**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 \* 105
* s consists only of printable ASCII characters.

Program:

class Solution {

public:

    bool isPalindrome(string s) {

        string ans="";

        int a=s.size();

        for(int i=0;i<a;i++)

        {

            if(isalnum(s[i]))

            {

                ans+=tolower(s[i]);

            }

        }

        int start=0,end=ans.size()-1;

        bool palin=true;

        while(start<=end)

        {

            if(ans[start]!=ans[end])

            {

                palin=false;

                break;

            }

            ++start;

            --end;

        }

        return palin;

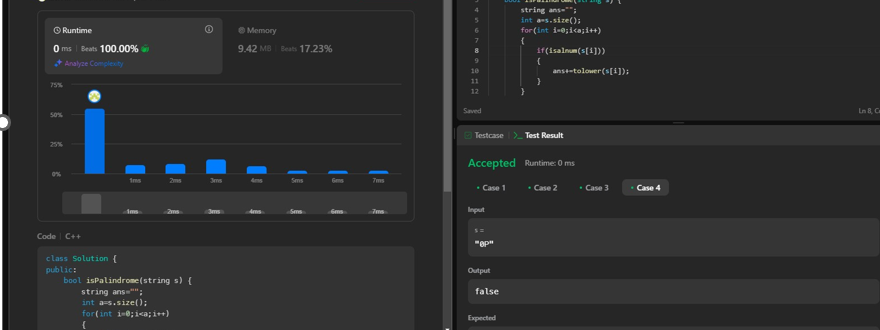
    }

};

Time Complexity: O(a)

Space Complexity: O(a)

Output:



Program: - space efficient

class Solution {

public:

    bool isPalindrome(string s) {

       int start=0;

       int end=s.size()-1;

       while(start<=end){

           if(!isalnum(s[start])){start++; continue;}

           if(!isalnum(s[end])){end--;continue;}

           if(tolower(s[start])!=tolower(s[end]))return false;

           else{

               start++;

               end--;

           }

       }

       return true;

}

};

Time Complexity: O(n)

Space complexity : O(1)

**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 <= 104
* s and t consist only of lowercase English letters.

**Follow up:** Suppose there are lots of incoming s, say s1, s2, ..., sk where k >= 109, and you want to check one by one to see if t has its subsequence. In this scenario, how would you change your code?

Program:

class Solution {

public:

    bool isSubsequence(string s, string t) {

        int a=s.size(),b=t.size();

        int ctr=0;

        int i=0;

        int j=0;

        while(i<a && j<b)

        {

            if(s[i]==t[j])

            {

                ++i;

                ++ctr;

            }

            ++j;

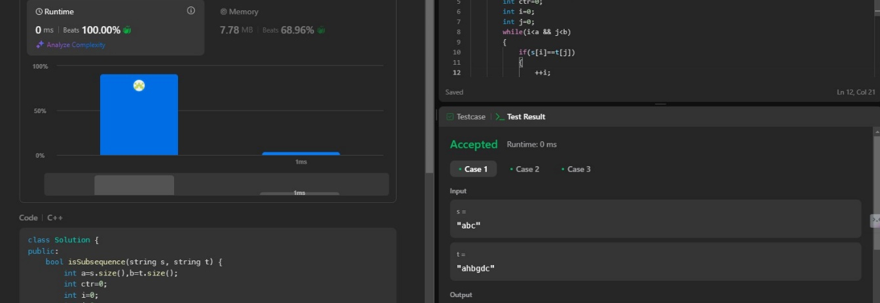
        }

        return ctr==a;

    }

};

Output:



Time Complexity: O(n)

Space Complexity: O(1)

Answer for Follow up: - GPT answer

class Solution {

public:

bool isSubsequence(string s, string t) {

// Preprocess `t` to build a map of character positions

unordered\_map<char, vector<int>> charPosMap;

for (int i = 0; i < t.size(); ++i) {

charPosMap[t[i]].push\_back(i);

}

// Function to check if `s` is a subsequence of `t`

auto isSubsequenceHelper = [&](const string& s) -> bool {

int prevIndex = -1; // Track the previous matched index in `t`

for (char c : s) {

if (charPosMap.find(c) == charPosMap.end()) {

return false; // `c` is not in `t`

}

// Find the first position of `c` > prevIndex using binary search

auto it = upper\_bound(charPosMap[c].begin(), charPosMap[c].end(), prevIndex);

if (it == charPosMap[c].end()) {

return false; // No valid position found

}

prevIndex = \*it; // Update the previous index

}

return true;

};

// Return result for the current string `s`

return isSubsequenceHelper(s);

}

};

Time Complexity: O(n + m log n)

Space Complexity: O(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[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].

