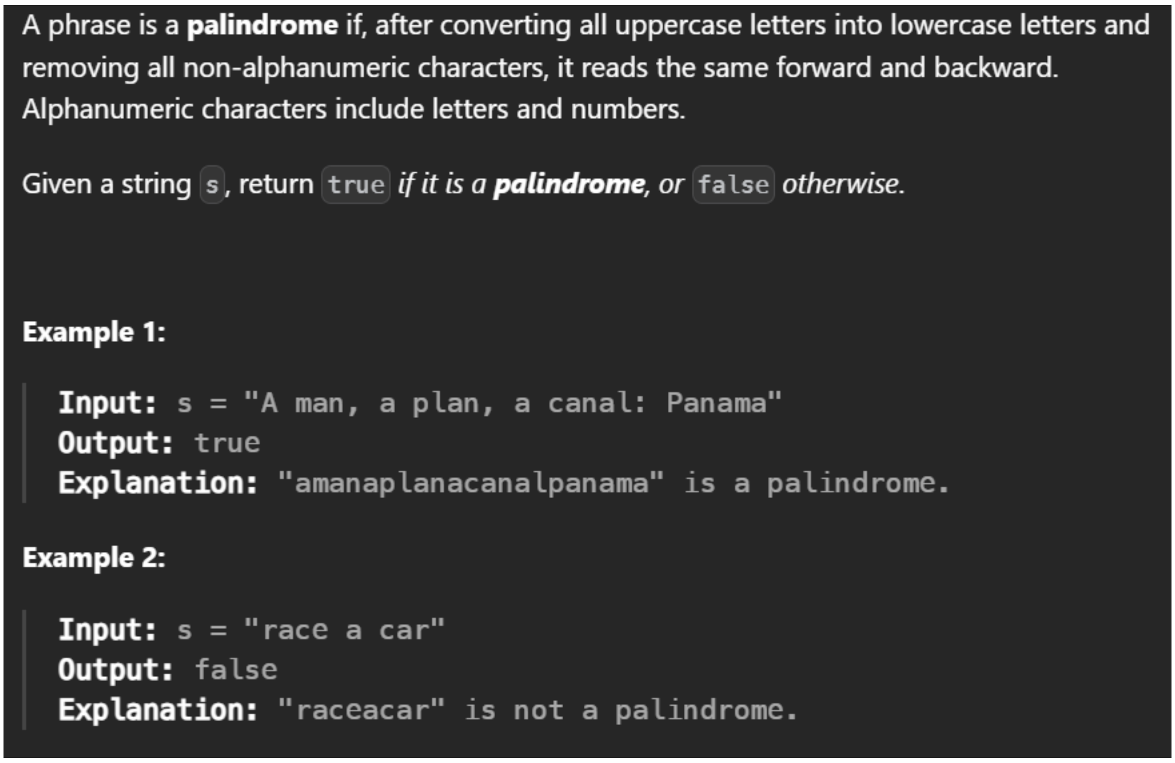
DSA Practice – 9

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.
```

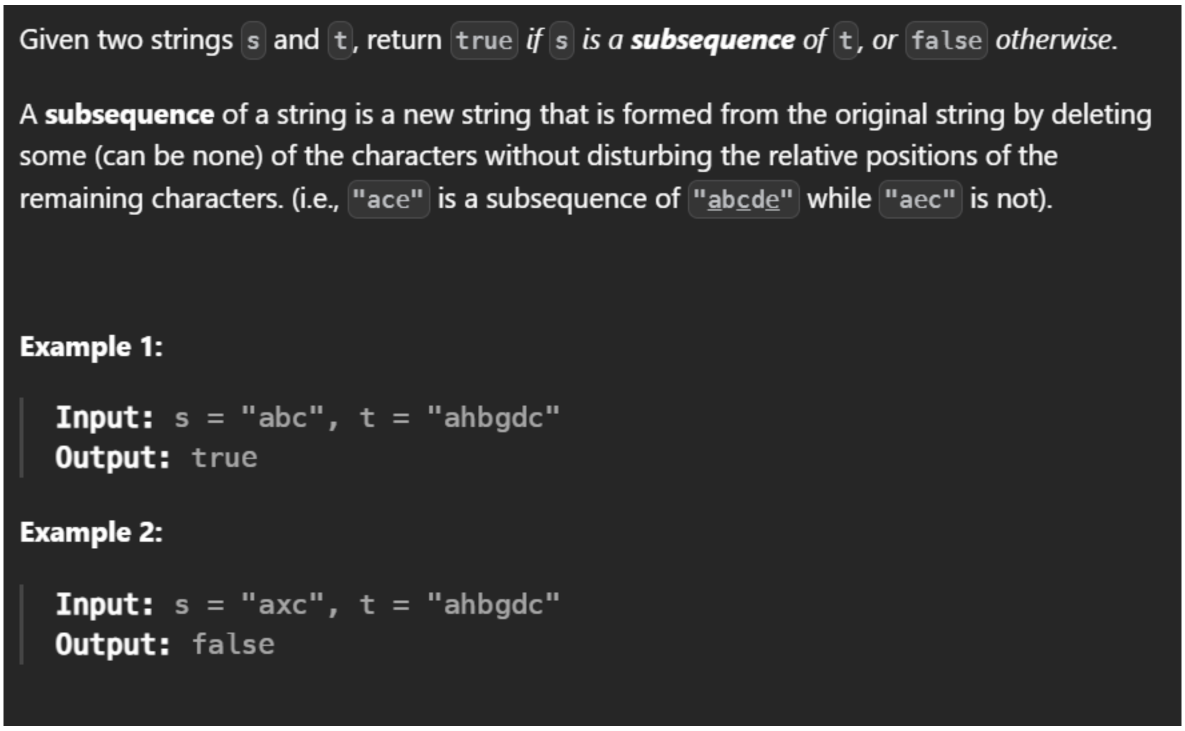
Code:

```
class Solution {
    public boolean isPalindrome(String s) {
        if (s.isEmpty()) {
            return true;
        }
        int sa = 0;
        int l = s.length() - 1;
        while (sa <= l) {
            char cf = s.charAt(sa);
            char cl = s.charAt(l);
            if (!Character.isLetterOrDigit(cf)) {
                sa++;
            } else if (!Character.isLetterOrDigit(cl)) {
                l--;
            } else {
                if (Character.toLowerCase(cf) != Character.toLowerCase(cl)) {
                    return false;
                }
                sa++;
                l--;
            }
        }
        return true;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        String testCase = "A man, a plan, a canal: Panama";
        boolean result = solution.isPalindrome(testCase);
        System.out.println( result);
    }
}
```

Output: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
```

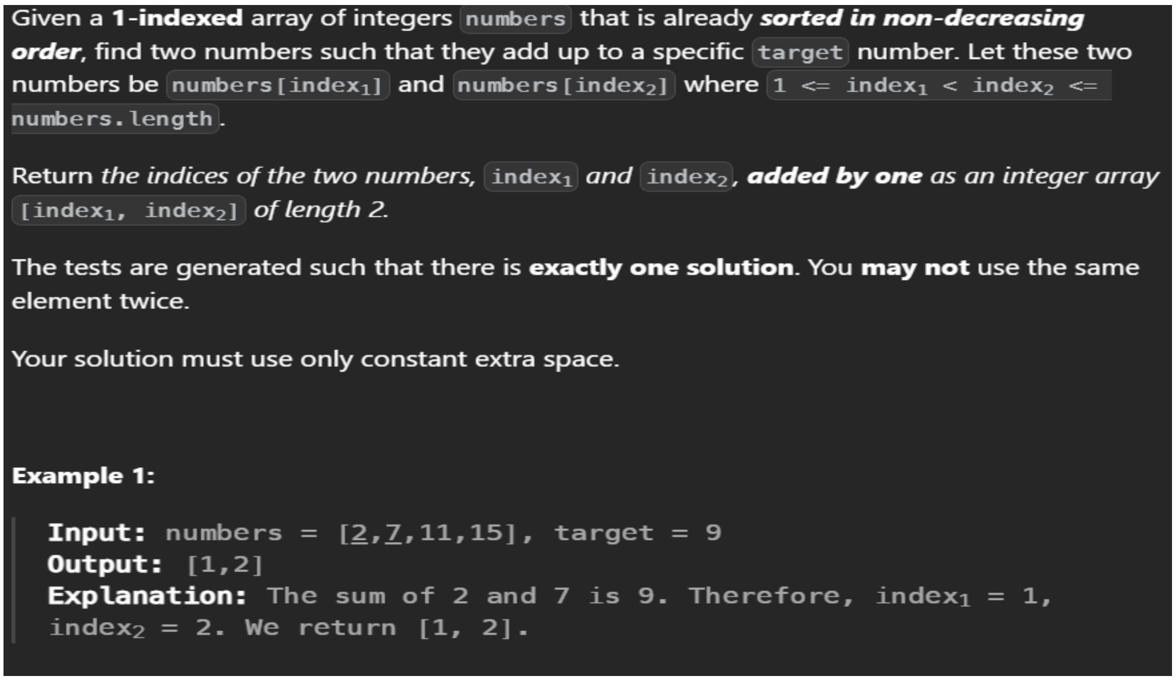
Code:

```
class Solution {
  public boolean isSubsequence(String s, String t) {
        int sp = 0, tp = 0;
        while (sp < s.length() && tp < t.length()) {
            if (s.charAt(sp) == t.charAt(tp)) {
                sp++;
            }
            tp++;
        }
        return sp == s.length();
    }
    public static void main(String[] args) {
        Solution solution = new Solution()
        String s = "abc";
        String t = "ahbgdc";
        boolean result = solution.isSubsequence(s, t);
        System.out.println( result);
    }
}
```

Output:true

Time Complexity: O(n)

3. Two Sum-II


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[index1]` and `numbers[index2]` where `1 <= index1 < index2 <= numbers.length`.

