Index - 2

Two Pointer
1. Three Sum Problem
2. Trapping Rain Water
3. Remove Duplicate from Sorted array
4. Max continuous number of 1’s
5. Max continuous number of 1’s if m 0’s are allowed to be flipped

Greedy Algorithm
6. N meeting in one room
7. Fractional Knapsack
8. Minimum Number of Platforms
9. Job Sequencing problem

BackTracking
10. N-Queens Problem
11. Solve the Sudoku (16th March 2021, 23:31)
12. M coloring Problem
13. Rat in a maze problem
14. Permutations of a given string
15. Word Break - 1 (Just Check)
16. Word Break - 2 (Print all the combinations)
17. Combination sum - 1
18. Combination sum - 2
19. Palindrome Partitioning
20. Subset Sum -1 (Print sum of all subset)
21. Print all non-duplicates subsets of the array
22. Kth Permutation Sequence
23. Generate All Valid Parentheses

SSP Id: 20210564982
0012318662

1) Three Sum Problem

i. Use 3 for loops. **t.c = n^3, s.c = 1**
ii. Using HashSet and It can be found. **t.c = n^2, s.c = n**
iii. Use Sorting and two pointer approach

```
public List<List<Integer>> threeSum(int[] nums) {
     List<List<Integer>> answer = new ArrayList<>();
     if(nums.length<3)
        return answer;

     Arrays.sort(nums);
     int i=0;
```

```
        while(i<nums.length-2){

            int left = i+1, right = nums.length-1;

            while(left<right){

                int sum = nums[left]+nums[right];

                if(-nums[i]==sum){
                    List<Integer> temp = new ArrayList<>();
                    temp.add(nums[i]);
                    temp.add(nums[left]);
                    temp.add(nums[right]);
                    answer.add(temp);
                    left++;
                    right--;
                    while(left<right && nums[left-1]==nums[left])
                        left++;
                    while(right>left && nums[right+1]==nums[right])
                        right--;
                }

                else if(sum<(-nums[i]))
                    left++;
                else
                    right--;
            }

            i++;
            while(i<nums.length-2 && nums[i-1]==nums[i])
                i++;
        }
        return answer;
    }
```

t.c = o(n^2), s.c = 1

2) Trapping Rain Water

i. Maintain two array finding the maximum on left and right side. Water stored is summation of a[i] - Math.max(right[i], left[i]). **t.c = n, s.c = n**
ii. Here in the above solution a pattern can be found so use pointers.

```
static int trappingWater(int a[], int n) {
        if(a.length<3)
            return 0;

        int leftMax = 0, rightMax = 0;
        int left = 0, right = a.length-1;
        int answer = 0;
```

```
        while(left<=right){
            if(a[left]<a[right]){
                if(a[left]>leftMax)
                    leftMax = a[left];
                answer = answer + leftMax - a[left];
                left++;
            }
            else{
                if(a[right]>rightMax)
                    rightMax = a[right];

                answer = answer + rightMax - a[right];
                right--;
            }
        }
        return answer;
    }
```

t.c = n, s.c = 1

3) Remove Duplicate from Sorted array

i. Using Extra space
ii. Use Two Pointer Approach

```
 public int removeDuplicates(int[] nums) {
        if(nums.length<2)
            return nums.length;

        int i=0;
        for(int j=1;j<nums.length ;j++)
            if(nums[i]!=nums[j])
                nums[++i] = nums[j];

        return i+1;
    }
```

T.c = n, s.c = 1

4) Max continuous number of 1's

i. Use two for loop and find it. **t.c = n^2, s.c = 1**
ii. Use Two pointer approach

```
  public int findMaxConsecutiveOnes(int[] nums) {
        if(nums.length<1)
            return 0;

        int i =0;
        while(i<nums.length && nums[i]!=1)
```

```
        i++;

    int j = i;
    int answer = 0;
    while(j<nums.length){
        while(j<nums.length && nums[j]==1)
            j++;
        answer = Math.max(answer ,j-i);
        while(j<nums.length && nums[j]==0)
            j++;
        i = j;
    }
    return answer;
}
```

t.c = n, s.c = 1

5) Max continuous number of 1's if m 0's are allowed to be flipped

i. Use two for loops and find number of 0's in every subarray. If they are less then or equal to m then ans = max(ans, subarray.length). **t.c = n^2, s.c = 1**

ii. Using an extra space of n
   i. First find the number of 0's in the array a
   ii. For every 0, find number of consecutive 1's on left and right of it and store them in 2 different arrays
   iii. Store the indices of the 0's in another array
   iv. From the zeroIndices array, for every consecutive m indices find the sum of 1's. And take the maximum of it and return the answer