**Example 2:**

**Input:** numbers = [2,3,4], target = 6

**Output:** [1,3]

**Explanation:** The sum of 2 and 4 is 6. Therefore index1 = 1, index2 = 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 index1 = 1, index2 = 2. We return [1, 2].

**Constraints:**

* 2 <= numbers.length <= 3 \* 104
* -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**.

Program:

class Solution {

public:

    vector<int> twoSum(vector<int>& nums, int target) {

        //int n=nums.size();

        int start=0,end=nums.size()-1;

        while(start<end)

        {

            int sum=nums[start]+nums[end];

            if(sum==target)

            {

                return {start+1,end+1};

            }

            else if(sum<target)

            {

                ++start;

            }

            else

            {

                --end;

            }

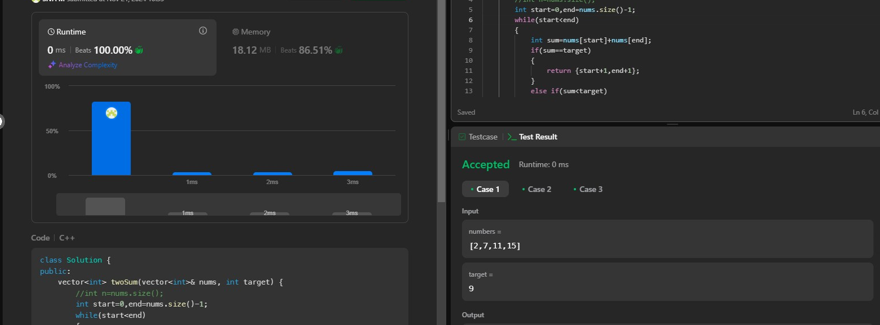
        }

        return {};

    }

};

Output:



Space Complexity: O(1)

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 ith 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 <= 105
* 0 <= height[i] <= 104

Program:

class Solution {

public:

    int maxArea(vector<int>& height) {

        int left=0;

        int right=height.size()-1;

        int ans=0;

        while(left<right)

        {

            int len=right-left;

            int bre=min(height[left],height[right]);

            int area=len\*bre;

            ans=max(ans,area);

            if(height[left]<height[right])

            {

                ++left;

            }

            else

            {

                --right;

            }

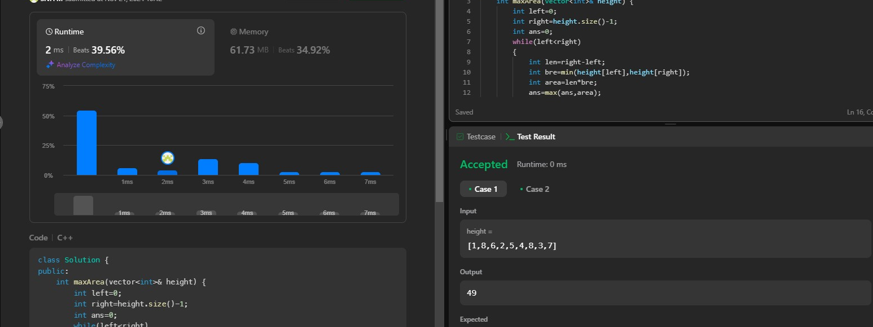
        }

        return ans;

    }

};

Output:



Time Complexity:O(N)

Space Complexity:O(1)

**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
* -105 <= nums[i] <= 105

Program:

class Solution {

public:

    vector<vector<int>> threeSum(vector<int>& nums) {

        int n=nums.size();

        sort(nums.begin(),nums.end());

        vector<vector<int>> ans;

        for(int i=0;i<n;i++)

        {

            if(i>0 && nums[i]==nums[i-1]) continue;

            int j=i+1,k=n-1;

            while(j<k)

            {

                int total=nums[i]+nums[j]+nums[k];

                if(total>0) --k;

                else if(total<0) ++j;

                else

                {

                    ans.push\_back({nums[i],nums[j],nums[k]});

