Worksheet - 9

Ques 1 :
https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/description/
Code:

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
    int ans[]= {-1,-1};
    ans[0]= search(nums, target, true);
    ans[1]= search(nums, target, false);

    return ans;
  }

  public int search(int[] nums, int target, boolean isStartIndex){ // binary search

    int start=0;
    int end= nums.length - 1;
    int ans= -1;

    while(start <= end){
      int mid = start + (end - start ) / 2;
      if(target > nums[mid]){
        start= mid+1;
      }else if(target < nums[mid]){
        end= mid-1;
      }else{
        ans= mid;
        if(isStartIndex){ //search in first half for potential ans
          end= mid-1;
        }else{  // search in second half for potential ans
          start= mid+1;
        }
      }
    }
    return ans;
  }
```

}

Explanation:

This solution uses binary search to find the starting and ending positions of a target element in a sorted array. The method searchRange initiates two searches using the search function. The first search looks for the starting index of the target by scanning the left half of the array, and the second search looks for the ending index by scanning the right half. The search function applies binary search, adjusting the start and end pointers based on the comparison between the target and the middle element. If the target is found, the index is stored, and further searches are done to check for other potential occurrences in the specified half. The time complexity is O(log N) since it uses binary search, and the space complexity is O(1) since no extra space is used apart from the output array.

Input : nums=[1,2,4,5,8,8,10]
Output:[4,5]

Ques 2: https://leetcode.com/problems/search-in-rotated-sorted-array/

Code:

```
class Solution {
    public int search(int[] nums, int target) {
        for(int i =0;i<nums.length;i++){
            if(nums[i]==target){
                return i;
            }
        }
        return -1;
    }
}
```

Explanation:

The solution of search in a rotated sorted array , first take a for loop from i=0 to i= nums.length. In the loop, the condition for checking which nums[i] is equal to target or not. If target is matched then return the index of nums array otherwise return -1.

Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4

Ques 3: https://leetcode.com/problems/next-permutation/description/

Code:

```
class Solution {
    public void nextPermutation(int[] nums) {
```

```
        int i = nums.length-2;
        while(i>=0 && nums[i] >= nums[i+1]){
            i--;
        }
        if(i>=0){
            int last = nums.length-1;
            while(last>=0 && nums[i] >= nums[last]){
                last--;
            }
            swap(nums,last,i);
        }
        reverse(nums,i+1);
    }
    public void swap(int[] nums,int i,int j){
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }

    public void reverse(int[] nums,int start){
        int end = nums.length-1;
        while(start < end){
            swap(nums,start,end);
            start++;
            end--;
        }
    }
}
```

Explanation:

This code implements the algorithm to find the next lexicographical permutation of a given array of integers. The method nextPermutation modifies the array in-place to transform it into the next permutation that is just larger than the current one. The algorithm first identifies the largest index i where nums[i] < nums[i+1]. Then, it finds the largest index last where nums[last] > nums[i] and swaps these two elements. After that, it reverses the order of the elements following index i to make them the smallest possible order. If no such i exists, the array is reversed to give the smallest permutation.

Input: nums=[1 2 3]

Output:[1 3 2]

Ques 4: https://leetcode.com/problems/search-insert-position/

Code:

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        int i = 0, j = nums.length;
        while (i < j) {
            int mid = (i + j) >>> 1;
            if (nums[mid] >= target) {
                j = mid;
            } else {
                i = mid + 1;
            }
        }
        return i;
    }
}
```

Explanation :

This code implements a binary search algorithm to find the position where a target value should be inserted in a sorted array (`nums`). The method `searchInsert` searches for the target and returns its index if found, or the index where it should be inserted to maintain the array's sorted order. It initializes two pointers, `i` and `j`, representing the search range. The midpoint `mid` is calculated, and based on the comparison between `nums[mid]` and the target, the search range is adjusted. The loop continues until the search range narrows down, and the final insertion index `i` is returned.

