Coding Practice – 9

Two pointers:

1. **Valid Palindrome**
   A phrase is a **palindrome** if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward.
   Alphanumeric characters include letters and numbers.
   Given a string s, return true *if it is a **palindrome**, or* false *otherwise*.

   Example 1:
   Input: s = "A man, a plan, a canal: Panama"
   Output: true
   Explanation: "amanaplanacanalpanama" is a palindrome.
   Example 2:
   Input: s = "race a car"
   Output: false
   Explanation: "raceacar" is not a palindrome.
   Example 3:
   Input: s = " "
   Output: true
   Explanation: s is an empty string "" after removing non-alphanumeric characters.
   Since an empty string reads the same forward and backward, it is a palindrome.

   Constraints:

- $1 <= s.length <= 2 * 10^5$
- s consists only of printable ASCII characters.

   Code:

```
class Solution {
   public static boolean isPalindrome(String s) {
      int left = 0, right = s.length() - 1;
      while (left < right) {
         while (left < right && !Character.isLetterOrDigit(s.charAt(left))) left++;
         while (left < right && !Character.isLetterOrDigit(s.charAt(right))) right--;
         if (Character.toLowerCase(s.charAt(left)) != Character.toLowerCase(s.charAt(right)))
return false;
         left++;
         right--;
      }
      return true;
   }

   public static void main(String[] args) {
      System.out.println(isPalindrome("A man, a plan, a canal: Panama"));
      System.out.println(isPalindrome("race a car"));
      System.out.println(isPalindrome(" "));
   }
}
```

Output:

```
true
false
true
```

Time Complexity: O(n)

2. **Is Subsequence**

   Given two strings s and t, return true *if* s *is a* ***subsequence*** *of* t*, or* false *otherwise*.

   A **subsequence** of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not).

   Example 1:
   Input: s = "abc", t = "ahbgdc"
   Output: true
   Example 2:
   Input: s = "axc", t = "ahbgdc"
   Output: false

   Constraints:

- 0 <= s.length <= 100
- 0 <= t.length <= 10^4
- s and t consist only of lowercase English letters.

Code:

```
class Solution {
    public boolean isSubsequence(String s, String t) {
        int i = 0, j = 0;
        while (i < s.length() && j < t.length()) {
            if (s.charAt(i) == t.charAt(j)) i++;
            j++;
        }
        return i == s.length();
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println(sol.isSubsequence("abc", "ahbgdc"));
        System.out.println(sol.isSubsequence("axc", "ahbgdc"));
    }
}
```

Output:

```
true
false
```

Time Complexity: O(m+n)

3. **Two Sum II – Input Array is sorted**

Given a **1-indexed** array of integers numbers that is already ***sorted in non-decreasing order***, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[$index_1$] and numbers[$index_2$] where 1 <= $index_1$ < $index_2$ <= numbers.length.
Return *the indices of the two numbers,* $index_1$ *and* $index_2$*,* ***added by one*** *as an integer array* [$index_1$, $index_2$] *of length 2.*
The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.
Your solution must use only constant extra space.

Example 1:
Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore, $index_1$ = 1, $index_2$ = 2. We return [1, 2].
Example 2:
Input: numbers = [2,3,4], target = 6
Output: [1,3]
Explanation: The sum of 2 and 4 is 6. Therefore $index_1$ = 1, $index_2$ = 3. We return [1, 3].
Example 3:
Input: numbers = [-1,0], target = -1
Output: [1,2]
Explanation: The sum of -1 and 0 is -1. Therefore $index_1$ = 1, $index_2$ = 2. We return [1, 2].

Constraints:
- 2 <= numbers.length <= 3 * 10^4
- -1000 <= numbers[i] <= 1000
- numbers is sorted in **non-decreasing order**.
- -1000 <= target <= 1000
- The tests are generated such that there is **exactly one solution**.