Return *the indices of the two numbers,* `index1` *and* `index2`*,* ***added by one*** *as an integer array* `[index1, index2]` *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, index1 = 1,
index2 = 2. We return [1, 2].
```

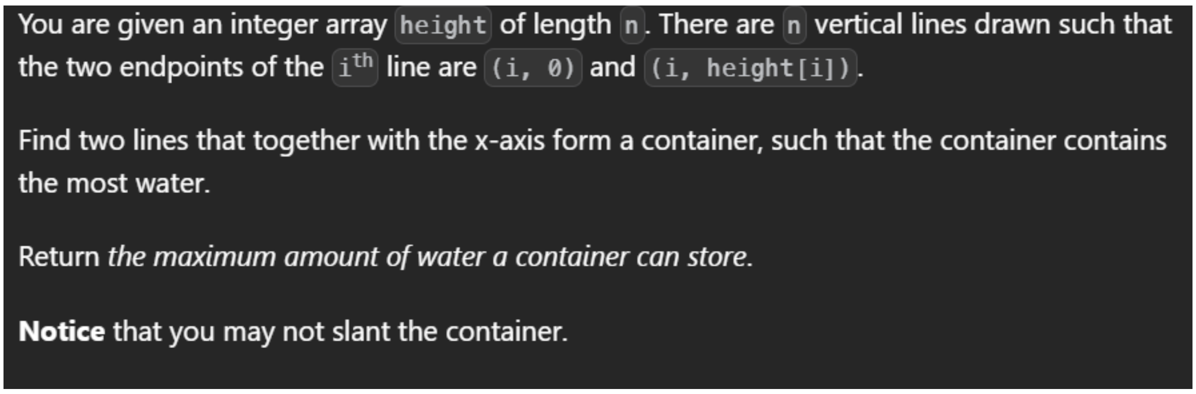
Code:

```
class Solution {
    public int[] twoSum(int[] numbers, int target) {
        int left = 0;
        int right = numbers.length - 1;
        while (left < right) {
            int total = numbers[left] + numbers[right];
            if (total == target) {
                return new int[]{left + 1, right + 1};
            } else if (total > target) {
                right--;
            } else {
                left++;
            }
        }
        return new int[]{-1, -1};
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] numbers = {2, 7, 11, 15};
        int target = 9;
        int[] result = solution.twoSum(numbers, target);
        System.out.println( [" + result[0] + ", " + result[1] + "]);
    }
}
```

Output: [1,2]

Time Complexity: O(n)

4.Container with most water

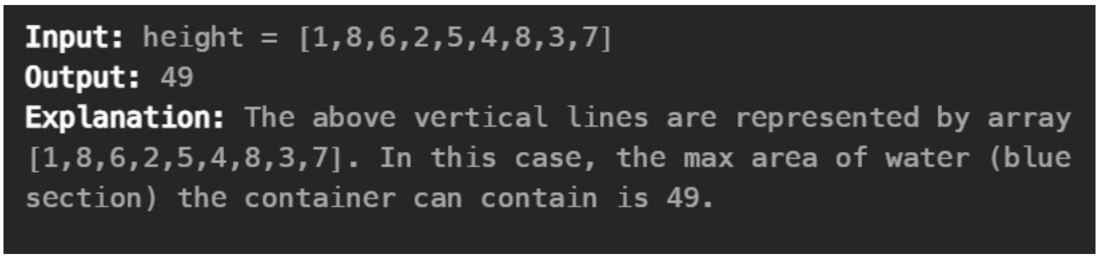


You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the i^{th} line are `(i, 0)` and `(i, height[i])`.

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.



```
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.
```

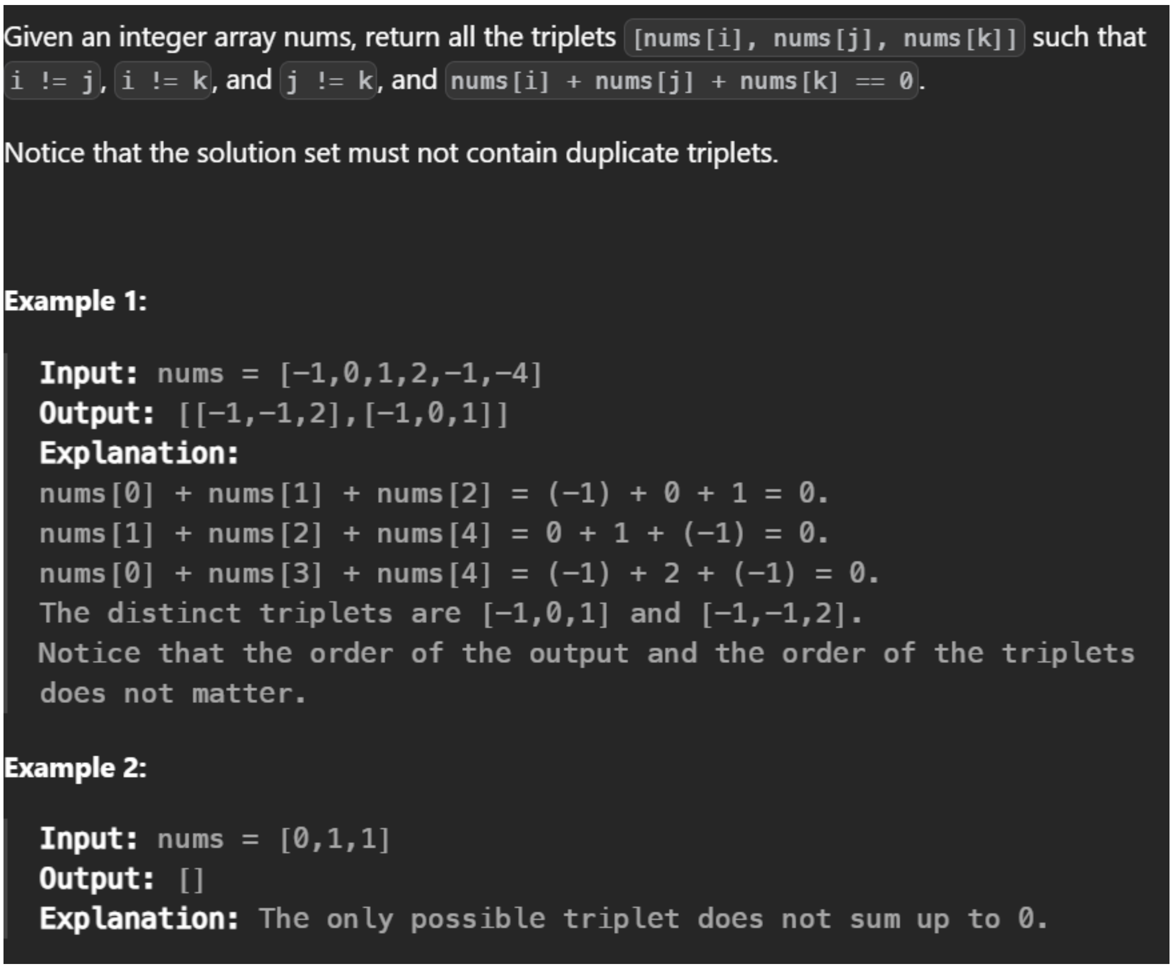
Code:

```
class Solution {
    public int maxArea(int[] height) {
        int left = 0;
        int right = height.length - 1;
        int maxA = 0;
        while (left < right) {
            int currA = Math.min(height[left], height[right]) * (right - left);
            maxA = Math.max(maxA, currA);
            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }
        return maxA;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] height = {1, 8, 6, 2, 5, 4, 8, 3, 7};
        int result = solution.maxArea(height);
        System.out.println( result);
    }
}
```

Output:49

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.
```

Code:

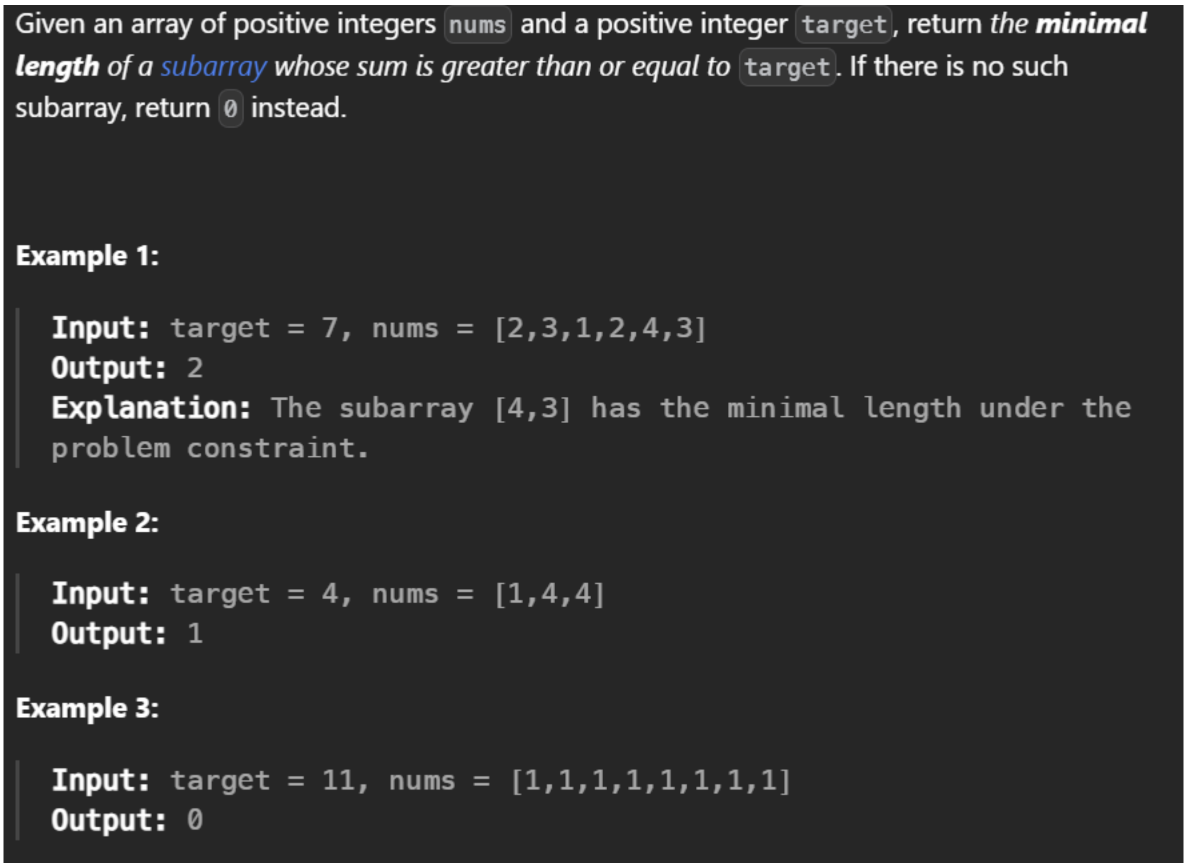
```
import java.util.*;
class Solution {
   public List<List<Integer>> threeSum(int[] nums) {
       List<List<Integer>> output = new ArrayList<>();
       Arrays.sort(nums);
       if ((nums[0] >= 0 && nums[nums.length - 1] != 0) || nums[nums.length - 1] < 0) {
           return output;
       }
       for (int index = 0; index < nums.length; index++) {
           if (index > 0 && nums[index] == nums[index - 1]) {
               continue;
           }
           if (nums[index] > 0) {
               break;
           }
           int left = index + 1;
```

```
            int right = nums.length - 1;
            while (left < right) {
                int threeSum = nums[index] + nums[left] + nums[right];

                if (threeSum > 0) {
                    right--;
                } else if (threeSum < 0) {
                    left++;
                } else { // threeSum == 0
                    output.add(Arrays.asList(nums[index], nums[left], nums[right]));
                    left++;
                    while (left < right && nums[left] == nums[left - 1]) {
                        left++;
                    }
                }
            }
        }
        return output;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {-1, 0, 1, 2, -1, -4};
        List<List<Integer>> result = solution.threeSum(nums);
        System.out.println( result);
    }
}
```