Input : nums= [1] target = 5

Output : 1

Ques 5:

https://leetcode.com/problems/substring-with-concatenation-of-all-words/

Code :

```
import java.util.*;
class Solution {
    public List<Integer> findSubstring(String s, String[] words) {
        List<Integer> result = new ArrayList<>();
        if (s == null || s.length() == 0 || words == null || words.length == 0) {
            return result;
        }
```

```
        int wordLength = words[0].length();
        int wordCount = words.length;
        int totalLength = wordLength * wordCount;
        Map<String, Integer> wordMap = new HashMap<>();
        for (String word : words) {
            wordMap.put(word, wordMap.getOrDefault(word, 0) + 1);
        }
        for (int i = 0; i < wordLength; i++) {
            int left = i, right = i, count = 0;
            Map<String, Integer> windowMap = new HashMap<>();
            while (right + wordLength <= s.length()) {
                String word = s.substring(right, right + wordLength);
                right += wordLength;
                if (wordMap.containsKey(word)) {
                    windowMap.put(word, windowMap.getOrDefault(word, 0) + 1);
                    count++;
                    while (windowMap.get(word) > wordMap.get(word)) {
                        String leftWord = s.substring(left, left + wordLength);
                        windowMap.put(leftWord, windowMap.get(leftWord) - 1);
                        left += wordLength;
                        count--;
                    }
                    if (count == wordCount) {
                        result.add(left);
                    }
                } else {
                    windowMap.clear();
                    count = 0;
                    left = right;
                }
            }
        }
        return result;
    }
}
```

Explanation :

This code finds all starting indices in the string `s` where a concatenation of all words in the array `words` (in any order) occurs as a substring. It uses a sliding window technique

to efficiently check each possible starting point. The function first calculates the total length of the concatenated words and creates a frequency map of the words. It then loops through `s` starting at different positions (based on word length). For each position, it slides the window, checking if the current substring matches any word in the `words` array. If all words are matched within the window, the starting index is added to the result list. If an invalid word is found or too many words appear, the window is adjusted accordingly.

Input: s = "barfoothefoobarman", words = ["foo","bar"]
Output: [0,9]

Ques 6: https://leetcode.com/problems/basic-calculator/
Code :

```
class Solution {
    public int calculate(String s) {
        Stack<Integer> stack = new Stack<>();
        int num = 0;
        int sign = 1; // 1 for positive, -1 for negative
        int result = 0;
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (Character.isDigit(c)) {
                num = num * 10 + (c - '0'); // Build the number
            } else if (c == '+') {
                result += sign * num; // Add the last number to result
                sign = 1; // Set sign to positive for next number
                num = 0; // Reset num to start building the next number
            } else if (c == '-') {
                result += sign * num; // Add the last number to result
                sign = -1; // Set sign to negative for next number
                num = 0; // Reset num to start building the next number
            } else if (c == '(') {
                stack.push(result); // Push result to stack
                stack.push(sign); // Push sign to stack
                result = 0; // Reset result for the new sub-expression
                sign = 1; // Default sign is positive
            } else if (c == ')') {
                result += sign * num; // Add the last number to result
                result *= stack.pop(); // Multiply by the sign before the parentheses
```

```
            result += stack.pop(); // Add the result before the parentheses
            num = 0; // Reset num for next calculation
        }
      }
      result += sign * num; // Add any remaining number to the result
      return result;
   }
}
```

Explanation :

This code evaluates a mathematical expression in string form that contains integers, addition (`+`), subtraction (`-`), and parentheses. It uses a stack to handle the parentheses and manage intermediate results. As it iterates through the string, it builds numbers and applies the appropriate signs (`+` or `-`). When encountering an opening parenthesis `(`, it pushes the current result and sign onto the stack and resets them for the sub-expression. Upon encountering a closing parenthesis `)`, it completes the sub-expression, multiplies by the sign before the parentheses, and adds the result back to what was stored on the stack. The final result is returned after processing the entire string.

Input: s = "1 + 1"

Output: 2