Code:
```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int left = 0, right = numbers.length - 1;
        while (left < right) {
            int sum = numbers[left] + numbers[right];
            if (sum == target) return new int[]{left + 1, right + 1};
            if (sum < target) left++;
            else right--;
        }
        return new int[]{};
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        int[] result1 = sol.twoSum(new int[]{2, 7, 11, 15}, 9);
        System.out.println(java.util.Arrays.toString(result1));
        int[] result2 = sol.twoSum(new int[]{2, 3, 4}, 6);
        System.out.println(java.util.Arrays.toString(result2));
        int[] result3 = sol.twoSum(new int[]{-1, 0}, -1);
```

```
        System.out.println(java.util.Arrays.toString(result3));
    }
}
```

Output:

```
[1, 2]
[1, 3]
[1, 2]
```

Time Complexity: O(n)

4. **Container with most water**

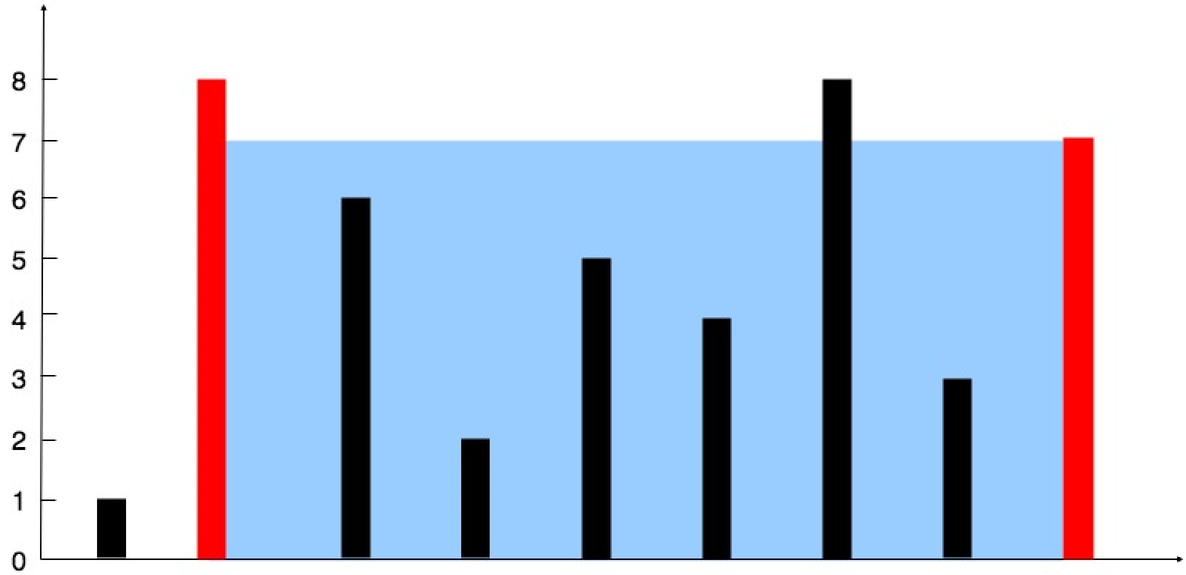
You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the i^{th} line are (i, 0) and (i, height[i]).

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return *the maximum amount of water a container can store*.

Notice that you may not slant the container.

Example 1:



Input: height = [1,8,6,2,5,4,8,3,7]

Output: 49

Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

Example 2:

Input: height = [1,1]

Output: 1

Constraints:

- n == height.length
- 2 <= n <= 10^5
- 0 <= height[i] <= 10^4

Code:

```
class Solution {
   public int maxArea(int[] height) {
      int left = 0, right = height.length - 1, maxArea = 0;
      while (left < right) {
         int area = Math.min(height[left], height[right]) * (right - left);
         maxArea = Math.max(maxArea, area);
         if (height[left] < height[right]) left++;
         else right--;
      }
      return maxArea;
   }
```

```
    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println(sol.maxArea(new int[]{1, 8, 6, 2, 5, 4, 8, 3, 7}));
        System.out.println(sol.maxArea(new int[]{1, 1}));
    }
}
```

Output:

```
49
1
```

Time Complexity: O(n)

5. **3Sum**
   Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.
   Notice that the solution set must not contain duplicate triplets.

   Example 1:
   Input: nums = [-1,0,1,2,-1,-4]
   Output: [[-1,-1,2],[-1,0,1]]
   Explanation:
   nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
   nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
   nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
   The distinct triplets are [-1,0,1] and [-1,-1,2].
   Notice that the order of the output and the order of the triplets does not matter.
   Example 2:
   Input: nums = [0,1,1]
   Output: []
   Explanation: The only possible triplet does not sum up to 0.
   Example 3:
   Input: nums = [0,0,0]
   Output: [[0,0,0]]
   Explanation: The only possible triplet sums up to 0.