T.C = n*m, S.C = n

iii. There is another efficient Approach

6) N meeting in one room

```
class Room{
    int st;
    int end;
    int pos;
    public Room(int st, int end, int pos){
        this.st = st;
        this.end = end;
        this.pos = pos;
    }
}

class meetingComparator implements Comparator<Room>{
    public int compare(Room o1,Room o2){
        if(o1.end<o2.end)
            return -1;
```

```
        else if(o1.end>o2.end)
            return 1;
        else if(o1.pos<o2.pos)
            return -1;

        return 1;
    }
}

class Meeting {

    public static int maxMeetings(int start[], int end[], int n) {
         ArrayList<Room> meet = new ArrayList<>();
         for(int i=0;i<n;i++)
           meet.add(new Room(start[i],end[i],i+1));

         meetingComparator mc = new meetingComparator();
         Collections.sort(meet, mc);

         int count = 1;
         int endTime = meet.get(0).end;
         for(int i=1;i<n;i++){
             if(meet.get(i).st>endTime){
                 count++;
                 endTime = meet.get(i).end;
             }
         }
         return count;
    }
```

t.c = nlogn , s.c = n

7) Fractional Knapsack

```
/*
class Item {
    int value, weight;
    Item(int x, int y){
        this.value = x;
        this.weight = y;
    }
}
*/

class AllItems{
    int value;
    int weight;
    double density;
    AllItems(int x ,int y, double z){
        value = x;
```

```
        weight = y;
        density = z;
    }
}
class knapsackComparator implements Comparator<AllItems>{
    public int compare(AllItems o1,AllItems o2){
        if(o1.density<o2.density)
            return 1;
        else if(o1.density>o2.density)
            return -1;
        else if(o1.weight<o2.weight)
            return -1;
        return 1;
    }
}
class Solution{
    double fractionalKnapsack(int w, Item a[], int n) {
        ArrayList<AllItems> array = new ArrayList<>();

        for(int i=0;i<n;i++){
            double x = (double)(a[i].value);
            double y = (double)(a[i].weight);
            array.add(new AllItems(a[i].value, a[i].weight,(double)(x/y)));
        }
        knapsackComparator kc = new knapsackComparator();
        Collections.sort(array, kc);

        double answer = findTheTotalValue(array, n, w);
        return answer;
    }
    double findTheTotalValue(ArrayList<AllItems> a, int n, int weight){
        double profit = 0;
        int i=0;
        while(weight>0 && i<n){
            if(weight>=a.get(i).weight){
                profit+=a.get(i).value;
                weight-=a.get(i).weight;
            }
            else{
                break;
            }

            i++;
        }
        if(i<n && weight>0){
            double x = (double)(weight*a.get(i).value);
            double y = (double)a.get(i).weight;
            double temp = (double)(x/y);
            profit+=temp;
        }
        return profit;
```

```
    }
}
```

t.c = nlogn, s.c = 1

8) Minimum Number of Platforms

```
static int findPlatform(int arr[], int dep[], int n)
   {
        Arrays.sort(arr);
        Arrays.sort(dep);
        int platform = 0, minPlatform = 0;

        int i=0,j=0;

        while(i<n && j<n){
            if(arr[i]<=dep[j]){
                platform++;
                i++;
            }
            else if(arr[i]>dep[j]){
                platform--;
                j++;
            }
            if(minPlatform<platform)
               minPlatform = platform;
        }
        return minPlatform;
   }
```

t.c = nlogn, s.c = 1

9) Job Sequencing problem

```
int[] JobScheduling(Job a[], int n){

        int jobs = 0, profit = 0;
        int maxTime = 0;
        for(int i=0;i<n;i++)
            maxTime = Math.max(maxTime, a[i].deadline);

        boolean timeArray[] = new boolean[maxTime];

         Arrays.sort(a, (x, y)-> y.profit-x.profit);
        for(int i=0;i<n;i++){

            for(int j = Math.min(maxTime-1,a[i].deadline-1);j>=0;j--){
                if(timeArray[j]==false){
                    timeArray[j] = true;
                    jobs++;
```

```
                profit+=a[i].profit;
                break;
            }
        }
    }

    return new int[]{jobs, profit};
}
```

t.c = n^2, s.c = Max(deadline)