                    ++j;

                    while(nums[j]==nums[j-1] && j<k){++j;}

                }

            }

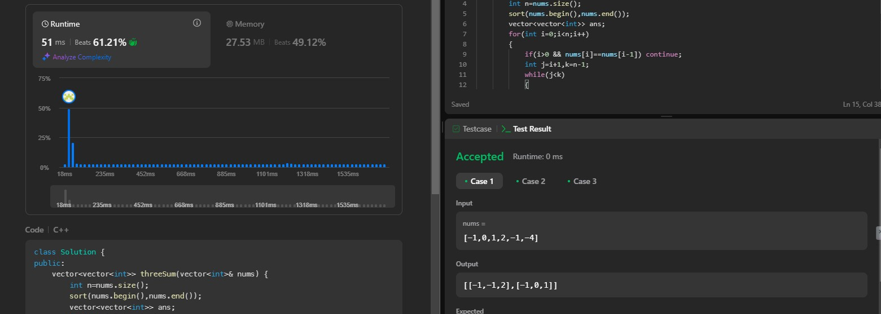
        }

        return ans;

    }

};

Output:



Time Complexity:O(n2)

Space Complexity: O(n)

**Stack**

**1.Valid Parenthesis**

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 <= 104
* s consists of parentheses only '()[]{}'.

Program:

class Solution {

public:

    bool isValid(string s) {

        stack<char> a;

        int n=s.size();

        for(int i=0;i<n;i++)

        {

            if(s[i]=='(' || s[i]=='{' || s[i]=='[')

            {

                a.push(s[i]);

            }

            else if(!a.empty() && s[i]==')' && a.top()=='(')

            {

                a.pop();

            }

            else if(!a.empty() && s[i]==']' && a.top()=='[')

            {

                a.pop();

            }

            else if(!a.empty() && s[i]=='}' && a.top()=='{')

            {

                a.pop();

            }

            else

            {

                return false;

            }

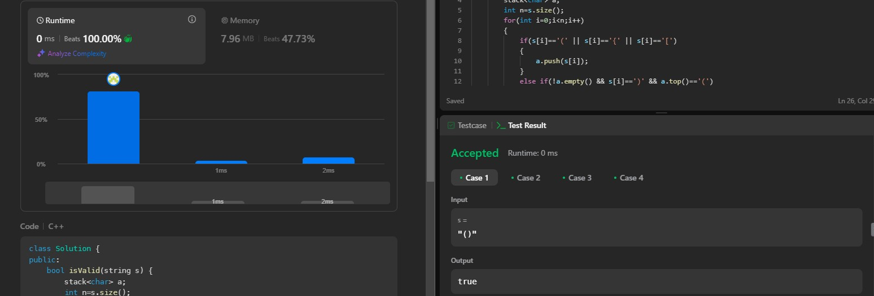
        }

        return a.empty();

    }

};

Output:



Time Complexity:O(n)

Space 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**.

**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.

Program:

class Solution {

public:

    vector<string> split(const string& str, char delimiter) {

        vector<string> result;

        stringstream ss(str);

        string token;

        while (getline(ss, token, delimiter))

        {

            result.push\_back(token);

        }

        return result;

    }

    string simplifyPath(string path) {

        vector<string> ans=split(path,'/');

        stack <string> req;

        for (int i = 0; i < ans.size(); ++i) {

            if (ans[i] != "." && ans[i] != ".." && !ans[i].empty()) req.push(ans[i]);

            else if (ans[i] == ".." && !req.empty()) req.pop();

        }

        string res = "";

        while (!req.empty()) {

            res ="/"+ req.top() + res;

            req.pop();

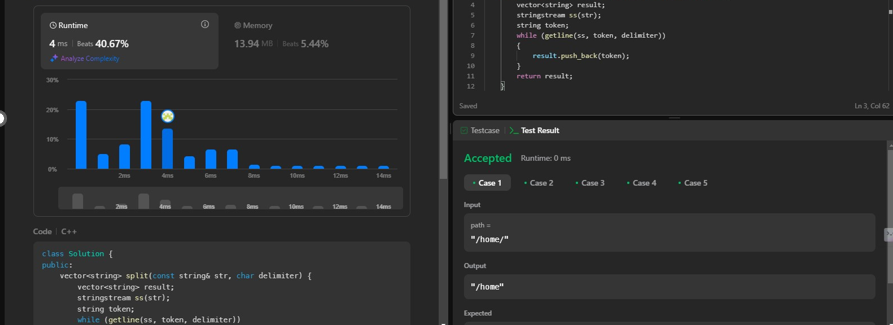
        }

        return res.empty() ? "/" : res;

    }

};

Output:



Time Complexity: O(n)

Space 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]

**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:**

* -231 <= val <= 231 - 1
* Methods pop, top and getMin operations will always be called on **non-empty** stacks.
* At most 3 \* 104 calls will be made to push, pop, top, and getMin.