   Constraints:
- 3 <= nums.length <= 3000
- -10^5 <= nums[i] <= 10^5

   Code:
```
import java.util.*;

class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        Arrays.sort(nums);
        for (int i = 0; i < nums.length - 2; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue;
            int left = i + 1, right = nums.length - 1;
            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];
                if (sum == 0) {
                    result.add(Arrays.asList(nums[i], nums[left], nums[right]));
                    while (left < right && nums[left] == nums[left + 1]) left++;
                    while (left < right && nums[right] == nums[right - 1]) right--;
                    left++;
                    right--;
                } else if (sum < 0) left++;
                else right--;
            }
        }
        return result;
    }
```

```java
    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println(sol.threeSum(new int[]{-1, 0, 1, 2, -1, -4}));
        System.out.println(sol.threeSum(new int[]{0, 1, 1}));
        System.out.println(sol.threeSum(new int[]{0, 0, 0}));
    }
}
```

Output:

```
[[-1, -1, 2], [-1, 0, 1]]
[]
[[0, 0, 0]]
```

Time Complexity: $O(n^2)$

Sliding Window

1. **Minimum Size Subarray Sum**
   Given an array of positive integers nums and a positive integer target, return *the* ***minimal length*** *of a*
   subarray
   whose sum is greater than or equal to target. If there is no such subarray, return 0 instead.

   Example 1:
   Input: target = 7, nums = [2,3,1,2,4,3]
   Output: 2
   Explanation: The subarray [4,3] has the minimal length under the problem constraint.
   Example 2:
   Input: target = 4, nums = [1,4,4]
   Output: 1
   Example 3:
   Input: target = 11, nums = [1,1,1,1,1,1,1,1]
   Output: 0

   Constraints:

- 1 <= target <= 10^9
- 1 <= nums.length <= 10^5
- 1 <= nums[i] <= 10^4

Code:

```
class Solution {
    public int minSubArrayLen(int target, int[] nums) {
        int left = 0, sum = 0, minLen = Integer.MAX_VALUE;
        for (int right = 0; right < nums.length; right++) {
            sum += nums[right];
            while (sum >= target) {
                minLen = Math.min(minLen, right - left + 1);
                sum -= nums[left++];
            }
        }
        return minLen == Integer.MAX_VALUE ? 0 : minLen;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println(sol.minSubArrayLen(7, new int[]{2, 3, 1, 2, 4, 3}));
        System.out.println(sol.minSubArrayLen(4, new int[]{1, 4, 4}));
        System.out.println(sol.minSubArrayLen(11, new int[]{1, 1, 1, 1, 1, 1, 1, 1}));
    }
}
```

Output:

```
2
1
0
```

Time Complexity: O(n)

2. **Longest Substring Without Repeating Characters**
Given a string s, find the length of the **longest substring**
without repeating characters.

Example 1:
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
Example 2:
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
Example 3:
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Constraints:

- $0 <= s.length <= 5 * 10^4$
- s consists of English letters, digits, symbols and spaces.

Code:

```
import java.util.HashSet;
import java.util.Set;

class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length();
        int maxLength = 0;
        Set<Character> set = new HashSet<>();
        int left = 0;

        for (int right = 0; right < n; right++) {
            while (set.contains(s.charAt(right))) {
                set.remove(s.charAt(left));
                left++;
            }
            set.add(s.charAt(right));
            maxLength = Math.max(maxLength, right - left + 1);
        }

        return maxLength;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println(sol.lengthOfLongestSubstring("abcabcbb"));
        System.out.println(sol.lengthOfLongestSubstring("bbbbb"));
        System.out.println(sol.lengthOfLongestSubstring("pwwkew"));
```

```
    }
}
```

Output:

```
3
1
3
```

Time Complexity: O(n)

3. **Substring With Concatenation of all Words**
   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**.

  Example 1:
  Input: s = "barfoothefoobarman", words = ["foo","bar"]
  Output: [0,9]
  Explanation:
  The substring starting at 0 is "barfoo". It is the concatenation of ["bar","foo"] which is a permutation of words.
  The substring starting at 9 is "foobar". It is the concatenation of ["foo","bar"] which is a permutation of words.
  Example 2:
  Input: s = "wordgoodgoodgoodbestword", words = ["word","good","best","word"]
  Output: []
  Explanation:
  There is no concatenated substring.
  Example 3:
  Input: s = "barfoofoobarthefoobarman", words = ["bar","foo","the"]
  Output: [6,9,12]
  Explanation:
  The substring starting at 6 is "foobarthe". It is the concatenation of ["foo","bar","the"].
  The substring starting at 9 is "barthefoo". It is the concatenation of ["bar","the","foo"].
  The substring starting at 12 is "thefoobar". It is the concatenation of ["the","foo","bar"].