10) N-Queens Problem

```
class Solution{
    static int[] a;
    static ArrayList<ArrayList<Integer>> answer;


    static ArrayList<ArrayList<Integer>> nQueen(int n) {
        a = new int[n+1];
        answer = new ArrayList<>();


        place(1,n);
        return answer;
    }
    static boolean isSafe(int row, int col){

        for(int i=1;i<row ;i++){
            if(a[i]==col)
                return false;
            else if(Math.abs(row-i)==Math.abs(a[i]-col))
                return false;
        }
        return true;
    }
    static void add(int n){

        ArrayList<Integer> temp = new ArrayList<>();
        for(int i=1;i<=n;i++)
            temp.add(a[i]);
        answer.add(temp);
    }
    static void place(int row, int n){

        for(int col=1; col<=n; col++){
            if(isSafe(row, col)){
               a[row] = col;
               if(row==n)
                   add(n);
```

```
            else
                place(row+1,n);
        }
    }
  }
}
```

t.c = n!, s.c = n

Time complexity :
O(N!). There is N possibilities to put the first queen, not more than N (N - 2) to put the second one, not more than N(N - 2)(N - 4) for the third one etc. In total that results in O(N!) time complexity.

Space complexity :O(N) to keep an information about diagonals and rows

11) Solve the Sudoku

```
 static boolean isSafe(int grid[][],int row, int col, int num){

        for(int i=0;i<grid.length ;i++)
            if(grid[i][col]==num)
                return false;

        for(int i=0;i<grid[0].length; i++)
            if(grid[row][i]==num)
                return false;

        int sqrt = (int)Math.sqrt(grid.length);

        int r = row - row%sqrt;
        int c = col - col%sqrt;

        for(int i=r;i<r+sqrt; i++){
            for(int j=c; j<c+sqrt; j++){
                if(grid[i][j]==num)
                    return false;
            }
        }
        return true;
    }
    static boolean SolveSudoku(int grid[][])
    {
        int row = -1;
        int col = -1;

        for(int i=0;i<grid.length; i++){
            for(int j=0;j<grid[0].length; j++){
                if(grid[i][j]==0){
                    row = i;
                    col = j;
```

```
                break;
            }
        }
    }

    if(row==-1 || col==-1)
        return true;

    for(int num=1; num<=grid.length; num++){
        if(isSafe(grid, row, col, num)){
            grid[row][col] = num;
            if(SolveSudoku(grid))
                return true;
            else
                grid[row][col] = 0;
        }
    }
    return false;
}
```

Time Complexity: O(9^N*N).
Auxiliary Space: O(N*N)

12) M coloring Problem

Approach:
Start from the first index and for every color from 1 to Number(n) if **i** can be used to color the cur index i.e if non of it's adjacent has same color if yes then use else move and do it recursively.
Graph is 0 - based indexed and is represented as array of list
List<Integer> graph[] = new ArrayList[V];
Arrays.fill(color, 0);

```
static boolean graphColoring(List<Integer> g[], int color[], int curVertex, int num)
{
        if(curVertex>=g.length)
                return true;

        for(int i=1; i<=num; i++)
        {
                if(isSafe(g, color, curVertex, i))
                {
                        color[curVertex] = i;

                        if(graphColoring(g, color, curVertex+1, num)
                                return true;
                        else
                                color[curVertex] = 0;
                }
        }
```

```
        return false;
}

static boolean isSafe(List<Integer> g[], int color[], int curVertex, int num)
{
        for(int i=0; i< g[curVertex].size(); i++)
                if(color[g[curVertex].get(i)]==num)
                        return false;

        return true;
}
```

Time Complexity: O(M^N).
Auxiliary Space: O(N)

13) Rat in a maze Problem

```
public static ArrayList<String> findPath(int[][] m, int n) {

      String s = "";
      ArrayList<String> answer = new ArrayList<>();
      traversePath(m, n,0 ,0,s, answer);

      Collections.sort(answer);
      return answer;
   }

   static void traversePath(int a[][],int n, int i ,int j,String s, ArrayList<String> answer){
      if(i<0 || i>=n || j<0 || j>=n)
         return;

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

      if(i==n-1 && j==n-1 && a[i][j]==1){
         answer.add(s);
         return;
      }

      a[i][j] = 0;

      traversePath(a, n, i+1 ,j, s+"D",answer);
      traversePath(a, n, i, j-1 ,s+"L",answer);
      traversePath(a, n, i ,j+1,s+"R",answer);
      traversePath(a, n ,i-1 ,j, s+"U",answer);

      a[i][j] = 1;
   }
```

Time Complexity: O(3^(n^2)).
As there are N^2 cells from each cell there are 3 unvisited neighbouring cells. So the time complexity O(3^(N^2).
Auxiliary Space: O(3^(n^2)).
As there can be atmost 3^(n^2) cells in the answer so the space complexity is O(3^(n^2)).

