1, String s2){

      int dp[][] = new int[p][q];

      for(int r[]:dp)
         Arrays.fill(r,-1);

      return findLcs(s1, s2, 0, 0, dp);
   }
   static int findLcs(String s1,String s2, int i1, int i2, int dp[][]){
      if(i1==s1.length() || i2==s2.length())
         return 0;
```

```
        if(dp[i1][i2]!=-1)
            return dp[i1][i2];

        int c1=0;
        if(s1.charAt(i1)==s2.charAt(i2))
            c1 = 1+findLcs(s1,s2,i1+1,i2+1, dp);

        int c2 = findLcs(s1,s2,i1+1, i2, dp);
        int c3 = findLcs(s1,s2,i1,i2+1, dp);

        return dp[i1][i2] = Math.max(c1,Math.max(c2,c3));
    }
```

T.C = O(s1.length() * s2.length())
S.C = O(s1.length() * s2.length())

4) Minimum number of deletions and insertions to convert one string to another

Here we need to use longest common subsequence.
Number of characters to be deleted in first string is,
deletion = s1.length() - lcs; (This gives extra characters)
Number of characters to be inserted in first string is,
insertion = s2.length() - lcs; (This gives extra characters to be deleted)

return insertion + deletion;

```
    public int minOperations(String str1, String str2)
    {
        int dp[][] = new int[str1.length()][str2.length()];
        for(int r[]:dp)
            Arrays.fill(r,-1);

        int commonSubsequence = lcs(str1, str2, 0, 0, dp);
        int insertion = str2.length() - commonSubsequence;
        int deletion = str1.length() - commonSubsequence;

        return insertion+deletion;
    }
```

T.C = O(s1.length() * s2.length())
S.C = O(s1.length() * s2.length())

5) 0 - 1 Knapsack Problem

```
static int knapSack(int w, int wt[], int val[], int n)
    {
        int dp[][] = new int[n][w+1];
        for(int r[]:dp)
            Arrays.fill(r,-1);
        int res = pickElements(val, wt, w, 0, dp);

        return res>0?res:0;
    }
    static int pickElements(int val[],int wt[],int w, int index, int dp[][]){
        if(index>=val.length || w<=0)
            return 0;

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

        int pickCur = 0;
        if(wt[index]<=w)
            pickCur = val[index] + pickElements(val, wt, w-wt[index],index+1, dp);
        int skipCur = pickElements(val ,wt ,w ,index+1 ,dp);

        return dp[index][w] = Math.max(pickCur, skipCur);
    }
```

T.C = O(n * weight)
S.C = O(n * weight)

6) Longest Palindromic Subsequence

```
public int longestPalinSubseq(String s)
    {
        int dp[][] = new int[s.length()][s.length()];
        for(int r[]:dp)
            Arrays.fill(r,0);
        return lps(s,0,s.length()-1, dp);
    }
    public int lps(String s ,int start ,int end, int dp[][]){
        if(start>end)
            return 0;
        if(start==end)
            return 1;
        if(dp[start][end]!=0)
            return dp[start][end];

        int c1 = 0;
        if(s.charAt(start)==s.charAt(end))
            c1 = 2 + lps(s, start+1, end-1, dp);
        int c2 = lps(s, start+1, end, dp);
        int c3 = lps(s, start, end-1, dp);

        return dp[start][end] = Math.max(c1,Math.max(c2,c3));
    }
```

T.C = O(n * n)
S.C = O(n * n)

7) Longest Palindrome in a String (Print the Substring)

Expand from middle

```
public String longestPalindrome(String s){
      if(s.length()<=1)
         return s;

      int start = 0, end=0;
      int i = 0;
      while(i<s.length()){
         int len1 = getLength(s, i,i);
         int len2 = getLength(s, i,i+1);
         int length = Math.max(len1, len2);

         if((end-start+1)<length){
            start = i - (length-1)/2;
            end = i + (length)/2;
         }
         i++;
      }
      return s.substring(start, end+1);
   }

   int getLength(String s, int i, int j){

      while(i>=0 && j<s.length() && s.charAt(i)==s.charAt(j)){
         i--;
         j++;
      }
      return j-i-1;
   }
```

T.C = O(n^2)
S.C = O(1)

8) Maximum path sum in matrix from 0th row to n-1th row

There are only three possible moves from a cell Matrix[r][c].

1. Matrix [r+1] [c]
2. Matrix [r+1] [c-1]
3. Matrix [r+1] [c+1]