  Constraints:
- 1 <= s.length <= 10^4
- 1 <= words.length <= 5000
- 1 <= words[i].length <= 30
- s and words[i] consist of lowercase English letters.

  Code:

```
class Solution {

  public List<Integer> findSubstring(String s, String[] words) {
    List<Integer> result = new ArrayList<>();
    if (words.length == 0 || s.length() < words[0].length() * words.length) {
      return result;
    }
    int wordLength = words[0].length();
    int substringLength = wordLength * words.length;
    Map<String, Integer> wordCount = new HashMap<>();
    for (String word : words) {
```

```
            wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);
        }
        for (int i = 0; i < wordLength; i++) {
            int left = i, right = i;
            Map<String, Integer> seenWords = new HashMap<>();
            int count = 0;
            while (right + wordLength <= s.length()) {
                String word = s.substring(right, right + wordLength);
                right += wordLength;
                if (wordCount.containsKey(word)) {
                    seenWords.put(word, seenWords.getOrDefault(word, 0) + 1);
                    count++;
                    while (seenWords.get(word) > wordCount.get(word)) {
                        String leftWord = s.substring(left, left + wordLength);
                        seenWords.put(leftWord, seenWords.get(leftWord) - 1);
                        left += wordLength;
                        count--;
                    }
                    if (count == words.length) {
                        result.add(left);
                    }
                } else {
                    seenWords.clear();
                    count = 0;
                    left = right;
                }
            }
        }
        return result;
    }
    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println(sol.findSubstring("barfoothefoobarman", new String[]{"foo", "bar"}));
        System.out.println(sol.findSubstring("wordgoodgoodgoodbestword", new String[]{"word", "good", "best", "word"}));
        System.out.println(sol.findSubstring("barfoofoobarthefoobarman", new String[]{"bar", "foo", "the"}));
    }
}
```

Output:

```
[0, 9]
[]
[6, 9, 12]
```

Time Complexity: O(s.length()×wordLength

4. **Minimum window substring**
   Given two strings s and t of lengths m and n respectively, return *the **minimum window substring***
   of s *such that every character in* t ***(including duplicates)*** *is included in the window*. If there is no such substring, return *the empty string* "".
   The testcases will be generated such that the answer is **unique**.

   Example 1:
   Input: s = "ADOBECODEBANC", t = "ABC"
   Output: "BANC"
   Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.
   Example 2:
   Input: s = "a", t = "a"
   Output: "a"
   Explanation: The entire string s is the minimum window.
   Example 3:
   Input: s = "a", t = "aa"
   Output: ""
   Explanation: Both 'a's from t must be included in the window.
   Since the largest window of s only has one 'a', return empty string.

   Constraints:

- m == s.length
- n == t.length
- 1 <= m, n <= 10^5
- s and t consist of uppercase and lowercase English letters.

Code:

```
import java.util.*;

class Solution {
   public String minWindow(String s, String t) {
      if (s.length() < t.length()) return "";

      Map<Character, Integer> tCount = new HashMap<>();
      for (char c : t.toCharArray()) {
         tCount.put(c, tCount.getOrDefault(c, 0) + 1);
      }

      int left = 0, right = 0, required = tCount.size(), formed = 0;
      Map<Character, Integer> windowCount = new HashMap<>();
      int[] ans = {-1, 0, 0};  // {length of window, left, right}

      while (right < s.length()) {
         char c = s.charAt(right);
         windowCount.put(c, windowCount.getOrDefault(c, 0) + 1);

         if (tCount.containsKey(c) && windowCount.get(c).intValue() ==
tCount.get(c).intValue()) {
            formed++;
```

```
            }

            while (left <= right && formed == required) {
                char leftChar = s.charAt(left);
                if (ans[0] == -1 || right - left + 1 < ans[0]) {
                    ans[0] = right - left + 1;
                    ans[1] = left;
                    ans[2] = right;
                }

                windowCount.put(leftChar, windowCount.get(leftChar) - 1);
                if (tCount.containsKey(leftChar) && windowCount.get(leftChar).intValue() <
tCount.get(leftChar).intValue()) {
                    formed--;
                }
                left++;
            }

            right++;
        }

        return ans[0] == -1 ? "" : s.substring(ans[1], ans[2] + 1);
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println(sol.minWindow("ADOBECODEBANC", "ABC"));
        System.out.println(sol.minWindow("a", "a"));
    }
}
```