Output: [[-1, -1, 2], [-1, 0, 1]]

Time Complexity: O(n^2)

6. 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
```

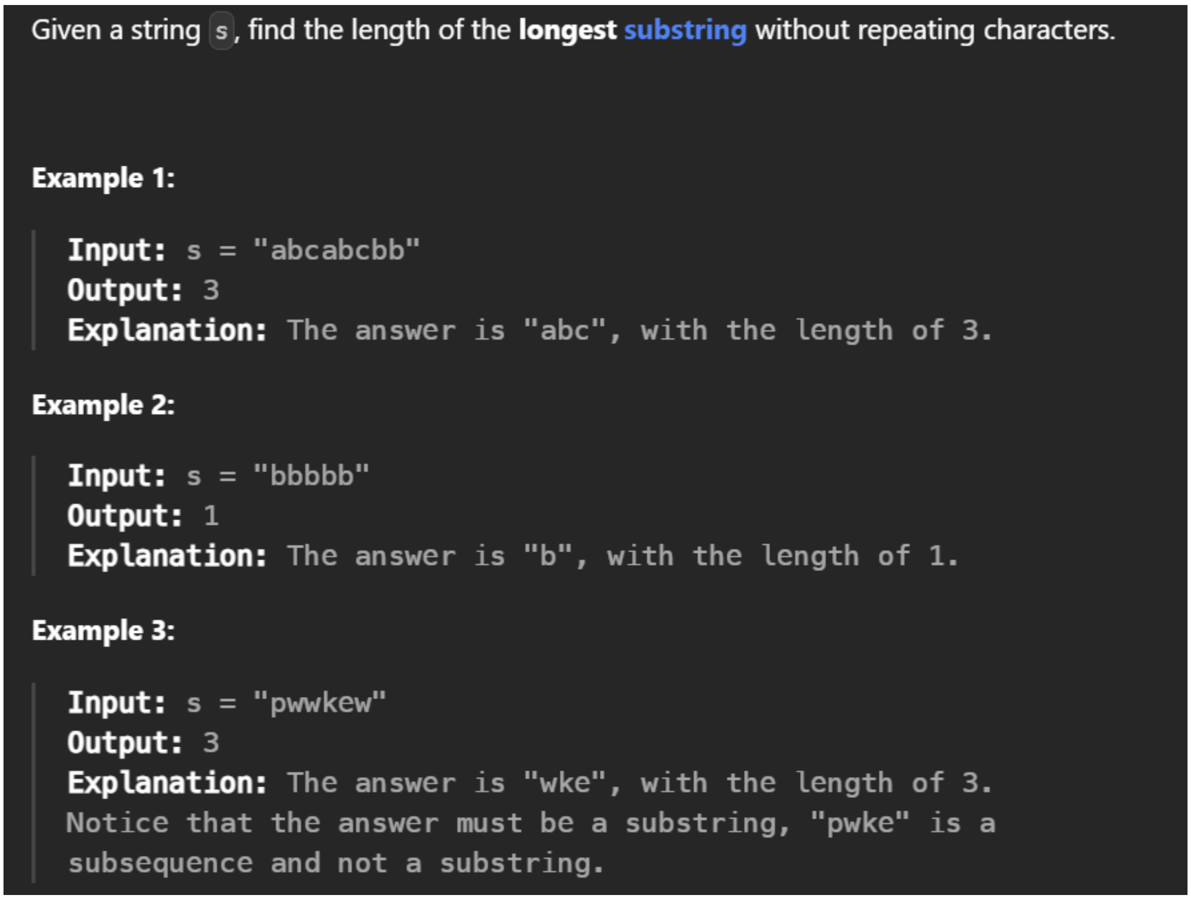
Code:

```
class Solution {
    public int minSubArrayLen(int target, int[] nums) {
        int m = Integer.MAX_VALUE;
        int s = 0;
        int c = 0;
        for (int e = 0; e < nums.length; e++) {
            c += nums[e];
            while (c >= target) {
                m = Math.min(m, e - s + 1);
                c -= nums[s];
                s++;
            }
        }
        return m == Integer.MAX_VALUE ? 0 : m;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        int target = 7;
        int[] nums = {2, 3, 1, 2, 4, 3};
        int result = solution.minSubArrayLen(target, nums);
        System.out.println( result);
    }
}
```

Output:2

Time Complexity:O(n)

7.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.
```

Code:

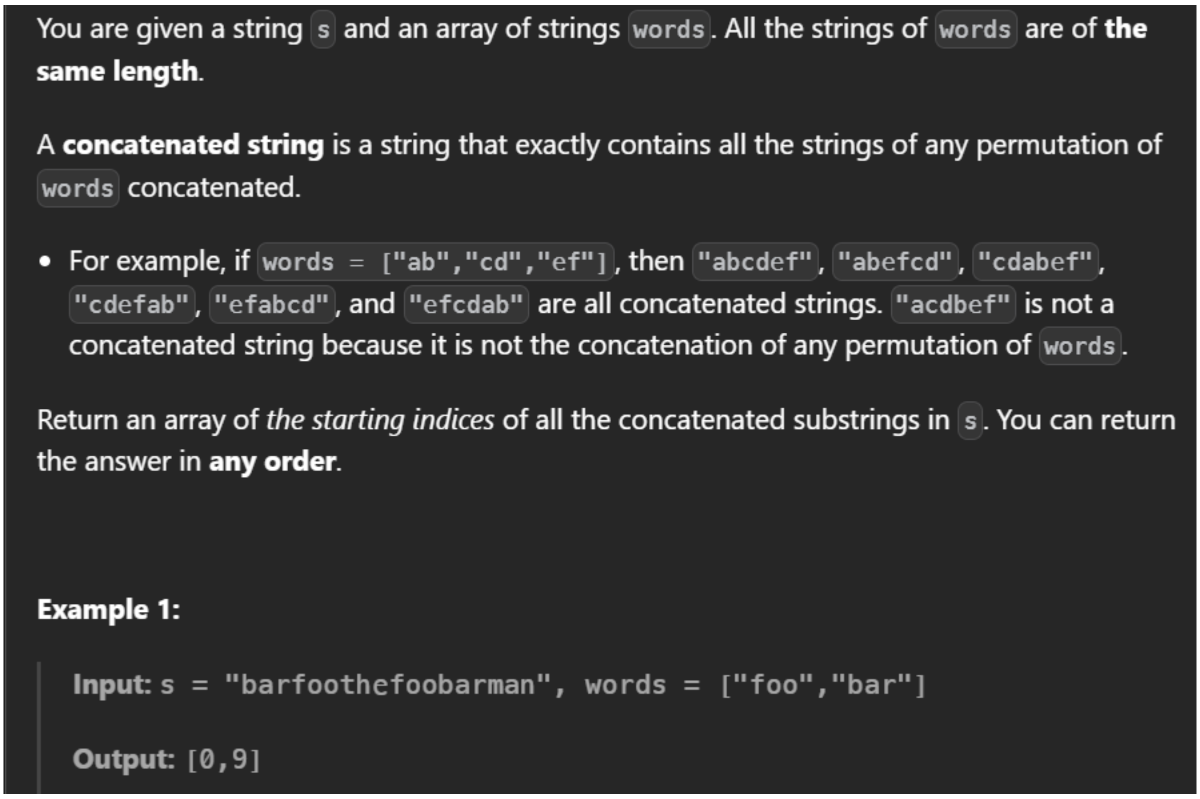
```
import java.util.*;
class Solution {
    public int lengthOfLongestSubstring(String s) {
        Set<Character> set = new HashSet<>();
        int maxLength = 0;
        int left = 0;
        for (int right = 0; right < s.length(); right++) {
            if (!set.contains(s.charAt(right))) {
                set.add(s.charAt(right));
                maxLength = Math.max(maxLength, right - left + 1);
            } else {
                while (s.charAt(left) != s.charAt(right)) {
```

```
                set.remove(s.charAt(left));
                left++;
            }
            set.remove(s.charAt(left));
            left++;
            set.add(s.charAt(right));
        }
    }
    return maxLength;
  }
  public static void main(String[] args) {
    Solution solution = new Solution();
    String input = "abcabcbb";
    int result = solution.lengthOfLongestSubstring(input);
    System.out.println( result);
  }
}
```

Output:3

Time Complexity:O(n)

8.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]`

Code:

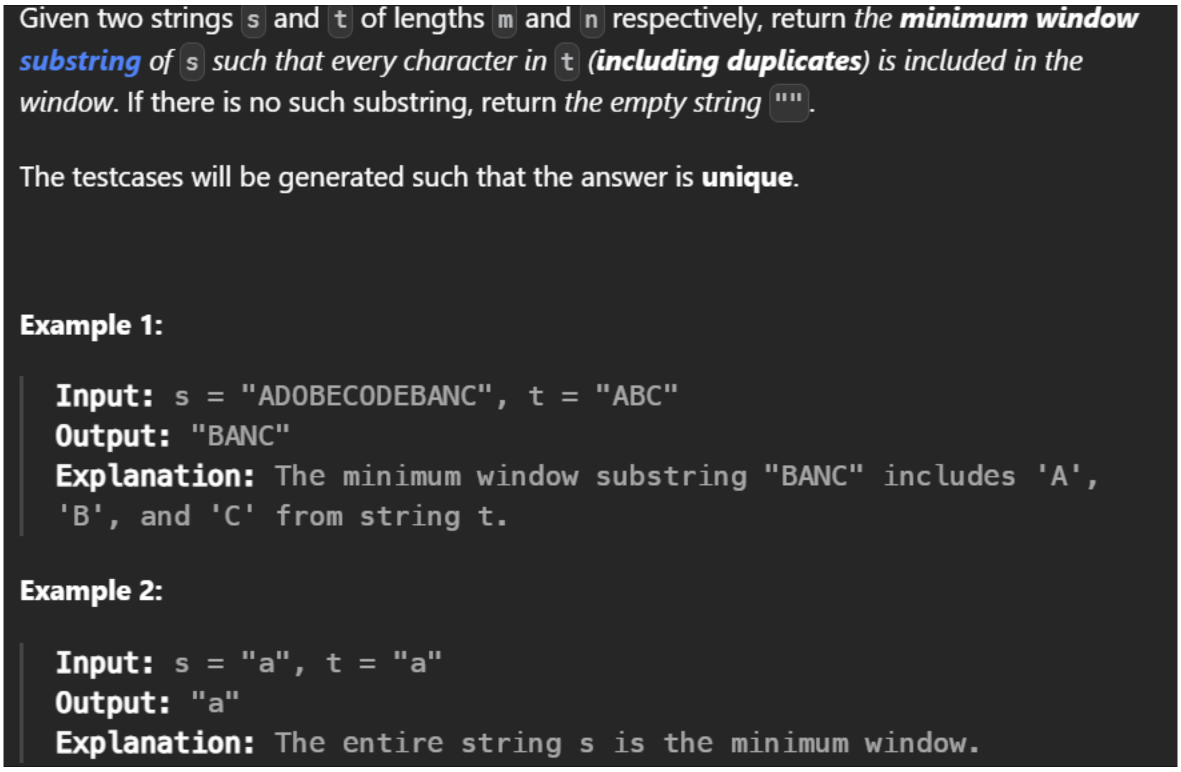
```java
import java.util.*;
class Solution {
   public List<Integer> findSubstring(String s, String[] words) {
      final Map<String, Integer> counts = new HashMap<>();
      for (final String word : words) {
         counts.put(word, counts.getOrDefault(word, 0) + 1);
      }
      final List<Integer> indexes = new ArrayList<>();
      final int n = s.length(), num = words.length, len = words[0].length();
      for (int i = 0; i < n - num * len + 1; i++) {
         final Map<String, Integer> seen = new HashMap<>();
         int j = 0;
         while (j < num) {
            final String word = s.substring(i + j * len, i + (j + 1) * len);
            if (counts.containsKey(word)) {
               seen.put(word, seen.getOrDefault(word, 0) + 1);
               if (seen.get(word) > counts.getOrDefault(word, 0)) {
                  break;
               }
            } else {
               break;
            }
            j++;
         }
         if (j == num) {
            indexes.add(i);
         }
      }
      return indexes;
   }
   public static void main(String[] args) {
      Solution solution = new Solution();

      String s = "barfoothefoobarman";
```

```
        String[] words = {"foo", "bar"};
        List<Integer> result = solution.findSubstring(s, words);
        System.out.println(result);
    }
}
```

Output: [0, 9]

Time Complexity: O(n * m)

9. 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.
```

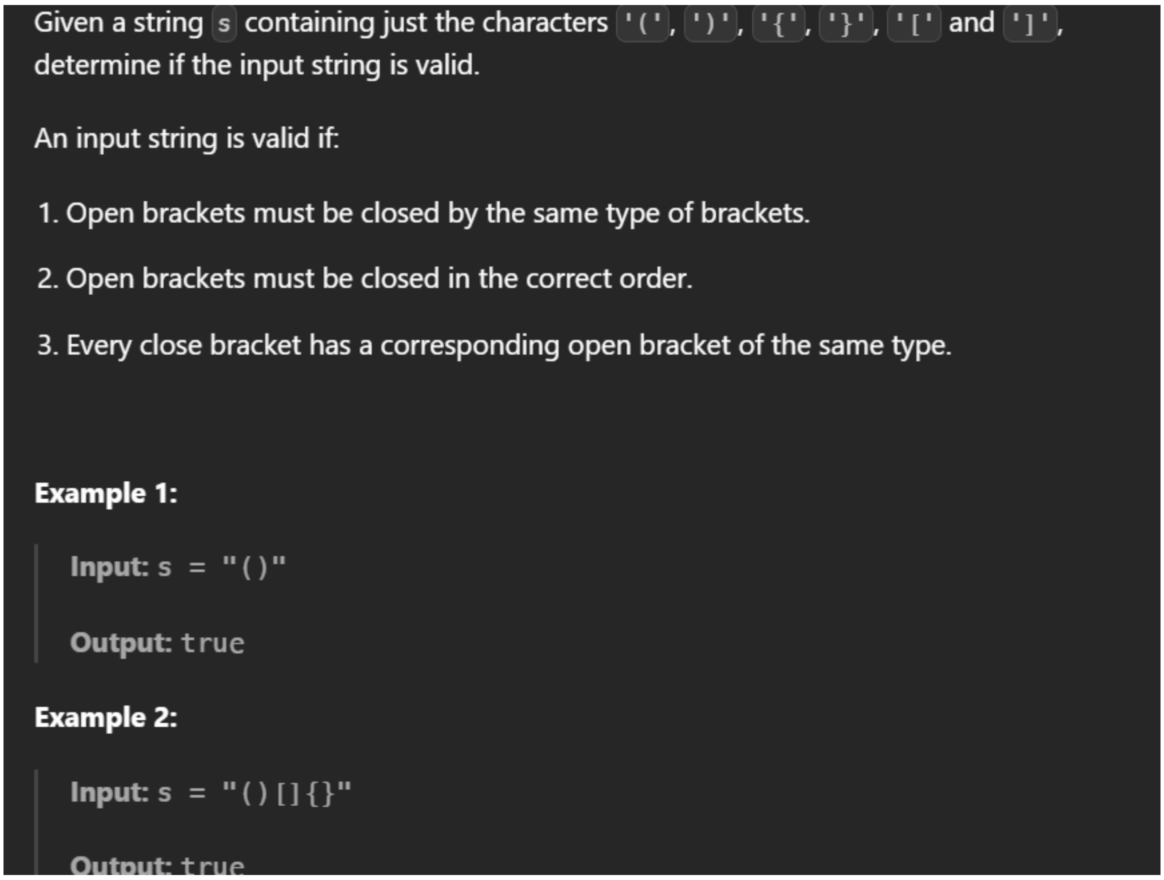
Code:

```
import java.util.*;
class Solution {
    public String minWindow(String s, String t) {
        if (s.length() < t.length()) return "";