Program:

class MinStack {

    vector<int> st;

    vector<int> ms;

public:

    MinStack() {

    }

    void push(int val) {

        if(ms.empty() || st[ms.back()]>=val)

        {

            ms.push\_back(st.size());

        }

        st.push\_back(val);

    }

    void pop() {

        if(st.back()==st[ms.back()])

        {

            ms.pop\_back();

        }

        st.pop\_back();

    }

    int top() {

        return st.back();

    }

    int getMin() {

        return st[ms.back()];

    }

};

/\*\*

 \* Your MinStack object will be instantiated and called as such:

 \* MinStack\* obj = new MinStack();

 \* obj->push(val);

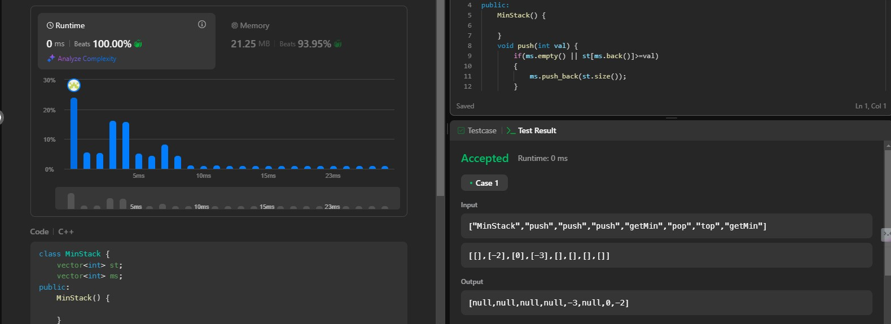
 \* obj->pop();

 \* int param\_3 = obj->top();

 \* int param\_4 = obj->getMin();

 \*/

Output:



Time Complexity : (push,pop,top,getMin) 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](http://en.wikipedia.org/wiki/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 <= 104
* tokens[i] is either an operator: "+", "-", "\*", or "/", or an integer in the range [-200, 200].

Program:

class Solution {

    #include<cctype>

public:

    int evalRPN(vector<string>& tokens) {

        stack<int> ans;

        int n=tokens.size();

        for(int i=0;i<n;i++)

        {

            if(tokens[i]=="+" || tokens[i]=="-"||tokens[i]=="\*"||tokens[i]=="/")

            {

                int x=ans.top();

                ans.pop();

                int y=ans.top();

                ans.pop();

                if(tokens[i]=="+") ans.push(x+y);

                else if(tokens[i]=="-") ans.push(y-x);

                else if(tokens[i]=="\*") ans.push(x\*y);

                else ans.push((int)(y/x));

            }

            else

            {

                ans.push(stoi(tokens[i]));

            }

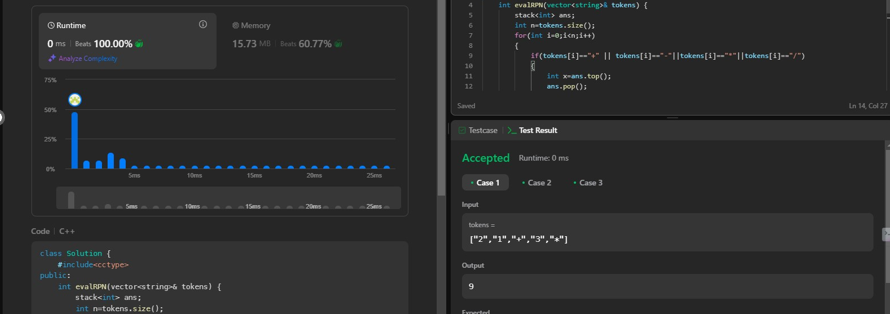
        }

        return ans.top();

    }

};

Output:



Time Complexity: O(n)

Space Complexity: O(n)

**4.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 \* 105
* 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.