Output:

```
BANC
a
```

Time Complexity: O(n)

Stack

1. **Valid parantheses**
   Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
   An input string is valid if:
1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:
Input: s = "()"
Output: true
Example 2:
Input: s = "()[]{}"
Output: true
Example 3:
Input: s = "(]"
Output: false
Example 4:
Input: s = "([])"
Output: true

Constraints:
- 1 <= s.length <= 10^4
- s consists of parentheses only '()[]{}'.

Code:
```
import java.util.Stack;

class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();

        for (char c : s.toCharArray()) {
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            } else {
                if (stack.isEmpty()) return false;
                char top = stack.pop();
                if ((c == ')' && top != '(') || (c == '}' && top != '{') || (c == ']' && top != '[')) {
                    return false;
                }
            }
        }

        return stack.isEmpty();
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
```

```
        System.out.println(sol.isValid("()"));
        System.out.println(sol.isValid("()[]{}"));
        System.out.println(sol.isValid("(]"));
        System.out.println(sol.isValid("([])"));
    }
}
```

Output:

```
true
true
false
true
```

Time complexity: O(n)

2. **Simplify path**

You are given an *absolute* path for a Unix-style file system, which always begins with a slash '/'. Your task is to transform this absolute path into its **simplified canonical path**.

The *rules* of a Unix-style file system are as follows:

- A single period '.' represents the current directory.
- A double period '..' represents the previous/parent directory.
- Multiple consecutive slashes such as '//' and '///' are treated as a single slash '/'.
- Any sequence of periods that does **not match** the rules above should be treated as a **valid directory or file name**. For example, '...' and '....' are valid directory or file names.

The simplified canonical path should follow these *rules*:

- The path must start with a single slash '/'.
- Directories within the path must be separated by exactly one slash '/'.
- The path must not end with a slash '/', unless it is the root directory.
- The path must not have any single or double periods ('.' and '..') used to denote current or parent directories.

Return the **simplified canonical path**.

Example 1:
Input: path = "/home/"
Output: "/home"
Explanation:
The trailing slash should be removed.
Example 2:
Input: path = "/home//foo/"
Output: "/home/foo"
Explanation:
Multiple consecutive slashes are replaced by a single one.
Example 3:
Input: path = "/home/user/Documents/../Pictures"
Output: "/home/user/Pictures"
Explanation:
A double period ".." refers to the directory up a level (the parent directory).
Example 4:
Input: path = "/../"
Output: "/"
Explanation:
Going one level up from the root directory is not possible.
Example 5:
Input: path = "/.../a/../b/c/../d/./"
Output: "/.../b/d"
Explanation:
"..." is a valid name for a directory in this problem.

Constraints:

- 1 <= path.length <= 3000
- path consists of English letters, digits, period '.', slash '/' or '_'.
- path is a valid absolute Unix path.

Code:

```
import java.util.Stack;
```

```java
class Solution {
    public String simplifyPath(String path) {
        Stack<String> stack = new Stack<>();
        String[] parts = path.split("/");

        for (String part : parts) {
            if (part.equals("") || part.equals(".")) {
                continue;
            }
            if (part.equals("..")) {
                if (!stack.isEmpty()) {
                    stack.pop();
                }
            } else {
                stack.push(part);
            }
        }

        StringBuilder result = new StringBuilder();
        for (String dir : stack) {
            result.append("/").append(dir);
        }

        return result.length() == 0 ? "/" : result.toString();
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println(sol.simplifyPath("/home/"));
        System.out.println(sol.simplifyPath("/home//foo/"));
        System.out.println(sol.simplifyPath("/home/user/Documents/../Pictures"));
        System.out.println(sol.simplifyPath("/../"));
        System.out.println(sol.simplifyPath("/.../a/../b/c/../d/./"));
    }
}
```

Output:

```
/home
/home/foo
/home/user/Pictures
/
/.../b/d
```

Time Complexity: O(n)

3. **Min stack**

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on the top of the stack.
- int top() gets the top element of the stack.
- int getMin() retrieves the minimum element in the stack.