14) Permutations of a given string

Approach 1: Using HashMap

```
public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> answer = new ArrayList<>();
        List<Integer> ds = new ArrayList<>();
        boolean visited[] = new boolean[nums.length];

        recursion(nums, ds, answer, visited);
        return answer;
}
public void recursion(int nums[],List<Integer> ds, List<List<Integer>> answer, boolean
visited[]){

        if(ds.size()==nums.length){
            answer.add(new ArrayList<>(ds));
            return;
        }

        for(int i=0;i<nums.length ;i++){
            if(!visited[i]){
                visited[i] = true;
                ds.add(nums[i]);
                recursion(nums, ds, answer ,visited);
                ds.remove(ds.size()-1);
                visited[i] = false;
            }
        }
    }
```

Time Complexity: O(n*n!) Note that there are n! permutations and it requires O(n) time to print a a permutation
Space Complexity: O(n) + O(n)

Approach 2:

```
List<String> answer = new ArrayList<>();

    public List<String> find_permutation(String s) {
        allPermutations(s,0,s.length()-1);
        Collections.sort(answer);
        return answer;
    }
```

```
void allPermutations(String s ,int i, int r){
    if(l==r){
        if(!answer.contains(s))
            answer.add(s);
        return;
    }

    for(int i=l;i<=r;i++){
        s = swap(s, i, l);
        allPermutations(s, l+1,r);
        s = swap(s ,i, l);
    }
}

String swap(String s ,int i, int r){
    char c[] = s.toCharArray();
    char t = c[l];
    c[l] = c[r];
    c[r] = t;

    return String.valueOf(c);
}
```

Time Complexity: O(n*n!) Note that there are n! permutations and it requires O(n) time to print a a permutation
Space Complexity: O(1)

15) Word Break - 1

Approach:
1. BackTracking takes more time t.c = 2^n

```
 public static int wordBreak(String s, ArrayList<String> a)
    {
        HashSet<String> set = new HashSet<>();
        for(String str: a)
            set.add(str);

        if(segment(s, s.length(),set))
            return 1;
        return 0;
    }

    public static boolean segment(String s, int n,HashSet<String> set){
        if(n==0)
            return true;

        for(int i = 1;i<=n;i++){
            String temp = s.substring(0,i);
            if(set.contains(temp)){
```

```
                if(segment(s.substring(i,n),n-i, set))
                    return true;
            }
        }
        return false;
    }
```

2. Using DP
- Maintain a dp array in which dp[i] is true is the substring of s from i to n can be segmented into valid words.
- And a middle word can be valid only if it's next word is valid as we are traversing backwards

```
 public boolean wordBreak(String s, List<String> wordDict) {
        HashSet<String> dictionary = new HashSet<>();

        for(String str:wordDict)
            dictionary.add(str);

        int n = s.length();
        boolean dp[] = new boolean[s.length()+1];
        dp[n] = true;

        for(int i=n-1;i>=0;i--){
            for(int j=i; j<n ;j++){
                if(dictionary.contains(s.substring(i, j+1)) && dp[j+1]){
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[0];
    }
```

t.c = n^2
s.c = n

16) Word Break - 2 (Print all the combinations)

```
 static List<String> wordBreak(int n, List<String> dict, String s)
    {
        List<String> answer = new ArrayList<>();
        HashSet<String> set = new HashSet<>();
        for(int i=0;i<dict.size();i++)
            set.add(dict.get(i));

        partitionString(s, set, answer,"");
        return answer;
    }
```

```
    static void partitionString(String s,HashSet<String> set,List<String> answer,String
temp){
        if(s.length()==0){
            answer.add(temp.substring(0, temp.length()-1));
            return;
        }

        for(int i=1;i<=s.length();i++){
            if(set.contains(s.substring(0,i))){
                partitionString(s.substring(i, s.length()),set, answer, temp+s.substring(0,i)+" ");
            }
        }
    }
```

Time Complexity: O(n^n). Because there are n^n combinations in The Worst Case.
Auxiliary Space: O(n2). Because of the Recursive Stack of wordBreakUtil(…) function in The Worst Case.

