Advanced Recursion

Introduction

As we have already discussed what recursion is and how to implement it. In this module, we are going to cover some other important concepts and problems based on the recursion. Recursion is a very powerful tool and it makes our code readable and short. But its applications are very vast therefore we will solve some more problems so that we can come in handy.

Generate all the subsets

Problem statement: You are given an array **arr** of **N** distinct integers. Your task is to find all the non-empty subsets of the array.

Approach:

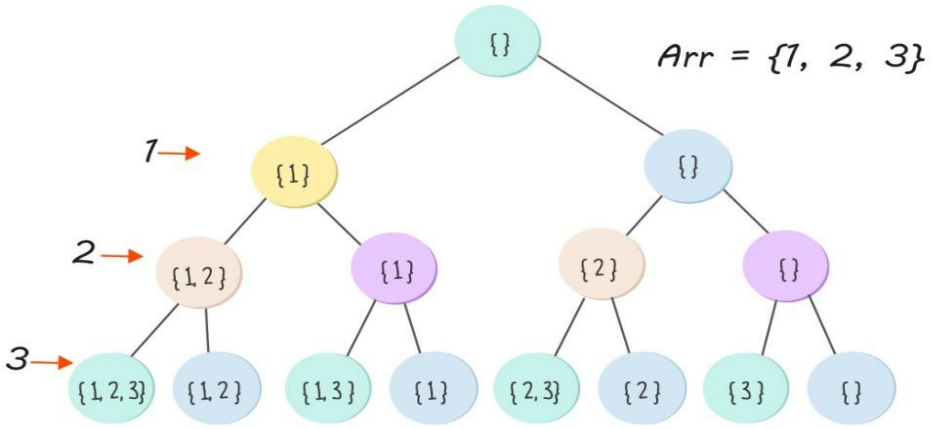
Each element has 2 choices, it can either be included or excluded.

Using recursion we can create all the possible subsets, we can add the current element to the temporary array, and make a recursive call for the next element, when the function call returns we remove the current element from the temporary array and again make a recursive call for the next element.

Each time we reach the end of the array, we will add the temporary array to the answer.

Algorithm:

- Initialize a 2D list **ans** to store all the subsets
- Initialize a arraylist **temp** to store the current subset
- Create a recursive function SubsetFinder which takes the following parameters: **index** to store current index, **temp** store current subset, **ans** 2D list to store all subsets, **arr** arraylist that is the input.
- The initial call is made with **i** equal to **0**, empty temp vector, empty 2D list **ans**, given **arr** input arraylist.
- In **SubsetFinder**, we add the **index** to the **temp** arraylist and recursively call SubsetFinder with **index+1.**
- When a function call is returned, we remove the **index** from temp, and again recursively call **SubsetFinder** with **index+1**
- If the value of **index** is equal to **size** of the input array, add temp to **ans**
- Finally, return **ans**



Code:

```
Public static void Subsets(ArrayList<Integer> nums, int index,
ArrayList<Integer> temp, List<List<Integer>> ans)
    {
        // Base Condition
        if (index == nums.size()) {
            if(temp.size()>0) ans.add(temp);
            return;
        }

        // Not Including Value which is at Index
        Subsets(nums,index+1,new ArrayList<>(temp),ans);

        // Including Value which is at Index
        temp.add(nums.get(index));
        Subsets(nums,index+1,new ArrayList<>(temp),ans);
    }

 public static void main(String[] args){
        Scanner s = new Scanner(System.in);
        int n = s.nextInt();
        List<List<Integer>> ans = new ArrayList<>();
        ArrayList<Integer> arr = new ArrayList<>();
        for(int i=0;i<n;i++){
            arr.add(s.nextInt());
        }

        Subsets(arr,0,new ArrayList<>(), ans);
        Collections.sort(ans, (o1, o2) -> {
            int m = Math.min(o1.size(), o2.size());
            for (int i = 0; i < m; i++) {
                if (o1.get(i) == o2.get(i)){
                    continue;
                }else{
                    return o1.get(i) - o2.get(i);
```

```
                }
            }
            return 1;
        });
        System.out.println();
        for(int i = 0; i < ans.size(); i++){
          for(int j = 0; j < ans.get(i).size(); j++){
              System.out.print(ans.get(i).get(j) + " ");
          }
          System.out.println();
        }
    }
}
```

Time Complexity: O(2^N)

Each element of the array has 2 options therefore overall **2^N** for **N** elements, Hence the time complexity is **O(2^N).**

Space Complexity: O(N * 2^N), where N is the size of the input array.

Since a total of **$2^{(N-1)}$** subsets are generated, and the maximum possible length of each subset is of order N . Hence the space complexity is **O(N * 2^N)**.

Combination Sum

Problem statement: Your task is to find all unique combinations in the array whose sum is equal to **B**. A number can be chosen any number of times from array/list **ARR.**

Algorithm:

- We will sort the array/list **nums** in ascending order.
- Let the current sum of the combination on which we are recursing over be **target** and the index which we are on in the array/list **nums** be **ind**.
- In each recursive call, we will:
- While currSum <= B:
- Increase **target** by **nums[ind]**.
- Insert **nums[ind]** in **temp**.
- Recurse over **Ind+ 1**.
- We remove all **num[ind]** inserted into **temp** for recursion.
- The state where the **target** is equal to B will be a valid combination and we will include it in our final answer.

