1. Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

You must write an algorithm with O(log n) runtime complexity.

Sol:

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        int l = search(nums, target);
        int r = search(nums, target + 1);
        return l == r ? new int[] {-1, -1} : new int[] {l, r - 1};
    }

    private int search(int[] nums, int x) {
        int left = 0, right = nums.length;
        while (left < right) {
            int mid = (left + right) >>> 1;
            if (nums[mid] >= x) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }
}
```

2. There is an integer array nums sorted in ascending order (with distinct values).

Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums after the possible rotation and an integer target, return *the index of* target *if it is in* nums*, or* -1 *if it is not in* nums.

You must write an algorithm with O(log n) runtime complexity.

Sol:

```
class Solution {
    public int search(int[] nums, int target) {
        int n = nums.length;
        int left = 0, right = n - 1;
        while (left < right) {
            int mid = (left + right) >> 1;
            if (nums[0] <= nums[mid]) {
                if (nums[0] <= target && target <= nums[mid]) {
                    right = mid;
                } else {
                    left = mid + 1;
                }
            } else {
                if (nums[mid] < target && target <= nums[n - 1]) {
                    left = mid + 1;
                } else {
                    right = mid;
                }
            }
        }
        return nums[left] == target ? left : -1;
    }
}
```

3. A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

- **For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].**

The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- **For example, the next permutation of arr = [1,2,3] is [1,3,2].**
- **Similarly, the next permutation of arr = [2,3,1] is [3,1,2].**
- **While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.**

Given an array of integers nums, *find the next permutation of* nums.

The replacement must be in place and use only constant extra memory.

Sol: class Solution {

```
  public void nextPermutation(int[] nums) {
    int n = nums.length;
    int i = n - 2;
    for (; i >= 0; --i) {
      if (nums[i] < nums[i + 1]) {
        break;
      }
    }
    if (i >= 0) {
      for (int j = n - 1; j > i; --j) {
        if (nums[j] > nums[i]) {
          swap(nums, i, j);
          break;
        }
      }
    }
```

```
    for (int j = i + 1, k = n - 1; j < k; ++j, --k) {
      swap(nums, j, k);
    }
  }

  private void swap(int[] nums, int i, int j) {
    int t = nums[j];
    nums[j] = nums[i];
    nums[i] = t;
  }
}
```

4. Given a string s representing a valid expression, implement a basic calculator to evaluate it, and return *the result of the evaluation*.

Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().

Sol:

```
class Solution {
  public int calculate(String s) {
    Deque<Integer> stk = new ArrayDeque<>();
    int sign = 1;
    int ans = 0;
    int n = s.length();
    for (int i = 0; i < n; ++i) {
      char c = s.charAt(i);
      if (Character.isDigit(c)) {
        int j = i;
        int x = 0;
        while (j < n && Character.isDigit(s.charAt(j))) {
          x = x * 10 + s.charAt(j) - '0';
```

```
            j++;
          }
          ans += sign * x;
          i = j - 1;
        } else if (c == '+') {
          sign = 1;
        } else if (c == '-') {
          sign = -1;
        } else if (c == '(') {
          stk.push(ans);
          stk.push(sign);
          ans = 0;
          sign = 1;
        } else if (c == ')') {
          ans = stk.pop() * ans + stk.pop();
        }
      }
      return ans;
    }
}
```

5. Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

Sol:

```
class Solution {
  public int calculate(String s) {
    Deque<Integer> stk = new ArrayDeque<>();
    int sign = 1;
    int ans = 0;
    int n = s.length();
    for (int i = 0; i < n; ++i) {
      char c = s.charAt(i);
      if (Character.isDigit(c)) {
        int j = i;
        int x = 0;
        while (j < n && Character.isDigit(s.charAt(j))) {
          x = x * 10 + s.charAt(j) - '0';
          j++;
        }
        ans += sign * x;
        i = j - 1;
      } else if (c == '+') {
        sign = 1;
      } else if (c == '-') {
        sign = -1;
      } else if (c == '(') {
        stk.push(ans);
        stk.push(sign);
        ans = 0;
        sign = 1;
      } else if (c == ')') {
```

```
            ans = stk.pop() * ans + stk.pop();
          }
        }
        return ans;
    }
}
```

6. You are given a string s and an array of strings words. All the strings of words are of the same length.

A concatenated string is a string that exactly contains all the strings of any permutation of words concatenated.

- **For example, if words = ["ab","cd","ef"],**
  then "abcdef", "abefcd", "cdabef", "cdefab", "efabcd", and "efcdab" are all concatenated strings. "acdbef" is not a concatenated string because it is not the concatenation of any permutation of words.

Return an array of *the starting indices* of all the concatenated substrings in s. You can return the answer in any order.

Sol:

```
class Solution {
    public List<Integer> findSubstring(String s, String[] words) {
        Map<String, Integer> cnt = new HashMap<>();
        for (String w : words) {
            cnt.merge(w, 1, Integer::sum);
        }
        int m = s.length(), n = words.length;
        int k = words[0].length();
        List<Integer> ans = new ArrayList<>();
        for (int i = 0; i < k; ++i) {
            Map<String, Integer> cnt1 = new HashMap<>();
            int l = i, r = i;
            int t = 0;
            while (r + k <= m) {
                String w = s.substring(r, r + k);
```

```
            r += k;
            if (!cnt.containsKey(w)) {
              cnt1.clear();
              l = r;
              t = 0;
              continue;
            }
            cnt1.merge(w, 1, Integer::sum);
            ++t;
            while (cnt1.get(w) > cnt.get(w)) {
              String remove = s.substring(l, l + k);
              l += k;
              cnt1.merge(remove, -1, Integer::sum);
              --t;
            }
            if (t == n) {
              ans.add(l);
            }
          }
        }
        return ans;
    }
}
```