17) Combination sum-1

1. Without Duplicates in the array

```
 public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> answer = new ArrayList<>();
        findCombination(candidates, 0, target, answer, new ArrayList<>());
        return answer;
    }

    public void findCombination(int a[],int index, int target,List<List<Integer>>
answer,ArrayList<Integer> ds){

        if(index==a.length){
            if(target==0){
                answer.add(new ArrayList<>(ds));
            }
            return;
        }

        if(a[index]<=target){
            ds.add(a[index]);
            findCombination(a, index, target-a[index],answer, ds);
            ds.remove(ds.size()-1);
        }
        findCombination(a, index+1, target, answer, ds);
    }
```

2. With Duplicates

Here Remove all the duplicates using HashSet

```
static ArrayList<ArrayList<Integer>> combinationSum(ArrayList<Integer> A, int B)
   {
       ArrayList<ArrayList<Integer>> answer = new ArrayList<>();
       HashSet<Integer> set = new HashSet<>(A);
       A.clear();
       A.addAll(set);
       findCombination(A, 0,B ,answer, new ArrayList<>());

       return answer;
   }


   static void findCombination(ArrayList<Integer> a, int index, int
target,ArrayList<ArrayList<Integer>> answer,ArrayList<Integer> ds){

       if(index==a.size()){
          if(target==0){
             answer.add(new ArrayList<>(ds));
          }
          return;
       }

       if(a.get(index)<=target){
          ds.add(a.get(index));
          findCombination(a, index, target-a.get(index), answer, ds);
          ds.remove(ds.size()-1);
       }
       findCombination(a, index+1, target, answer, ds);
   }
```

T.C = Exponential 2^e*k (k->MaxSize of data structure)
S.C = hypothetical (dependent on number of combinations) k*x(x->Combinations)

18) Combination sum-2

Approach 1:

```
public List<List<Integer>> combinationSum2(int[] candidates, int target) {
       List<List<Integer>> answer = new ArrayList<>();
       Arrays.sort(candidates);
       findCombinations(candidates, 0, target, answer, new ArrayList<>());
       return answer;
   }
   public void findCombinations(int a[],int index ,int target,List<List<Integer>>
answer,ArrayList<Integer> ds){
       if(target==0 && !answer.contains(ds)){
        answer.add(new ArrayList<>(ds));
          return;
       }
```

```
        if(index==a.length){
            return;
        }

        if(a[index]<=target){
            ds.add(a[index]);
            findCombinations(a, index+1, target-a[index], answer ,ds);
            ds.remove(ds.size()-1);
        }
        findCombinations(a, index+1, target, answer, ds);
    }
```

t.c = 2^n*k* logn (logn because we are using contains)
s.c = k*x

In LeetCode with Above solution the Runtime is 488ms
Below approach takes 2ms

Approach 2:

Now see striver's video if you don't understand the code.

Idea is that, if you have taken x as ith element ,jth then don't take same x as ith element if it is present in (j+1)th index

```
public List<List<Integer>> combinationSum2(int[] candidates, int target) {
        List<List<Integer>> answer = new ArrayList<>();
        Arrays.sort(candidates);
        findCombinations(candidates, 0, target, answer, new ArrayList<>());
        return answer;
    }


    public void findCombinations(int a[],int index, int target,List<List<Integer>> 
answer,ArrayList<Integer> ds){
        if(target==0){
             answer.add(new ArrayList<>(ds));
            return;
        }

        for(int i=index ;i<a.length ;i++){
            if(i>index && a[i]==a[i-1])
                continue;
            if(a[i]>target)
                break;

            ds.add(a[i]);
            findCombinations(a, i+1, target-a[i], answer, ds);
            ds.remove(ds.size()-1);
```

```
        }

    }
```

t.c = Same as above without the logn Factor
s.c = Same

19) Palindrome Partitioning

Separate the given strings into palindrome segments and prints all combination

```
 public List<List<String>> partition(String s) {
        List<List<String>> answer = new ArrayList<>();
        List<String> ds = new ArrayList<>();

        function(s, 0, answer ,ds);
        return answer;
    }
    public void function(String s ,int index,List<List<String>> answer,List<String> ds)
    {
        if(index==s.length()){
            answer.add(new ArrayList<>(ds));
            return;
         }

        for(int i=index ;i<s.length();i++){
            if(sPalindrome(s, index, i)){
                ds.add(s.substring(index ,i+1));
                function(s, i+1, answer, ds );
                ds.remove(ds.size()-1);
            }
        }
    }
    public boolean isPalindrome(String s ,int start, int end]){

        int x=start ,y=end;
        while(start<end){
            if(s.charAt(start++)!=s.charAt(end--))
                return false;
        }
        return true;
    }
```