        Map<Character, Integer> targetMap = new HashMap<>();
        for (char c : t.toCharArray()) {
            targetMap.put(c, targetMap.getOrDefault(c, 0) + 1);
        }
        int left = 0, right = 0, start = 0, minLength = Integer.MAX_VALUE, matchCount = 0;
        Map<Character, Integer> windowMap = new HashMap<>();
        while (right < s.length()) {
            char cRight = s.charAt(right);
            if (targetMap.containsKey(cRight)) {
                windowMap.put(cRight, windowMap.getOrDefault(cRight, 0) + 1);
                if (windowMap.get(cRight).equals(targetMap.get(cRight))) {
                    matchCount++;
                }
            }
            while (matchCount == targetMap.size()) {
                if (right - left + 1 < minLength) {
                    minLength = right - left + 1;
                    start = left;
                }
                char cLeft = s.charAt(left);
                if (targetMap.containsKey(cLeft)) {
                    if (windowMap.get(cLeft).equals(targetMap.get(cLeft))) {
                        matchCount--;
                    }
                    windowMap.put(cLeft, windowMap.get(cLeft) - 1);
                }
                left++;
            }
            right++;
        }
        return minLength == Integer.MAX_VALUE ? "" : s.substring(start, start + minLength);
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        String s = "ADOBECODEBANC";
        String t = "ABC";
        System.out.println("Output: \"" + solution.minWindow(s, t) + "\"");
    }
}
```

Output: "BANC"

Time Complexity: O(n+m)

10. Valid Parentheses



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

Code:

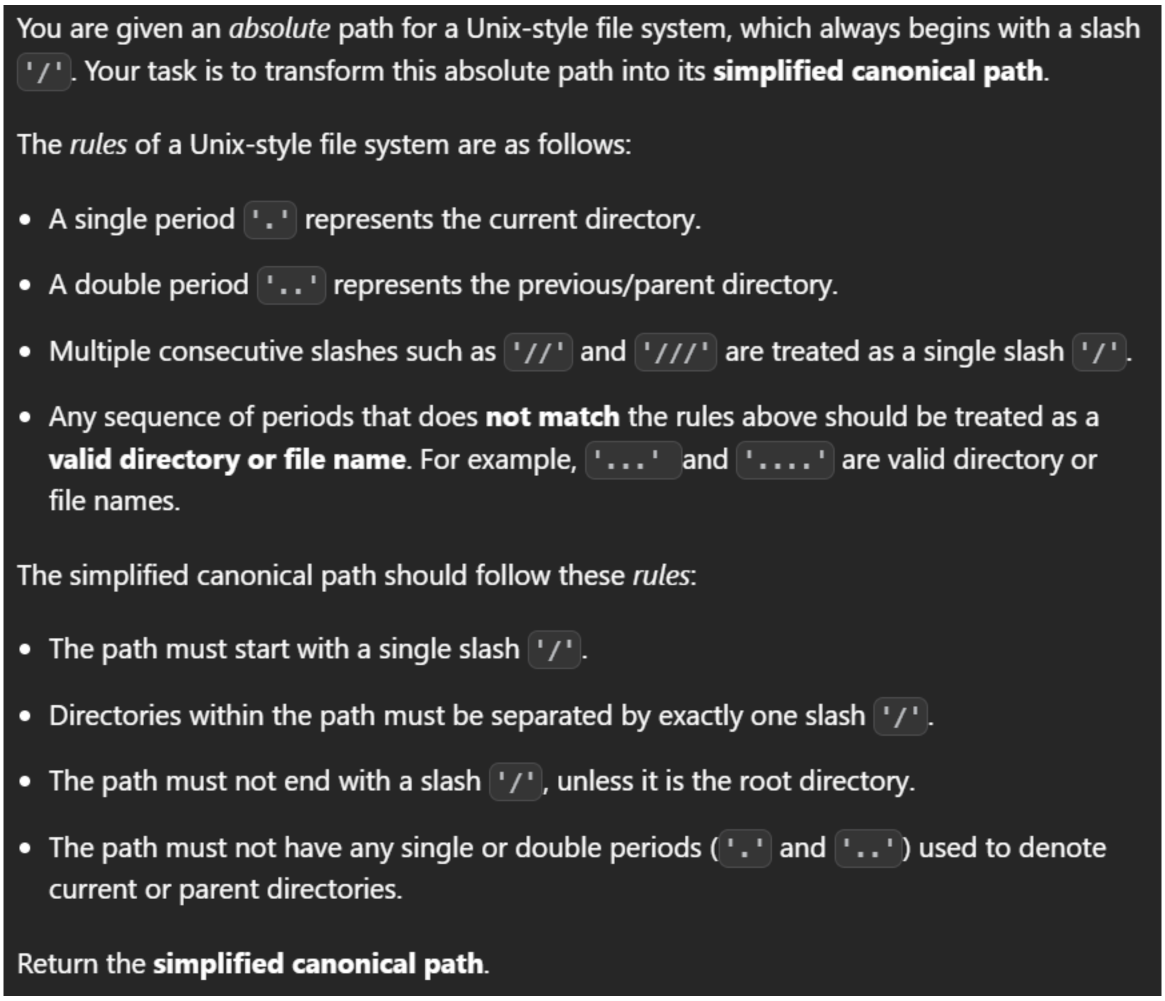
```
import java.util.Stack;


class Parantheses {
   public static boolean isValid(String s) {
      Stack<Character> stack = new Stack<Character>();
      for (char c : s.toCharArray()) {
         if (c == '(')
            stack.push(')');
         else if (c == '{')
            stack.push('}');
         else if (c == '[')
            stack.push(']');
         else if (stack.isEmpty() || stack.pop() != c)
            return false;
      }
      return stack.isEmpty();
   }
```

```
    public static void main(String[] args) {
        String s = "(())";
        boolean result = isValid(s);
        System.out.println(result ? "Balanced" : "Not Balanced");
    }
}
```

Output: Balanced

Time Complexity: O(n)

11. 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**.

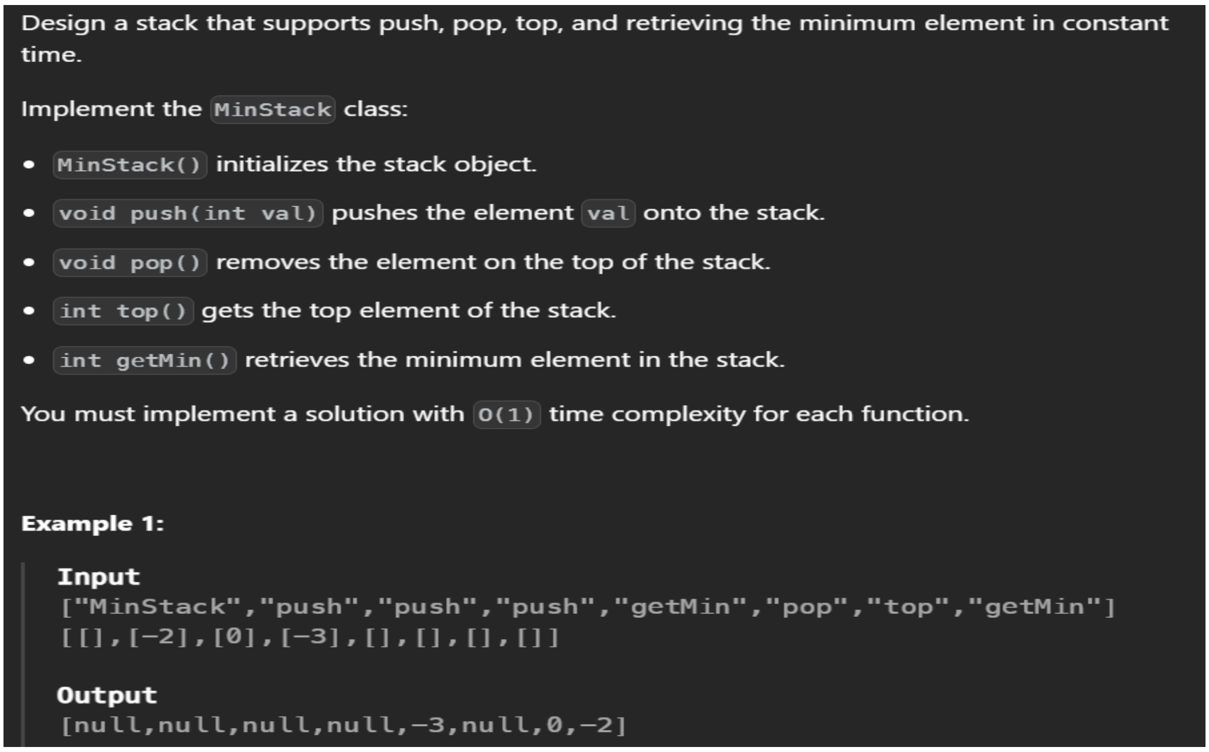
Code:

```
import java.util.*;
class Solution {
    public String simplifyPath(String path) {
        Stack<String> stack = new Stack<>();
        String[] directories = path.split("/");
        for (String dir : directories) {
            if (dir.equals(".") || dir.isEmpty()) {
                continue;
            } else if (dir.equals("..")) {
                if (!stack.isEmpty()) {
                    stack.pop();
                }
            } else {
                stack.push(dir);
            }
        }
        return "/" + String.join("/", stack);
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        String path = "/home/../usr//bin/./test";
        System.out.println("Output: \"" + solution.simplifyPath(path) + "\"");
    }
}
```

Output: "/usr/bin/test"

Time Complexity:O(n)

12. 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]
```

Code:

```
import java.util.*;
class MinStack {
    private List<int[]> st;
    public MinStack() {
        st = new ArrayList<>();
    }
    public void push(int val) {
        int[] top = st.isEmpty() ? new int[]{val, val} : st.get(st.size() - 1);
        int min_val = top[1];
        if (min_val > val) {
            min_val = val;
        }
        st.add(new int[]{val, min_val});
    }
    public void pop() {
        st.remove(st.size() - 1);
    }
    public int top() {
        return st.isEmpty() ? -1 : st.get(st.size() - 1)[0];
    }
    public int getMin() {
        return st.isEmpty() ? -1 : st.get(st.size() - 1)[1];
    }
    public static void main(String[] args) {
        MinStack minStack = new MinStack();
        minStack.push(-2);
        minStack.push(0);
        minStack.push(-3);
        System.out.println("Current Minimum: " + minStack.getMin());
        minStack.pop();
        System.out.println("Top Element: " + minStack.top());
        System.out.println("Current Minimum: " + minStack.getMin());
    }
}
```

Output:

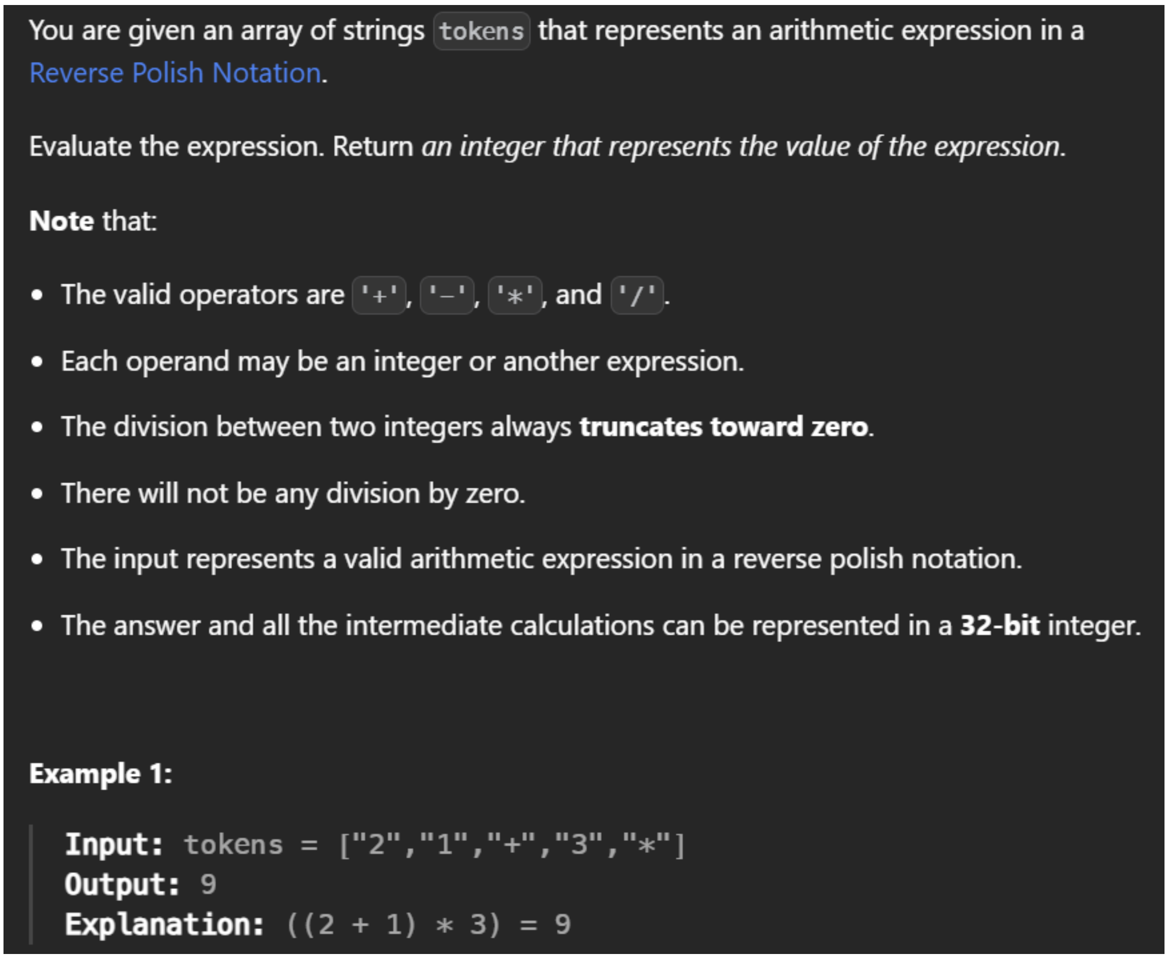
Current Minimum: -3

Top Element: 0

Current Minimum: -2

Time Complexity:O(1)

13. 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
```

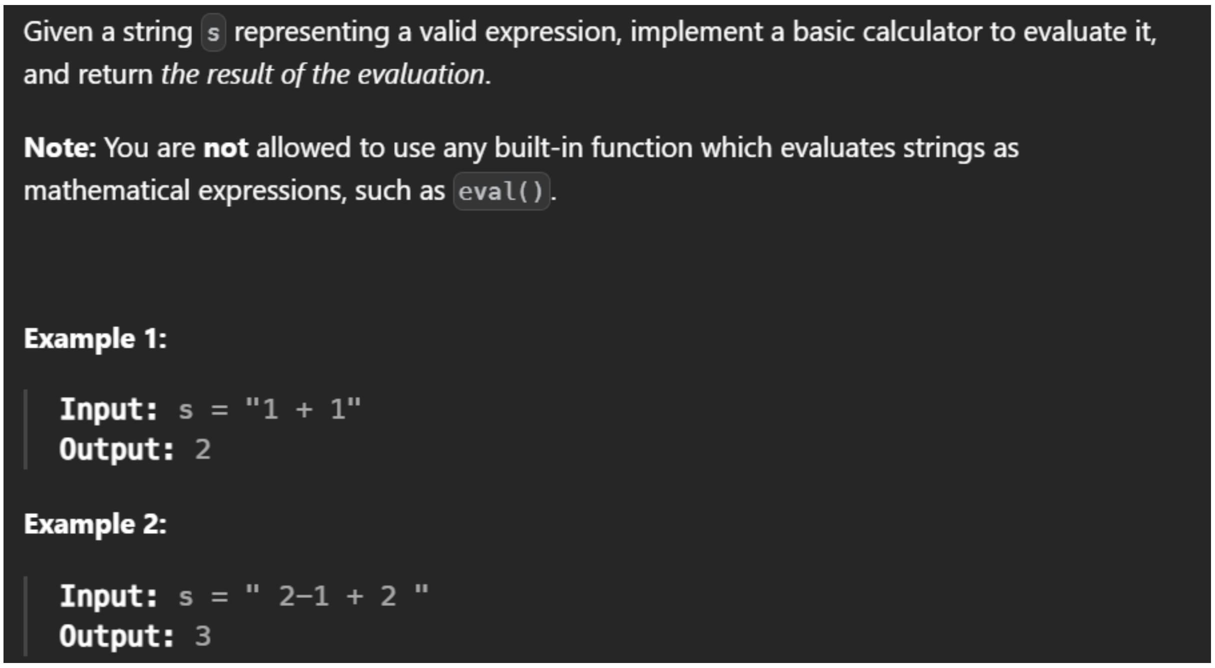
Code:

```
import java.util.*;
class Solution {
    public int evalRPN(String[] tokens) {
        Stack<String> st = new Stack<>();
        for (int i = 0; i < tokens.length; i++) {
            if (!tokens[i].equals("+") && !tokens[i].equals("-") && !tokens[i].equals("*") &&
!tokens[i].equals("/")) {
                st.push(tokens[i]);
            } else {
                int ans = 0;
                int val1 = Integer.parseInt(st.pop());
                int val2 = Integer.parseInt(st.pop());
                if (tokens[i].equals("+"))
                    ans = val1 + val2;
                else if (tokens[i].equals("-"))
                    ans = val2 - val1;
                else if (tokens[i].equals("*"))
                    ans = val1 * val2;
                else if (tokens[i].equals("/"))
                    ans = val2 / val1;
                String temp = Integer.toString(ans);
                st.push(temp);
            }
        }
        return Integer.parseInt(st.peek());
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        String[] tokens = {"2", "1", "+", "3", "*"};
        System.out.println("Solution.evalRPN(tokens));
    }
}
```

Output:9

Time Complexity:O(n)

14. 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
```

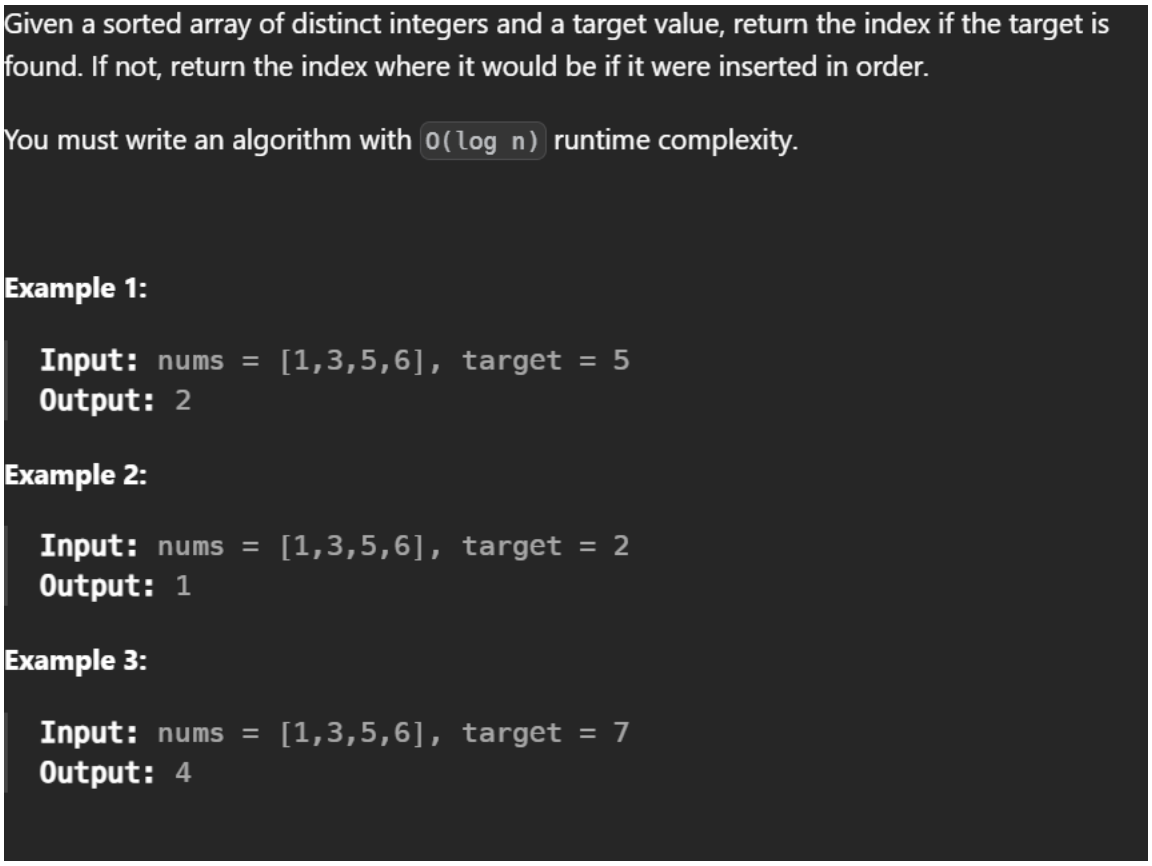
Code:

```java
import java.util.*;
class Solution {
    public int calculate(String s) {
        int number = 0;
        int signValue = 1;
        int result = 0;
        Stack<Integer> operationsStack = new Stack<>();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (Character.isDigit(c)) {
                number = number * 10 + (c - '0');
            } else if (c == '+' || c == '-') {
                result += number * signValue;
                signValue = (c == '-') ? -1 : 1;
                number = 0;
            } else if (c == '(') {
                operationsStack.push(result);
                operationsStack.push(signValue);
                result = 0;
                signValue = 1;
            } else if (c == ')') {
                result += signValue * number;
                result *= operationsStack.pop();
                result += operationsStack.pop();
                number = 0;
            }
        }
        return result + number * signValue;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        String expression = "1 + (2 - (3 + 4))";
        System.out.println("Output: " + solution.calculate(expression));
    }
}
```

Output: -4

Time Complexity:O(n)