Program:

class Solution {

public:

    int calculate(string s) {

        int sum = 0;

        int sign = 1;

        stack<pair<int,int>> st;

        for(int i=0; i<s.size();i++){

            if(isdigit(s[i])){

                int num = 0;

                while(i<s.size() && isdigit(s[i])){

                    num = num \* 10 + (s[i] - '0');

                    i++;

                }

                i--;

                sum += num \* sign;

                sign = 1;

            }

            else if(s[i] == '('){

                st.push({sum, sign});

                sum = 0;

                sign = 1;

            }

            else if(s[i] == ')'){

                sum = st.top().first + (st.top().second \* sum);

                st.pop();

            }

            else if(s[i] == '-'){

                sign = -1 \* sign;

            }

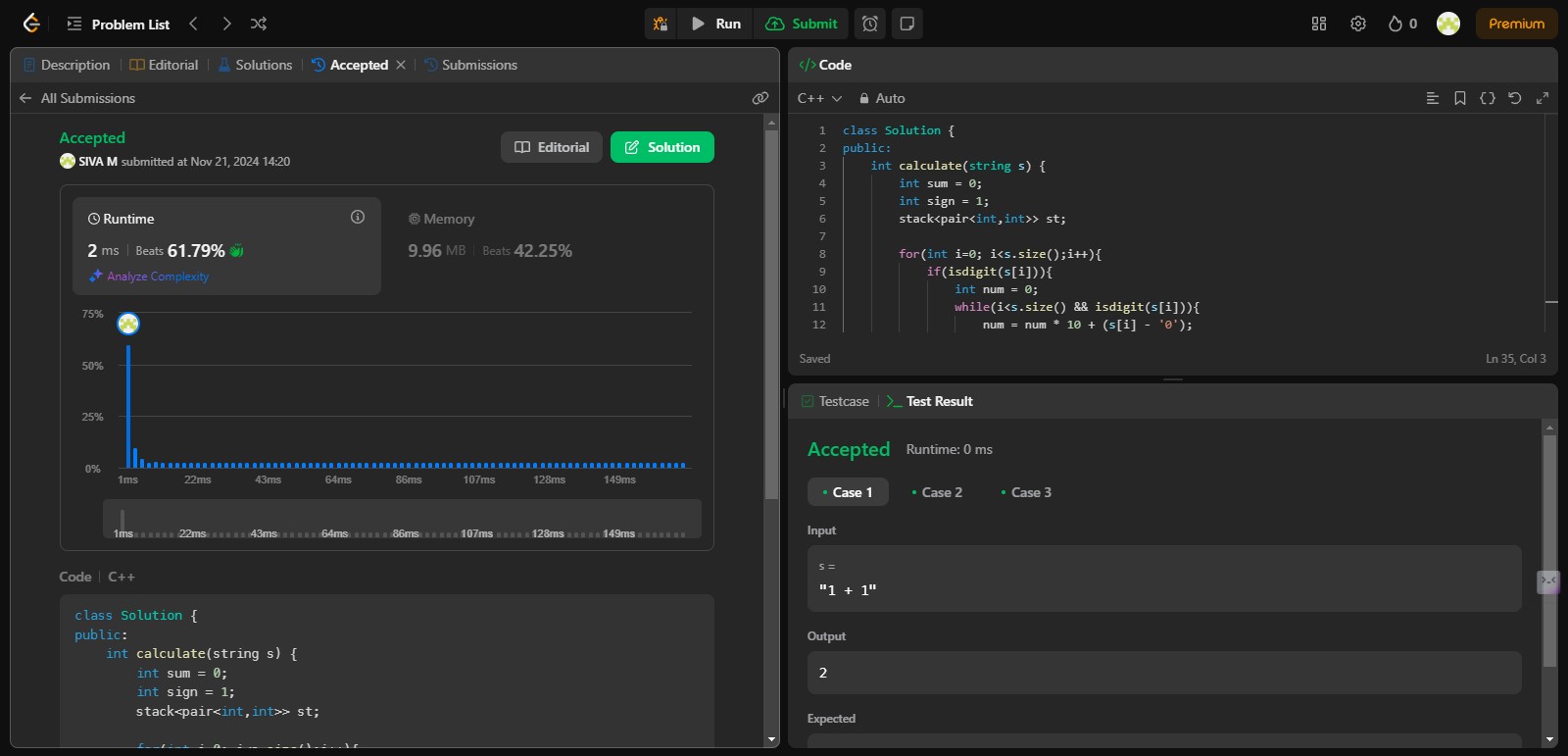
        }

        return sum;

    }

};