t.c = 2^n * n
s.c = n

20) Subset Sum -1 (Print sum of all subset)

```
 ArrayList<Integer> subsetSums(ArrayList<Integer> arr, int N){
        ArrayList<Integer> answer = new ArrayList<>();
```

```
        function(arr ,0, 0 ,answer);
        Collections.sort(answer);
        return answer;
    }
    void function(ArrayList<Integer> a, int ind, int sum,ArrayList<Integer> answer){
        if(ind==a.size()){
            answer.add(sum);
            return;
        }
        //Pick current element
        function(a, ind+1, sum+a.get(ind),answer);

        //Don't pick current element
        function(a, ind+1, sum, answer);
    }
```

t.c = 2^n
s.c = 2^n

21) Print all non-duplicates subsets of the array

```
   public List<List<Integer>> subsetsWithDup(int[] nums) {
        Arrays.sort(nums);
        List<List<Integer>> answer = new ArrayList<>();
        addSubset(nums, 0, answer, new ArrayList<>());
        return answer;
    }
    public void addSubset(int a[],int ind, List<List<Integer>> answer,ArrayList<Integer> ds){
        answer.add(new ArrayList<>(ds));

        for(int i=ind; i<a.length ;i++){
            if(i>ind && a[i]==a[i-1])
                continue;
            ds.add(a[i]);
            addSubset(a ,i+1, answer, ds);
            ds.remove(ds.size()-1);
        }
    }
```

t.c = 2^n
s.c = 2^n

22) Kth Permutation Sequence

1. Approach:
Find out all the possible permutation and then store it in some data structure. Sort the data structure and finally return the kth sequence

t.c = n! * n + n! log(n!) (to sort)
s.c = n!

2. Approach

Given n = 4 and k = 17 (24 possible permutations)

Answer is 3 4 1 2

Process:

There are certain sequences which starts with 1, 2, 3, 4.

Sequence starting with 1

1 choose(2, 3, 4) - 6 combinations

Similarly for sequences starting with 2, 3, 4 there will be 6 combinations each.
Making 24 combinations in total
Notice that number of combinations starting with 1, 2, 3, 4 are (n-1)! each.

So if we index the sequences it will look as follows:

Starting with:	Index
1	0 - 5
2	6 - 11
3	12 - 17
4	18 - 23

Now I want the 17th combination I.e combination at 16th in zero base indexing.

Array : 1 2 3 4
Index: 0 1 2 3

16/(n-1)! Gives the first index.
16/6 = 2. (element at 2nd index i.e 3)

I got the starting number so remove it from array.

Array : 1 2 4
Index: 0 1 2

Repeat the same steps to get final answer.
N = 3,
Number of combinations with different starting numbers is (n-1)! = 2.

1 - (0-1)
2 - (2-3)
4 - (4-5)

I want 16%6 = 4th sequence after 12
I.e k = 4
Starting with 4

4/2 = 2(index)
Remove 4

Array : 1 2
Index: 0 1

4%2 = 0. Next k

i.e
1

And finally the remaining element

Answer = 3 4 1 2

```
String kThPermutation(int n, int k)
{
        StringBuffer ans = new StringBuffer();
        List<Integer> numbers = new ArrayList<>();
        int fact = 1;

        for(int i=1; i<n; i++)
        {
                fact = fact*i;
                numbers.add(i);
        }
        numbers.add(n);
        k = k-1;

        while(true)
        {
                answer.append(numbers.get(k/fact);
                numbers.remove(k/fact);

                if(numbers.size()==0)
                        break;
                k = k%fact;
                fact = fact/numbers.size();
        }

        return answer.toString();
}
```

t.c =. (N^2)
s.c = n

24) Generate All Valid Parentheses

```
public List<String> AllParenthesis(int n)
   {
      List<String> answer = new ArrayList<>();
      validParenthesis(n, n,"",answer);
      return answer;
```

```
    }
    public void validParenthesis(int open, int close,String str, List<String> answer){
        if(open==0 && close==0){
            answer.add(str);
            return;
        }

        if(open!=0){
            validParenthesis(open-1, close, str+"(",answer);
        }
        if(close>open){
            validParenthesis(open, close-1, str+")",answer);
        }
    }
```

T.C = (2N * N).
S.C = (2*N*X), X = Number of valid Parenthesis.