You must implement a solution with O(1) time complexity for each function.

Example 1:

Input

["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output

[null,null,null,null,-3,null,0,-2]

Explanation

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top();    // return 0
minStack.getMin(); // return -2
```

Constraints:

- -2^{31} <= val <= $2^{31} - 1$
- Methods pop, top and getMin operations will always be called on **non-empty** stacks.
- At most $3 * 10^4$ calls will be made to push, pop, top, and getMin.

Code:

```
public class Main {
    public static void main(String[] args) {
        MinStack minStack = new MinStack();
        minStack.push(-2);
        minStack.push(0);
        minStack.push(-3);

        System.out.println(minStack.getMin());
        minStack.pop();
        System.out.println(minStack.top());
        System.out.println(minStack.getMin());
    }
}
```

Output:

```
-3
0
-2
```

Time Complexity: O(1)

4. **Evaluate Reverse Polish Notation**
You are given an array of strings tokens that represents an arithmetic expression in a [Reverse Polish Notation](#).
Evaluate the expression. Return *an integer that represents the value of the expression*.
Note that:

- The valid operators are '+', '-', '*', and '/'.
- Each operand may be an integer or another expression.
- The division between two integers always **truncates toward zero**.
- There will not be any division by zero.
- The input represents a valid arithmetic expression in a reverse polish notation.
- The answer and all the intermediate calculations can be represented in a **32-bit** integer.

Example 1:
Input: tokens = ["2","1","+","3","*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9
Example 2:
Input: tokens = ["4","13","5","/","+"]
Output: 6
Explanation: (4 + (13 / 5)) = 6
Example 3:
Input: tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
Output: 22
Explanation: ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22

Constraints:

- 1 <= tokens.length <= 10^4
- tokens[i] is either an operator: "+", "-", "*", or "/", or an integer in the range [-200, 200].

Code:
```
import java.util.Stack;

class Solution {
    public int evalRPN(String[] tokens) {
        Stack<Integer> stack = new Stack<>();
        for (String token : tokens) {
            if ("+-*/".contains(token)) {
                int b = stack.pop();
                int a = stack.pop();
                if (token.equals("+")) stack.push(a + b);
                else if (token.equals("-")) stack.push(a - b);
                else if (token.equals("*")) stack.push(a * b);
                else stack.push(a / b);
            } else {
                stack.push(Integer.parseInt(token));
```

```
            }
        }
        return stack.pop();
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        String[] tokens1 = {"2", "1", "+", "3", "*"};
        System.out.println(solution.evalRPN(tokens1));

        String[] tokens2 = {"4", "13", "5", "/", "+"};
        System.out.println(solution.evalRPN(tokens2));

        String[] tokens3 = {"10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"};
        System.out.println(solution.evalRPN(tokens3));
    }
}
```

Output:

```
9
6
22
```

Time Complexity: O(n)

5. **Basic calculator**

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().

Example 1:
Input: s = "1 + 1"
Output: 2
Example 2:
Input: s = " 2-1 + 2 "
Output: 3
Example 3:
Input: s = "(1+(4+5+2)-3)+(6+8)"
Output: 23

Constraints:

- 1 <= s.length <= 3 * 10^5
- s consists of digits, '+', '-', '(', ')', and ' '.
- s represents a valid expression.
- '+' is **not** used as a unary operation (i.e., "+1" and "+(2 + 3)" is invalid).
- '-' could be used as a unary operation (i.e., "-1" and "-(2 + 3)" is valid).
- There will be no two consecutive operators in the input.
- Every number and running calculation will fit in a signed 32-bit integer.

Code:

```
class Solution {
    public int evalRPN(String[] tokens) {
        Stack<Integer> stack = new Stack<>();
        for (String token : tokens) {
            if ("+-*/".contains(token)) {
                int b = stack.pop();
                int a = stack.pop();
                if (token.equals("+")) stack.push(a + b);
                else if (token.equals("-")) stack.push(a - b);
                else if (token.equals("*")) stack.push(a * b);
                else stack.push(a / b);
            } else {
                stack.push(Integer.parseInt(token));
            }
        }
        return stack.poclass Solution {
    public int calculate(String s) {
        Stack<Integer> stack = new Stack<>();
        int result = 0, number = 0, sign = 1;
        for (char c : s.toCharArray()) {
            if (Character.isDigit(c)) {
                number = number * 10 + (c - '0');
            } else if (c == '+') {
                result += sign * number;
                number = 0;
```