Output:



Time Complexity:O(n)

Space Complexity: O(n)

**Binary Search**

**1.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

**Constraints:**

* 1 <= nums.length <= 104
* -104 <= nums[i] <= 104
* nums contains **distinct** values sorted in **ascending** order.
* -104 <= target <= 104

Program:

class Solution {

public:

    int searchInsert(vector<int>& nums, int target) {

        int low=0;

        int high=nums.size()-1;

        while(low<=high){

            int mid=(low+high)/2;

            if(nums[mid]==target)

            {

                return mid;

            }

            if(nums[mid]<target)

            {

                low=mid+1;

            }

            else

            {

                high=mid-1;

            }

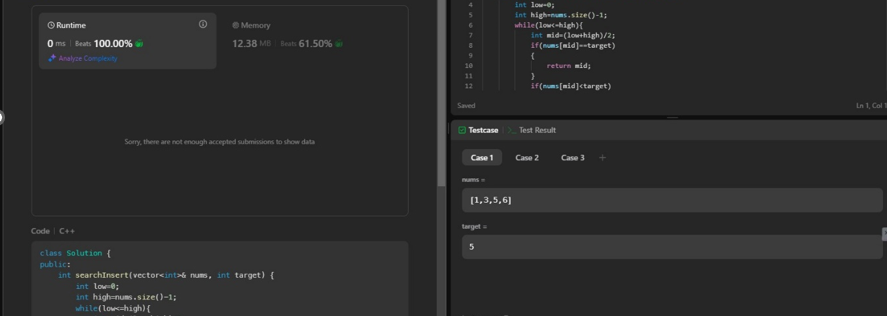
        }

        return low;

    }

};

Output:



Time Complexity: O(log n)

Space Complexity : O(1)

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:**



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

**Output:** true

**Example 2:**



**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
* -104 <= matrix[i][j], target <= 104

Program:

class Solution {

public:

    bool searchMatrix(vector<vector<int>>& matrix, int target) {

        int n=matrix.size();

        int m=matrix[0].size();

        if(m==0 || n==0) return false;

        for(int i=0;i<n;i++)

        {

            int low=0;

            int high=m-1;

            if(target<matrix[i][0] && target>matrix[i][m-1]) continue;

            while(low<=high)

            {

                int mid=(low+high)/2;

                if(matrix[i][mid]==target)

                {

                    return true;

                }

                if(matrix[i][mid]<target)

                {

                    low=mid+1;

                }

                else

                {

                    high=mid-1;

                }

            }

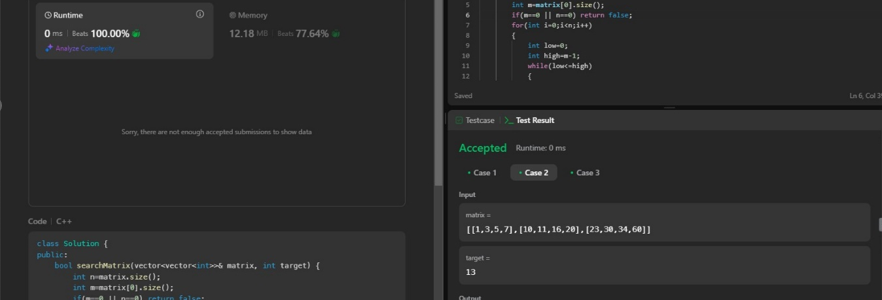
        }

        return false;

    }

};

Output:

  
Time Complexity: O(n log m)

**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
* -231 <= nums[i] <= 231 - 1
* nums[i] != nums[i + 1] for all valid i.

Program:

class Solution {

public:

    int findPeakElement(vector<int>& nums) {

        int left=0;

        int right=nums.size()-1;

        while(left<right)

        {

            int mid=(left+right)/2;

            if((mid==0 || nums[mid]>nums[mid-1])&&(mid==nums.size()-1 || nums[mid]>nums[mid+1]))

            {

                return mid;

            }

            else if(nums[mid]<nums[mid+1])

            {

                left=mid+1;

            }

            else

            {

                right=mid;