```
static int maximumPath(int n, int a[][])
   {

      for(int i=1;i<a.length; i++){
         for(int j=0;j<a[0].length; j++){
            if(j==0)
               a[i][j] = a[i][j] + Math.max(a[i-1][j],a[i-1][j+1]);
            else if(j==a[0].length-1)
```

```
                a[i][j] = a[i][j] + Math.max(a[i-1][j],a[i-1][j-1]);
            else
                a[i][j] = a[i][j] + Math.max(a[i-1][j],Math.max(a[i-1][j-1],a[i-1][j+1]));
        }
    }
    int max = a[a.length-1][0];
    for(int i=1;i<a[0].length; i++)
        max = Math.max(max, a[a.length-1][i]);

    return max;
}
```

T.C = O(n^2)
S.C = O(n^2)

9) Longest Increasing Subsequence

i. **Use backTracking, for every element there is 2 possibilities either it is part of subsequence or not. Finally return the answer.**
    T.C = O(2^n)
    S.C = O(n)

ii. Use Dynamic Programming

```
static int longestSubsequence(int size, int a[])
    {
        int dp[] = new int[size];
        Arrays.fill(dp, 1);

        for(int i=1;i<size; i++){
            for(int j=i-1;j>=0;j--){
                if(a[i]>a[j] && dp[i]<=dp[j]){
                    dp[i] = dp[j]+1;
                }
            }
        }
        int max = 1;
        for(int i=0;i<size; i++)
            max = Math.max(max, dp[i]);
        return max;
    }
```

T.C = O(n^2)
S.C = O(n)

iii. Using Binary Search Function(This Only Returns the length)

Store the subsequence with minimum values in it. Finally return it's size.

Arrays.binarySearch(int[], int start, int end, int key);
return key position if it is present else it returns (-(positionTobeInserted)-1)

Similarly Collections.binarySearch(List<>, int start, int end, key);
Same logic.

```
static int longestSubsequence(int size, int a[])
   {
      int dp[] = new int[size];
      int index = -1;
      int i = 0;

      while(i<size){
         if(index==-1 || dp[index]<a[i])
            dp[++index] = a[i];
         else if(dp[index]>a[i]){
            int position = Arrays.binarySearch(dp ,0, index ,a[i]);
            position = Math.abs(position+1);
            dp[position] = a[i];
         }
         i++;
      }

      return index+1;
   }
```

T.C = O(nlogn)
S.C = O(n)

10) Number of Longest Increasing Subsequence

Use Dp approach and simultaneously count

```
public int findNumberOfLIS(int[] a) {

      int dp[] = new int[a.length];
      Arrays.fill(dp, 1);
      int count[] = new int[a.length];
      Arrays.fill(count,1);



      for(int i=1; i<a.length; i++)
      {
        for(int j=i-1; j>=0; j- -)
        {
                if(a[i]>a[j])
                {
                        if(dp[i]<=dp[j])
                        {
                                dp[i] = dp[j] +1;
                                count[i] = count[j];
                        }
                        else if(dp[j]+1==dp[i])
                                count[i] = count[i] + count[j];   // Storing at last element of sq
                }
            }
        }

        int max = 1;
        int ans = 0;
        for(int i=0;i<a.length ;i++)
            max = Math.max(max, dp[i]);

        for(int i=0;i<a.length; i++)
            if(dp[i]==max)
                ans+=count[i];

        return ans;
    }
```

T.C = O(n^2)
S.C = O(n)

11) Subset Sum DP

```
static boolean checkTheSum(int a[],int n, int sum){
        boolean dp[][] = new boolean[n+1][sum+1];

        for(int i=0;i<=sum; i++)
            dp[0][i] = false;

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

        for(int i=1;i<=n;i++){
            for(int j=1;j<=sum; j++){
                if(j<a[i-1])
                    dp[i][j] = dp[i-1][j];
                else
                    dp[i][j] = dp[i-1][j] || dp[i-1][j-a[i-1]];
            }
        }
        return dp[n][sum];
    }
```

T.C = O(n*sum)
S.C = O(n*sum)

12) Count all possible paths from top left to bottom right