15. Search Insert Position



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
```

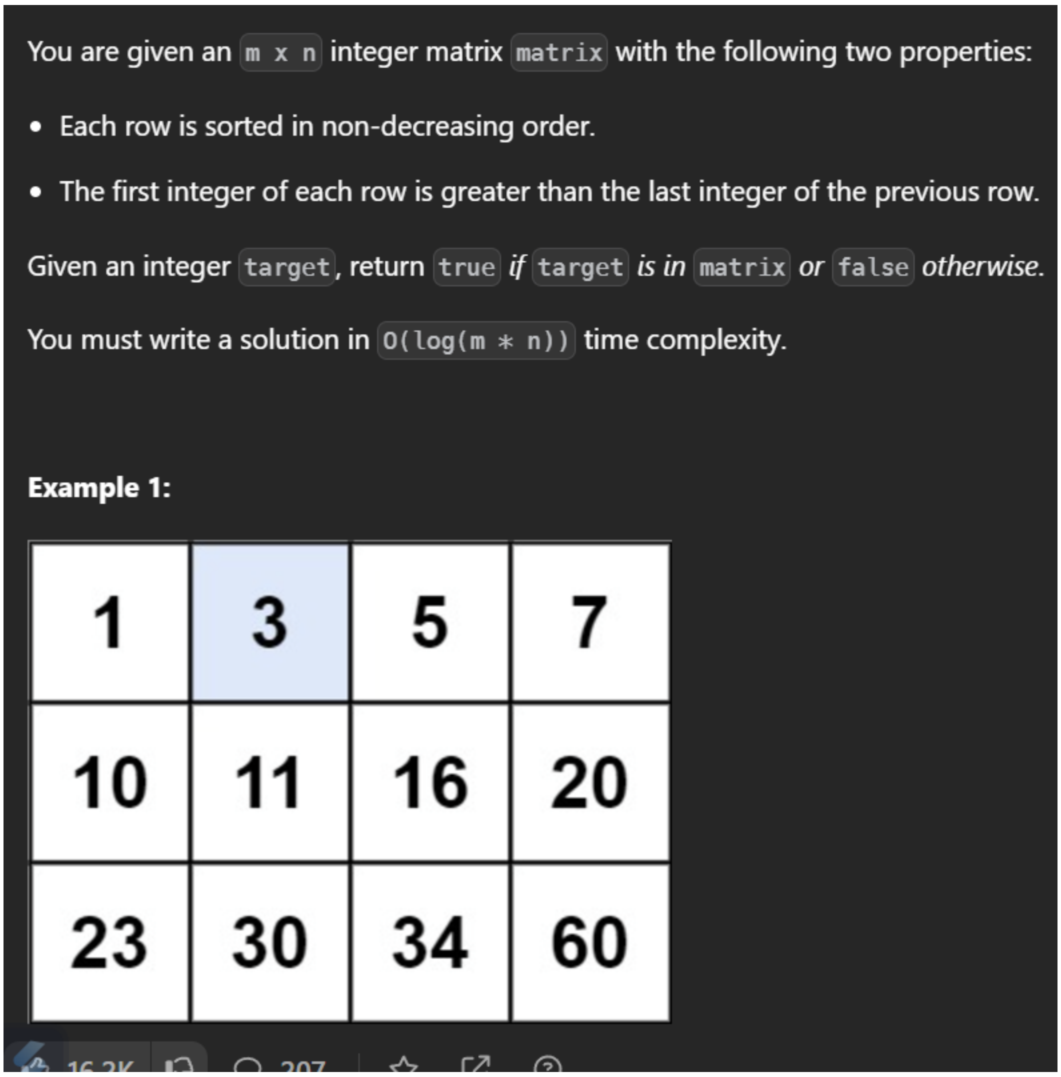
Code:

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        int l = 0;
        int r = nums.length - 1;
        if (target < nums[l]) return 0;
        if (target > nums[r]) return nums.length;
        while (l < r) {
            int val = l + (r - l) / 2;
            if (nums[val] == target) return val;
            if (nums[val] < target) l = val + 1;
            else r = val;
        }
        return l;
    }
    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {1, 3, 5, 6};
        int target = 5;
        System.out.println("Output: " + solution.searchInsert(nums, target));
    }
}
```

Output:2

Time Complexity:O(logn)

16. 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

Code:

```
class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        int top = 0;
        int bot = matrix.length - 1;
        while (top <= bot) {
            int mid = (top + bot) / 2;
            if (target >= matrix[mid][0] && target <= matrix[mid][matrix[mid].length - 1]) {
```

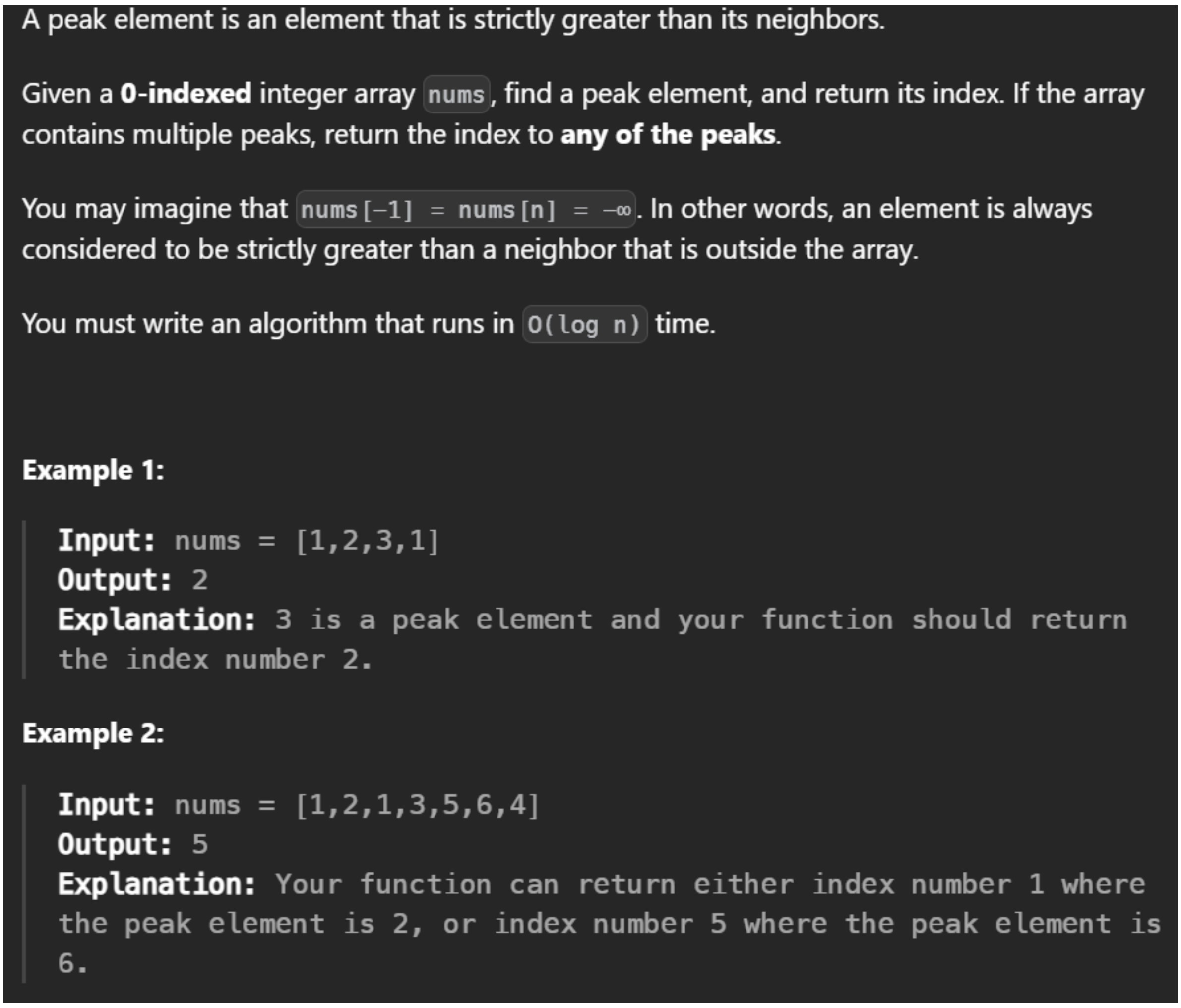
```
            top = mid;
            break;
        } else if (matrix[mid][0] > target) {
            bot = mid - 1;
        } else {
            top = mid + 1;
        }
    }

    if (top > bot) return false;
    int row = top;
    int left = 0;
    int right = matrix[row].length - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        if (matrix[row][mid] == target) {
            return true;
        } else if (matrix[row][mid] > target) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return false;
}
public static void main(String[] args) {
    Solution solution = new Solution();
    int[][] matrix = {
        {1, 3, 5, 7},
        {10, 11, 16, 20},
        {23, 30, 34, 60}
    };
    int target = 3;
    System.out.println("Output: " + solution.searchMatrix(matrix, target));
}
```

}

Output: true

Time Complexity:O(logn+logm)