Code:

```
public static void combinationSum(int[] nums, int ind, int n,int target, ArrayList<Integer> temp){

    if(target==0)  // combination sum is found
    {
        System.out.println(temp);
        return;
     }

    if(ind>=n || target-nums[ind]<0)     // no combination sum found
        return;

     // recursive call
     //pick the ith StackTraceElement
     temp.add(nums[ind]);
     combinationSum(nums,ind,n,target-nums[ind],temp);
     temp.remove(temp.size()-1);
     combinationSum(nums,ind+1,n,target,temp);
  }
```

```
    public static void main(String[] args){
      Scanner sc= new Scanner(System.in);

      int n = sc.nextInt();
      int target = sc.nextInt();

      int[] nums = new int[n];
      for(int i=0;i<n;i++)
        nums[i] = sc.nextInt();

      Arrays.sort(nums);

      ArrayList<Integer> temp = new ArrayList<>();

      combinationSum(nums,0,n,target,temp);

    }
}
```

Time Complexity: O(2^N), where **N** denotes the number of elements of the array/list.

Since we are recursing over all possible combinations of the array, the time complexity will be **O(2^N)**.

Space Complexity: O(N*2^N), where **N** denotes the number of elements of the array/list.

The space complexity due to the recursion stack will be **O(N)**. For each possible state, we store all possible elements in **temp**. Hence, the overall space complexity will be **O(N*N)**.

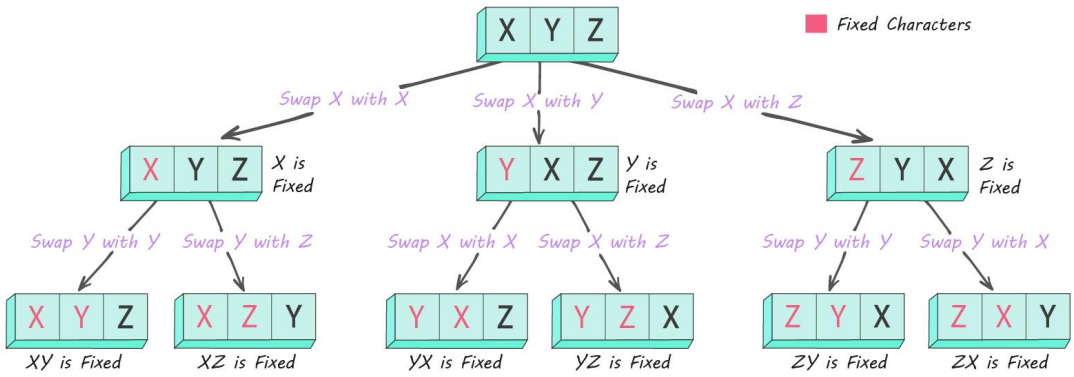
Permutations of the string:

Problem statement:

You are given a string **Str** consisting of lowercase English letters. Your task is to return all permutations of the given string in lexicographically increasing order.

Algorithm:

- Let's define a function **Permutaions( )**. This function generates the permutations of the substring.
- Call the function: **Permutaions()**.
- If **index** is equal to the string length, we have found a permutation. Push the string **Str** in the **ans** list.
- Else, iterate on the indices from **i** to **n**.
- Let **i** denote the current index.
- Fix **ith** character on the index , to do this swap **Str[i]** and **Str[index]**.
- Call the function for the rest of the characters: **Permutaions(Str, index + 1, ans).**
- Backtrack and swap the character **Str[index]** and **Str[i]** again.
- Now, we have the list **ans** which contains all the permutations of the given string. To order this in lexicographically increasing order, we will sort the list.



Recursion Tree for Permutation of String "XYZ"

Code:

```
public static String swap(String a, int i, int j){
        char temp;
        char[] charArray = a.toCharArray();
        temp = charArray[i] ;
        charArray[i] = charArray[j];
        charArray[j] = temp;
        return String.valueOf(charArray);
    }

public static void Permutations(String str,int index,ArrayList<String>
ans){
        // Base condition
        if(index == str.length()){
            ans.add(str);
            return;
        }

        // recursive calls ?
        int n = str.length();
        for(int i=index;i<n;i++){
            str = swap(str,i,index);
            Permutations(str,index+1,ans);
```

```
            str = swap(str,i,index);
        }
    }

    public static void main(String[] args){
        Scanner s = new Scanner(System.in);
        String str = s.nextLine();
        ArrayList<String> ans = new ArrayList<>();
        Permutations(str,0,ans);
        Collections.sort(ans);
        for(int i=0;i<ans.size();i++){
            System.out.println(ans.get(i));
        }
    }
}
```

Time Complexity: O(N! * log(N!)), Where **N** is the length of the given string.

There are **N!** Permutations of a string of length **N**. In each recursive call, we are traversing the string which will take **O(N)** time in the worst case. Thus, generating all permutations of a string takes **O(N * N!)** time.

We are also sorting the **ans** list of size **O(N!)** which will take **O(N! * log(N!))** time.

Thus, the final time complexity is **O(N! * log(N!) + N * N!)** ~ **O(N! * log(N!))**

Space Complexity: O(N * N!), Where **N** is the length of the given string.

O(N) recursion stack is used by the recursive function, we are also storing the permutations in a list which will **O(N * N!)** space. Thus, the final space complexity is **O(N + N * N!) ~ O(N * N!)**.