```
int mod = 1000000007;
    long numberOfPaths(int m, int n)
```

```
{
    long dp[][] = new long[m][n];
    return findPath(m-1, n-1, dp);
}
long findPath(int m, int n, long dp[][]){
    if(m==0 || n==0)
        return 1;
    if(dp[m][n]!=0)
        return dp[m][n];

    dp[m][n] =  (findPath(m-1 ,n, dp)%mod + findPath(m ,n-1, dp)%mod)%mod;
    return dp[m][n];
}
```

T.C = O(n*m)
S.C = O(n*m)

13) Special Matrix(Find Total number of paths with some blocked Cells)

```
public int FindWays(int n, int m, int[][] blocked_cells)
  {
      int a[][] = new int[n][m];
      for(int i=0;i<blocked_cells.length; i++){
          int x = blocked_cells[i][0]-1;
          int y = blocked_cells[i][1]-1;
          a[x][y] = -1;
      }

      int mod = 1000000007;
      boolean flag = false;
      for(int i=0;i<m;i++){
          if(a[0][i]==-1){
              flag = true;
              continue;
          }
         a[0][i] = !flag?1:0;
      }
      flag = false;
      for(int i=0;i<n;i++){
          if(a[i][0]==-1){
              flag = true;
              continue;
          }
         a[i][0] = !flag?1:0:
      }

      for(int i=1;i<n;i++){
          for(int j=1;j<m; j++){
              if(a[i][j]==-1)
                  continue;
              int top = a[i-1][j]==-1?0:a[i-1][j];
```

```
                int left = a[i][j-1]==-1?0:a[i][j-1];

                a[i][j] = (top%mod + left%mod)%mod;
            }
        }
        return a[n-1][m-1]==-1?0:a[n-1][m-1]%mod;
    }
```

T.C = O(n*m)
S.C = O(n*m)

14) Coin Change(Find All Combinations)

i. Using 2D array

```
 public long count(int s[], int m //s.length, int n // this is Sum)
    {
        long dp[][] = new long[m][n+1];

        for(int i=0;i<=n;i++)
            if(i%s[0]==0)
                dp[0][i] = 1;
            else
                dp[0][i] = 0;

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

        for(int i=1;i<m;i++){
            for(int j=1;j<=n; j++){
                if(j<s[i])
                    dp[i][j] = dp[i-1][j];
                else
                    dp[i][j] = dp[i-1][j] + dp[i][j-s[i]];
            }
        }

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

T.C = O(n*sum)
S.C = O(n*sum)

ii. Using 1D array

```
long count(int s[], int m, int n)
{
        long dp[] = new long[n+1];
        Arrays.fill(dp, 0);
        dp[0] = 1;

        for(int i=0; i<s.length; i++)
```

```
                for(int j=s[i]; j<=n; j++)
                        dp[j] = dp[j] + dp[j-s[i]];

        return dp[n];
}
```

T.C = O(n*sum)
S.C = O(n)

15) Coin Change(Find Minimum Number of coins for the given sum)

i. Iterative

```
public int coinChange(int[] coins, int amount) {

      int n = coins.length;
      int dp[][] = new int[n][amount+1];

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

      for(int i=0;i<=amount; i++){
         if(i%coins[0]==0)
            dp[0][i] = i/coins[0];
         else
            dp[0][i] = Integer.MAX_VALUE-10;
      }

      for(int i=1;i<n;i++){
         for(int j=1;j<=amount; j++){
            if(j<coins[i])
               dp[i][j] = dp[i-1][j];
            else
               dp[i][j] = Math.min(dp[i-1][j],dp[i][j-coins[i]]+1);
         }
      }

      return dp[n-1][amount]==Integer.MAX_VALUE-10?-1:dp[n-1][amount];
   }
}
```

ii. Recursive

```
public int coinChange(int[] coins, int amount)
{
        int n = coins.length;
        int dp[][] = new int[n][amount+1];

        int ans = findMinCoins(coins, 0, amount, dp);
        if(ans>=100000)
                ans = -1;
        return ans;
}
public int findMinCoins(int a[], int index, int amount, int dp[][])
{
        if(amount==0)
                return 0;
        if(amount<0 || index>=a.length)
                return 100000;

        if(dp[index][amount]!=0)
                return dp[index][amount];

        int pick = 100000;
        if(a[index]<=amount)
                pick = 1 + findMinCoins(a, index, amount-a[index], dp);

        int skip = findMinCoins(a, index+1, amount, dp);

        return dp[index][amount] = Math.min(pick, skip);
}
```