```
                sign = 1;
            } else if (c == '-') {
                result += sign * number;
                number = 0;
                sign = -1;
            } else if (c == '(') {
                stack.push(result);
                stack.push(sign);
                result = 0;
                sign = 1;
            } else if (c == ')') {
                result += sign * number;
                number = 0;
                result *= stack.pop();
                result += stack.pop();
            }
        }
        return result + (sign * number);
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        System.out.println(solution.calculate("1 + 1"));
        System.out.println(solution.calculate(" 2-1 + 2 "));
        System.out.println(solution.calculate("(1+(4+5+2)-3)+(6+8)"));
    }
}
p();
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        String[] tokens1 = {"2", "1", "+", "3", "*"};
        System.out.println(solution.evalRPN(tokens1));

        String[] tokens2 = {"4", "13", "5", "/", "+"};
        System.out.println(solution.evalRPN(tokens2));

        String[] tokens3 = {"10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"};
        System.out.println(solution.evalRPN(tokens3));
    }
}
```

Output:

```
2
3
23
```

Time Complexity: O(n)

Binary Search

1. **Search Insert Positon**
   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.

   Example 1:
   Input: nums = [1,3,5,6], target = 5
   Output: 2
   Example 2:
   Input: nums = [1,3,5,6], target = 2
   Output: 1
   Example 3:
   Input: nums = [1,3,5,6], target = 7
   Output: 4

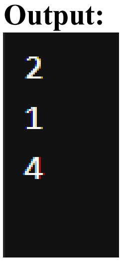
   Constraints:

- 1 <= nums.length <= 10^4
- -10^4 <= nums[i] <= 10^4
- nums contains **distinct** values sorted in **ascending** order.
- -10^4 <= target <= 10^4

Code:

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) return mid;
            if (nums[mid] < target) left = mid + 1;
            else right = mid - 1;
        }
        return left;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        System.out.println(solution.searchInsert(new int[]{1, 3, 5, 6}, 5));
        System.out.println(solution.searchInsert(new int[]{1, 3, 5, 6}, 2));
        System.out.println(solution.searchInsert(new int[]{1, 3, 5, 6}, 7));
    }
}
```

Output:



Time Complexity: O(log n)

2. **Search a 2D matrix:**

You are given an m x n integer matrix matrix with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true *if* target *is in* matrix *or* false *otherwise*.

You must write a solution in O(log(m * n)) time complexity.

Example 1:

1	3	5	7
10	11	16	20
23	30	34	60

Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
Output: true
Example 2:

1	3	5	7
10	11	16	20
23	30	34	60

Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13
Output: false

Constraints:

- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 100
- -10^4 <= matrix[i][j], target <= 10^4

Code:

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        int m = matrix.length, n = matrix[0].length;
```

```
        int left = 0, right = m * n - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int midValue = matrix[mid / n][mid % n];
            if (midValue == target) return true;
            if (midValue < target) left = mid + 1;
            else right = mid - 1;
        }
        return false;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        System.out.println(solution.searchMatrix(new int[][]{
            {1, 3, 5, 7},
            {10, 11, 16, 20},
            {23, 30, 34, 60}
        }, 3));
        System.out.println(solution.searchMatrix(new int[][]{
            {1, 3, 5, 7},
            {10, 11, 16, 20},
            {23, 30, 34, 60}
        }, 13));
    }
}
```

Output:

```
true
false
```

Time Complexity: O(log(m×n))

3. **Find Peak element**

A peak element is an element that is strictly greater than its neighbors.
Given a **0-indexed** integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.
You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.
You must write an algorithm that runs in O(log n) time.

Example 1:
Input: nums = [1,2,3,1]
Output: 2
Explanation: 3 is a peak element and your function should return the index number 2.
Example 2:
Input: nums = [1,2,1,3,5,6,4]
Output: 5
Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

Constraints:
- 1 <= nums.length <= 1000
- -2^{31} <= nums[i] <= 2^{31} - 1
- nums[i] != nums[i + 1] for all valid i.

Code:
```
class Solution {
    public int findPeakElement(int[] nums) {
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > nums[mid + 1]) right = mid;
            else left = mid + 1;
        }
        return left;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        System.out.println(solution.findPeakElement(new int[]{1, 2, 3, 1}));
        System.out.println(solution.findPeakElement(new int[]{1, 2, 1, 3, 5, 6, 4}));
    }
}
```

Output:
```
2
5
```

Time Complexity: O(log n)

4. **Search in rotated sorted array**
   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.

   Example 1:
   Input: nums = [4,5,6,7,0,1,2], target = 0
   Output: 4
   Example 2:
   Input: nums = [4,5,6,7,0,1,2], target = 3
   Output: -1
   Example 3:
   Input: nums = [1], target = 0
   Output: -1

   Constraints:

- 1 <= nums.length <= 5000
- -10^4 <= nums[i] <= 10^4
- All values of nums are **unique**.
- nums is an ascending array that is possibly rotated.
- -10^4 <= target <= 10^4

Code:

```
class Solution {
    public int search(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) return mid;
            if (nums[left] <= nums[mid]) {
                if (nums[left] <= target && target < nums[mid]) right = mid - 1;
                else left = mid + 1;
            } else {
                if (nums[mid] < target && target <= nums[right]) left = mid + 1;
                else right = mid - 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        System.out.println(solution.search(new int[]{4, 5, 6, 7, 0, 1, 2}, 0));
        System.out.println(solution.search(new int[]{4, 5, 6, 7, 0, 1, 2}, 3));
        System.out.println(solution.search(new int[]{1}, 0));
    }
```

}

Output:

```
4
-1
-1
```

Time Complexity: O(log n)

5. **Find First and Last Position of Element in Sorted Array**
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.

Example 1:
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
Example 2:
Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]
Example 3:
Input: nums = [], target = 0
Output: [-1,-1]

Constraints:

- 0 <= nums.length <= 10^5
- -10^9 <= nums[i] <= 10^9
- nums is a non-decreasing array.
- -10^9 <= target <= 10^9

Code:

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        int[] result = {-1, -1};
        result[0] = findBound(nums, target, true);
        if (result[0] != -1) result[1] = findBound(nums, target, false);
        return result;
    }

    private int findBound(int[] nums, int target, boolean isFirst) {
        int left = 0, right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) {
                if (isFirst) {
                    if (mid == left || nums[mid - 1] != target) return mid;
                    right = mid - 1;
                } else {
                    if (mid == right || nums[mid + 1] != target) return mid;
                    left = mid + 1;
                }
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return -1;
    }
```

```java
    public static void main(String[] args) {
        Solution solution = new Solution();
        System.out.println(java.util.Arrays.toString(solution.searchRange(new int[]{5, 7, 7, 8, 8, 10}, 8)));
        System.out.println(java.util.Arrays.toString(solution.searchRange(new int[]{5, 7, 7, 8, 8, 10}, 6)));
        System.out.println(java.util.Arrays.toString(solution.searchRange(new int[]{}, 0)));
    }
}
```

Output:

```
[3, 4]
[-1, -1]
[-1, -1]
```

Time Complexity: O(log n)

6. **Find Minimum in Rotated Sorted Array**
Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

- [4,5,6,7,0,1,2] if it was rotated 4 times.
- [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].
Given the sorted rotated array nums of **unique** elements, return *the minimum element of this array*.
You must write an algorithm that runs in O(log n) time.

Example 1:
Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.
Example 2:
Input: nums = [4,5,6,7,0,1,2]
Output: 0
Explanation: The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.
Example 3:
Input: nums = [11,13,15,17]
Output: 11
Explanation: The original array was [11,13,15,17] and it was rotated 4 times.

Constraints:

- n == nums.length
- 1 <= n <= 5000
- -5000 <= nums[i] <= 5000
- All the integers of nums are **unique**.
- nums is sorted and rotated between 1 and n times.

Code:
```
class Solution {
    public int findMin(int[] nums) {
        int left = 0, right = nums.length - 1;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > nums[right]) left = mid + 1;
            else right = mid;
        }
        return nums[left];
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        System.out.println(solution.findMin(new int[]{3, 4, 5, 1, 2}));
        System.out.println(solution.findMin(new int[]{4, 5, 6, 7, 0, 1, 2}));
        System.out.println(solution.findMin(new int[]{11, 13, 15, 17}));
    }
}
```

Output:

```
1
0
11
```

Time Complexity: O(log n)