17. 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.
```

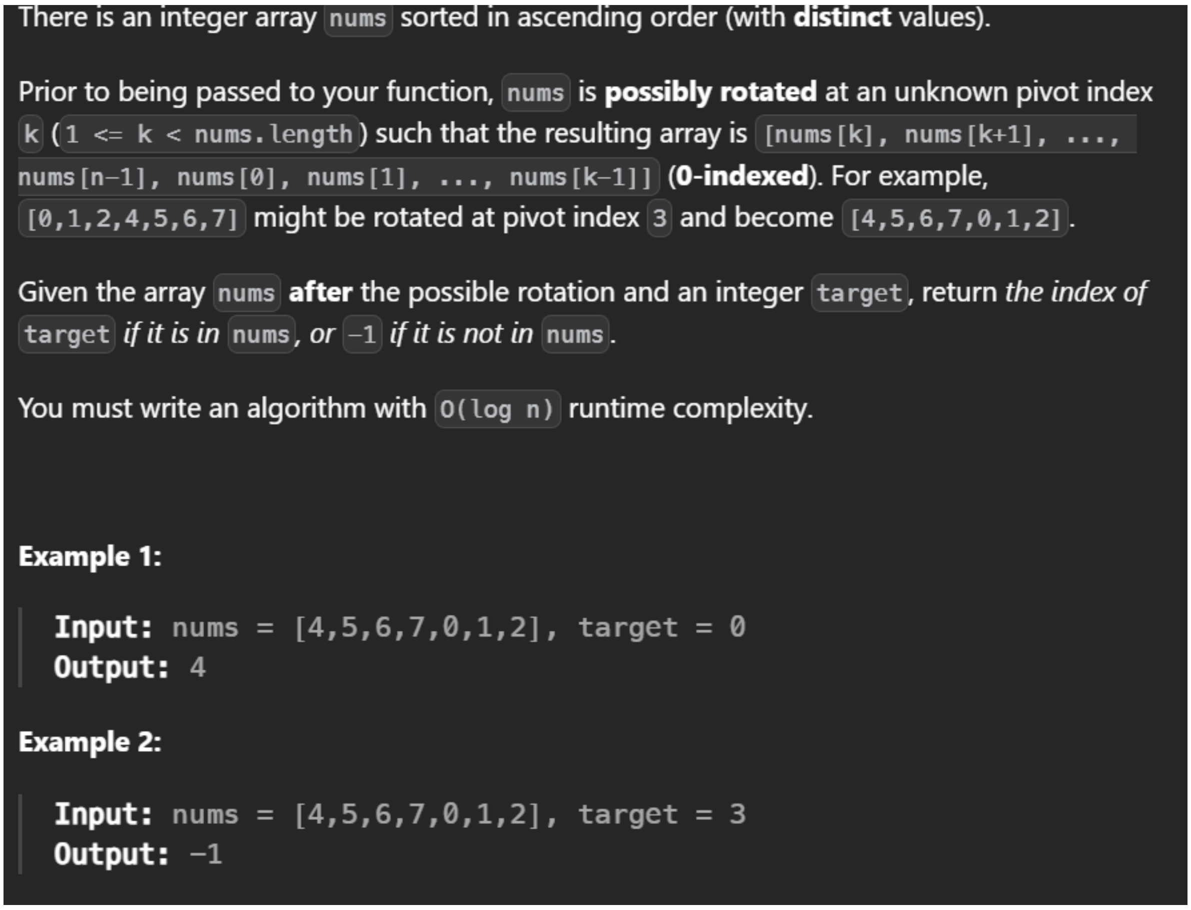
Code:

```
class Solution {
    public int findPeakElement(int[] nums) {
        int left = 0;
        int right = nums.length - 1;
        while (left < right) {
            int mid = (left + right) / 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();

        // Test case
        int[] nums = {1, 2, 3, 1};
        int peakIndex = solution.findPeakElement(nums);
        System.out.println( nums[peakIndex]);
    }
}
```

Output: 3

Time Complexity:O(log n)

18. 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
```

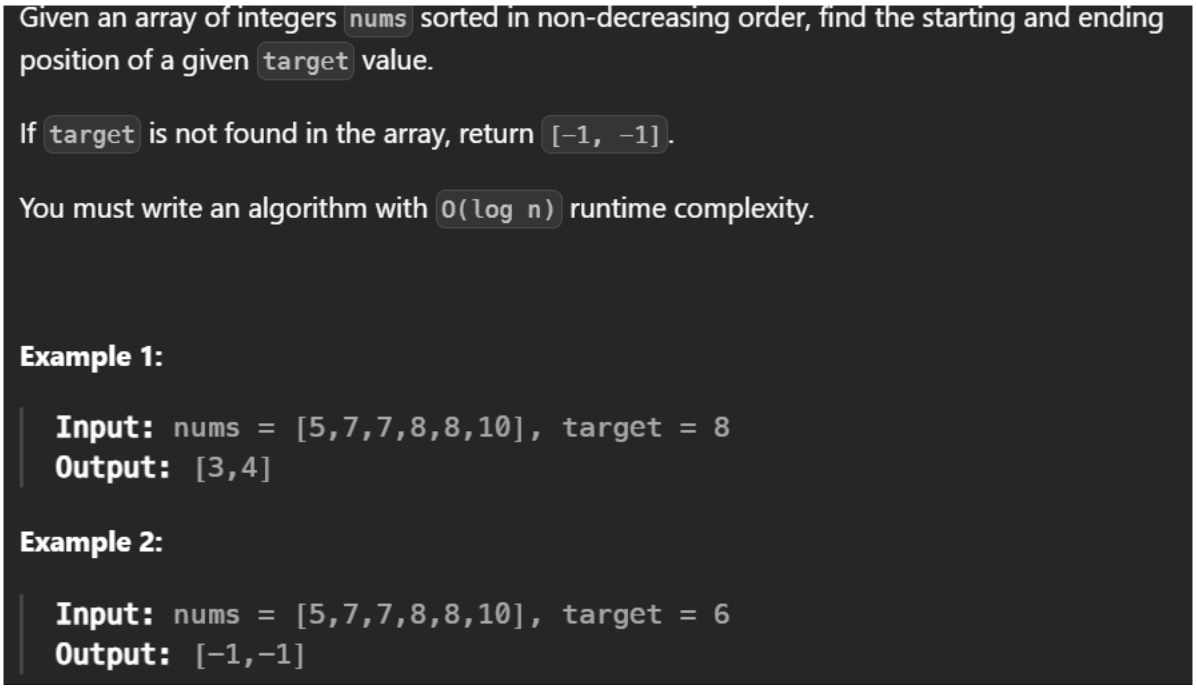
Code:

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

        // Test case
        int[] nums = {4, 5, 6, 7, 0, 1, 2};
        int target = 0;
        int index = solution.search(nums, target);
        System.out.println("Index of target: " + index);
    }
}
```

Output: Index of target: 4

Time Complexity:O(n)

19. Find First and last position



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]
```

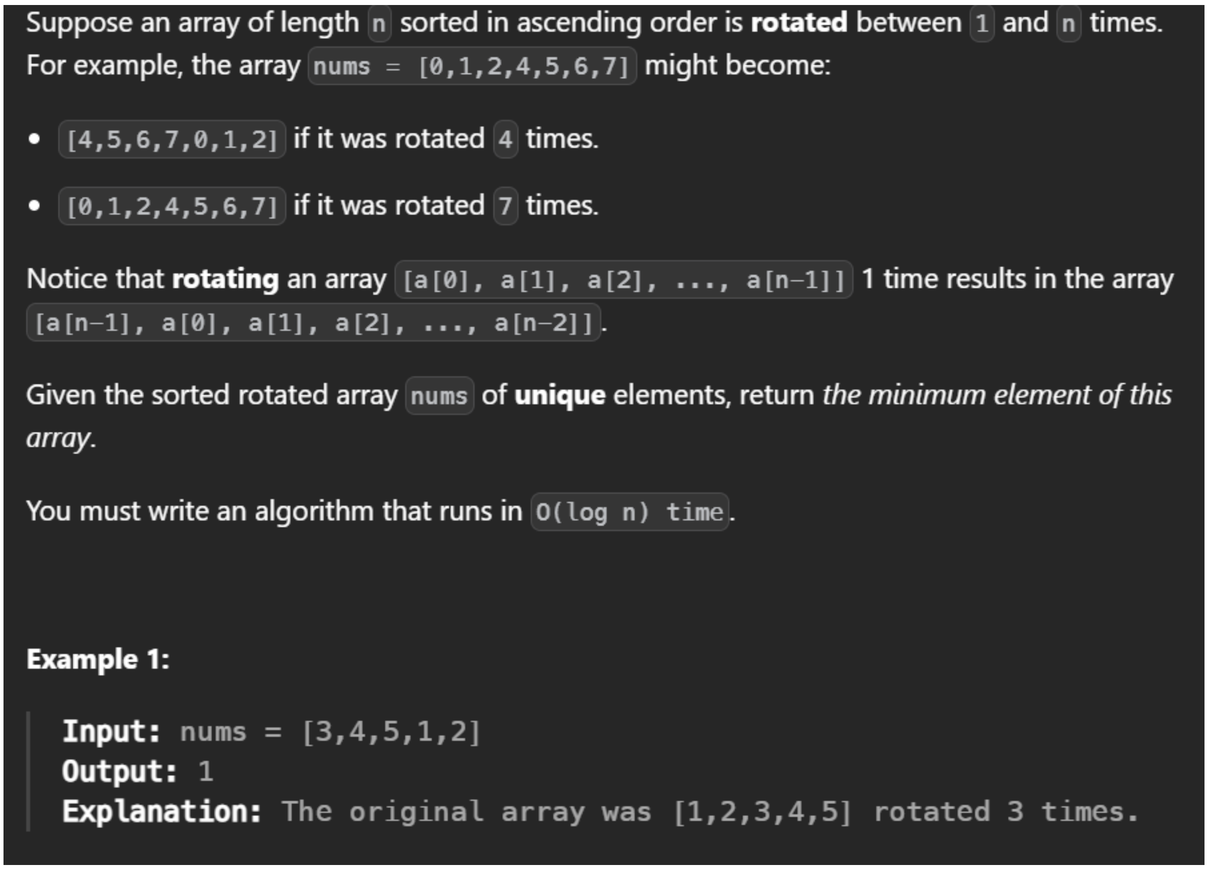
Code:

```
class Solution {
    public int[] searchRange(int[] nums, int target) {
        int[] result = {-1, -1};
        int left = binarySearch(nums, target, true);
        int right = binarySearch(nums, target, false);
        result[0] = left;
        result[1] = right;
        return result;
    }
    private int binarySearch(int[] nums, int target, boolean isSearchingLeft) {
        int left = 0;
        int right = nums.length - 1;
        int idx = -1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > target) {
                right = mid - 1;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                idx = mid;
                if (isSearchingLeft) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }
        }

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

        // Test case
        int[] nums = {5, 7, 7, 8, 8, 10};
        int target = 8;
        int[] result = solution.searchRange(nums, target);
        System.out.println("Range: [" + result[0] + ", " + result[1] + "]");
    }
}
```

Output: Range: [3, 4]

Time Complexity:O(log n)

20. Find Minimum in Rotated 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.
```

Code:

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

            if (nums[mid] < nums[right]) {
```

```
            right = mid - 1;
          }
          if (nums[mid] > nums[left]) {
            left = mid + 1;
          }
        }
        return nums[left];
      }
      public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {3, 4, 5, 1, 2};
        int min = solution.findMin(nums);
        System.out.println("Minimum element: " + min);
      }
    }
```

Output: Minimum element: 1

Time Complexity:O(log n)