T.C = O(n*sum)
S.C = O(n*sum)

16) Longest Common Substring

```
int longestCommonSubstr(String s1, String s2, int n, int m){

      int dp[][] = new int[n+1][m+1];

      int res = 0;

      for(int i=0;i<=n;i++){
         for(int j=0;j<=m; j++){
            if(i==0 || j==0)
               dp[i][j] = 0;
            else if(s1.charAt(i-1)==s2.charAt(j-1)){
               dp[i][j] = dp[i-1][j-1]+1;
               res = Math.max(res, dp[i][j]);
            }
            else
               dp[i][j] = 0;
         }
      }
      return res;
}
```

T.C = O(n*m)
S.C = O(n*m)

17) Count ways to reach the n'th stair

We can climb either 1 step or 2 step at a time

```
int countWays(int m)
   {
      if(m==1 || m==2)
         return m;

      int dp[] = new int[m];

      dp[0] = 1;
      dp[1] = 2;
      int mod = 1000000007;
      for(int i=2;i<m;i++)
         dp[i] = (dp[i-1]%mod+dp[i-2]%mod)%mod;

      return dp[m-1]%mod;
   }
```

T.C = O(m)
S.C = O(m)

18) Nth Fibonacci Number

```
static int mod = 1000000007;
   static long nthFibonacci(long n){

      long dp[] = new long[(int)n+1];
      return fibonacci((int)n, dp);
   }
   static long fibonacci(int n, long dp[]){
      if(n==0 || n==1)
         return n;

      if(dp[n]!=0)
         return dp[n];

      return dp[n] = (fibonacci(n-1, dp)%mod+fibonacci(n-2, dp)%mod)%mod;
   }
```

T.C = O(n)
S.C = O(n)

19) Minimum number of jumps to reach the end of the array

i. Min jumps to reach any cell is the min jump from any index+1.

```
static int minJumps(int[] a){
      if(a.length<1)
         return 0;
```

```
        if(a[0]==0)
           return -1;

        int jumps[] = new int[a.length];
        Arrays.fill(jumps,Integer.MAX_VALUE);
        jumps[0] = 0;

        for(int i=1;i<a.length; i++){
           for(int j=0;j<i; j++){
              if(i<=a[j]+j && jumps[j]!=Integer.MAX_VALUE){
                 jumps[i] = Math.min(jumps[i],jumps[j]+1);
                 break;
              }
           }
        }
        return jumps[a.length-1]==Integer.MAX_VALUE?-1:jumps[a.length-1];
     }
```

T.C = O(n^2)
S.C = O(n)

ii. Optimal Solution

For every index find the farthest distance we can jump. So it says that we can reach any index from i+1 to the farthest. Now from i+1 to farthest find the next farthest we can jump and when we reach the previous farthest it is sure that we need one more jump. So add one and make the end as the next farthest . Do it till n-1. Because we are not worried about the jump at a[n-1]. This gives the min jumps.

```
static int jumps(int a[], int n)
{
        if(a.length<=1)
                return 0;
        if(a[0]==0)
                return -1;

        int currentEnd = 0, farthest = 0;
        int minJumps = 0;

        for(int i=0; i<n-1; i++)
        {
                //Find the farthest distance from cur index
                farthest = Math.max(farthest, i+a[i]);

                // add one jump if we are ant end of the array
                if(i==currentEnd)
                {
                        minJumps++;
                        currentEnd = farthest;
                }
        }

        return currentEnd>=n-1?minJumps:-1;
}
```
T.C = O(n)
S.C = O(1)

20) Jump Game

Check if we can reach last index of the array from 0

```
int lastReachableIndex = 0;
      for(int i=0;i<a.length; i++){
         if(lastReachableIndex>=i)
            lastReachableIndex = Math.max(lastReachableIndex, i+a[i]);
      }


        // Can iterate from backwards
      // int lastReachableIndex = n-1;
      // for(int i=n-1;i>=0;i--)
      //     if(a[i]+i>=lastReachableIndex)
      //         lastReachableIndex = i;

      return lastReachableIndex>=n-1?1:0;
}
```

T.C = O(n)
S.C = O(1)

21) Max length chain

You are given N pairs of numbers. In every pair, the first number is always smaller than the second number. A pair (c, d) can follow another pair (a, b) if b < c. Chain of pairs can be formed in this fashion. You have to find the longest chain which can be formed from the given set of pairs.

```
int maxChainLength(Pair arr[], int n)
   {

      Arrays.sort(arr, (a, b)->a.y-b.y);
      int count = 1;
      Pair p = arr[0];

      for(int i=1;i<n;i++){
         if(arr[i].x>p.y){
            count++;
            p = arr[i];
         }
      }
      return count;
```

```
    }
```

T.C = O(nlogn)
S.C = (1)

22) Player with max score

Player can prick first or last element of an array. We are playing with computer and it picks the maximum value of first or last element. Find out if player1 can win with computer.

If we pick max(left, right) then obviously computer is going pic max too and will give the minimum of the left and right.
So apply the logic.

```
static Boolean is1winner(int n, int a[]){

        int sum = 0;
        for(int i=0;i<n;i++)
            sum+=a[i];

        int dp[][] = new int[n][n];
        int p1Score = maxPoints(a, 0,n-1, dp);
        int p2Score = sum - p1Score;


        return p1Score>p2Score;
    }

    static int maxPoints(int a[],int left, int right, int dp[][]){
        if(left>right)
            return 0;
        if(left==right)
            return a[left];

        if(dp[left][right]!=0)
            return dp[left][right];
        int pickFirst = a[left] + Math.min(maxPoints(a, left+2, right, dp), maxPoints(a, left+1,
                                                                        right-1, dp));
        int pickLast = a[right] + Math.min(maxPoints(a, left+1, right-1, dp),maxPoints(a,  left,
                                                                        right-2, dp));

        return dp[left][right] = Math.max(pickFirst, pickLast);
    }
```

T.C = O(n*n)
S.C = (n*n)

23) Maximize The Cut Segments

Given an integer **N** denoting the Length of a line segment. You need to cut the line segment in such a way that the cut length of a line segment each time is either **x** , **y** or **z.** Here x, y, and z are integers.
After performing all the cut operations, your **total number of cut segments must be maximum.**

i. **Recursive**

```
public int maximizeCuts(int n, int x, int y, int z)
   {
      int dp[] = new int[n+1];
      int ans = cutSegment(n, x, y, z, dp);
      return ans<0?0:ans;
   }
   public int cutSegment(int n, int x, int y, int z, int dp[]){
      if(n==0)
         return 0;
      if(n<0)
         return Integer.MIN_VALUE;
      if(dp[n]!=0)
         return dp[n];

      int first = 1 + cutSegment(n-x, x, y, z, dp);
      int second = 1 + cutSegment(n-y, x, y, z, dp);
      int third = 1 + cutSegment(n-z, x, y, z, dp);

      return dp[n] = Math.max(first,Math.max(second, third));
   }
```

ii. Iterative

```
int cutSegment(int n, int x, int y, int z)
{
        int dp[] = new int[n+1];
        Arrays.fill(dp, -1);

        dp[0] = 0;

        for(int i=0; i<n; i++)
        {
                if(dp[i]==-1)   // no way to reach it
                        continue;

                if(i+x<=n)
                        dp[i+x] = Math.max(dp[i+x], dp[i]+1);
                if(i+y<=n)
                        dp[i+y] = Math.max(dp[i+y], dp[i]+1);
                if(i+z<=n)
                        dp[i+z] = Math.max(dp[i+z], dp[i]+1);
        }
        return dp[n];
}
```

For Both
        T.C : O(n)
        S.C: O(n)

24) Ways To Tile A Floor

```
static long mod = 1000000007;
   static Long numberOfWays(int N) {
      long dp[] = new long[N+1];
      return fibonacci(N ,dp)%mod;
   }
   static long fibonacci(int n, long dp[]){
      if(n==1 || n==2)
         return n;
      if(dp[n]!=0)
        return dp[n];
      dp[n] = (fibonacci(n-1 ,dp)%mod+fibonacci(n-2, dp)%mod)%mod;
      return dp[n];
   }
```