            }

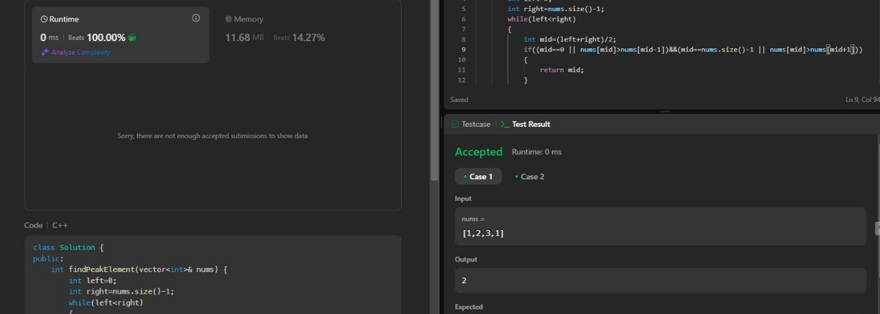
        }

        return left;

    }

};

Output:



Time Complexity: O(log N)

Space Complexity: O(1)

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
* -104 <= nums[i] <= 104
* All values of nums are **unique**.
* nums is an ascending array that is possibly rotated.
* -104 <= target <= 104

Program:

class Solution {

public:

    int search(vector<int>& nums, int target) {

        int low=0;

        int n=nums.size();

        int high=n-1;

        while(low<=high){

            int mid=low+(high-low)/2;

            if(nums[mid]==target)

            {

                return mid;

            }

            if(nums[low]<=nums[mid])

            {

                if(nums[low]<=target && target<nums[mid])

                {

                    high=mid-1;

                }

                else

                {

                    low=mid+1;

                }

            }

            else

            {

                if(nums[mid]<target && target<=nums[high])

                {

                    low=mid+1;

                }

                else

                {

                    high=mid-1;

                }

            }

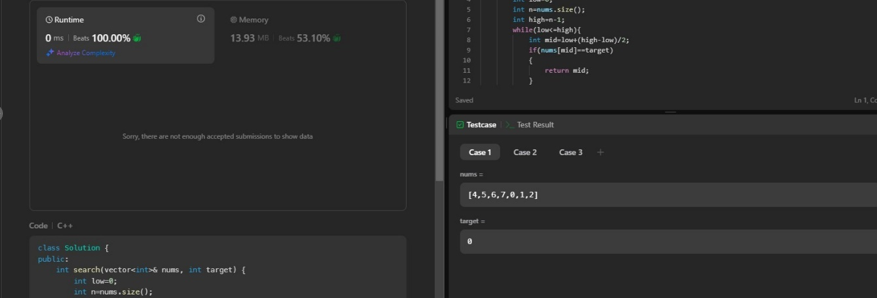
        }

        return -1;

    }

};

Output:



Time Complexity: O(log n)

4.Find first and last position of an element in an 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 <= 105
* -109 <= nums[i] <= 109
* nums is a non-decreasing array.
* -109 <= target <= 109

Program:

class Solution {

public:

    vector<int> searchRange(vector<int>& nums, int target) {

        vector<int> result={-1,-1};

        int left=binarysearch(nums,target,true);

        int right=binarysearch(nums,target,false);

        result[0]=left;

        result[1]=right;

        return result;

    }

    int binarysearch(vector<int>&nums,int target,bool issearchleft)

    {

        int left=0;

        int right=nums.size()-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(issearchleft)

                {

                    right=mid-1;

                }

                else

                {

                    left=mid+1;

                }

            }

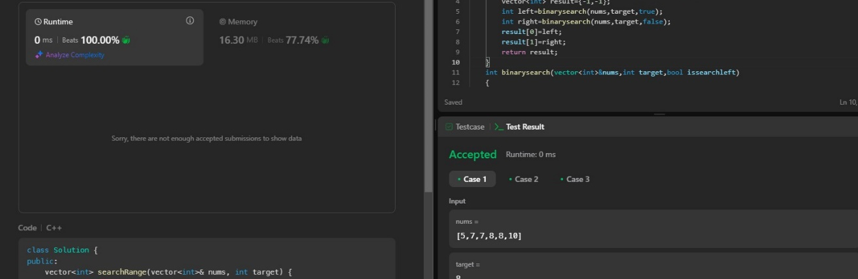
        }

        return idx;

    }

};

Output:



Time complexity:O(log n)

Space Complexity:O(1)

5.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.

Program:

class Solution {

public:

    int findMin(vector<int>& arr) {

        int left=0;

        int right=arr.size()-1;

        while(left<right)

        {

            int mid=left+(right-left)/2;

            if(arr[mid]>arr[right]) left=mid+1;

            else right=mid;

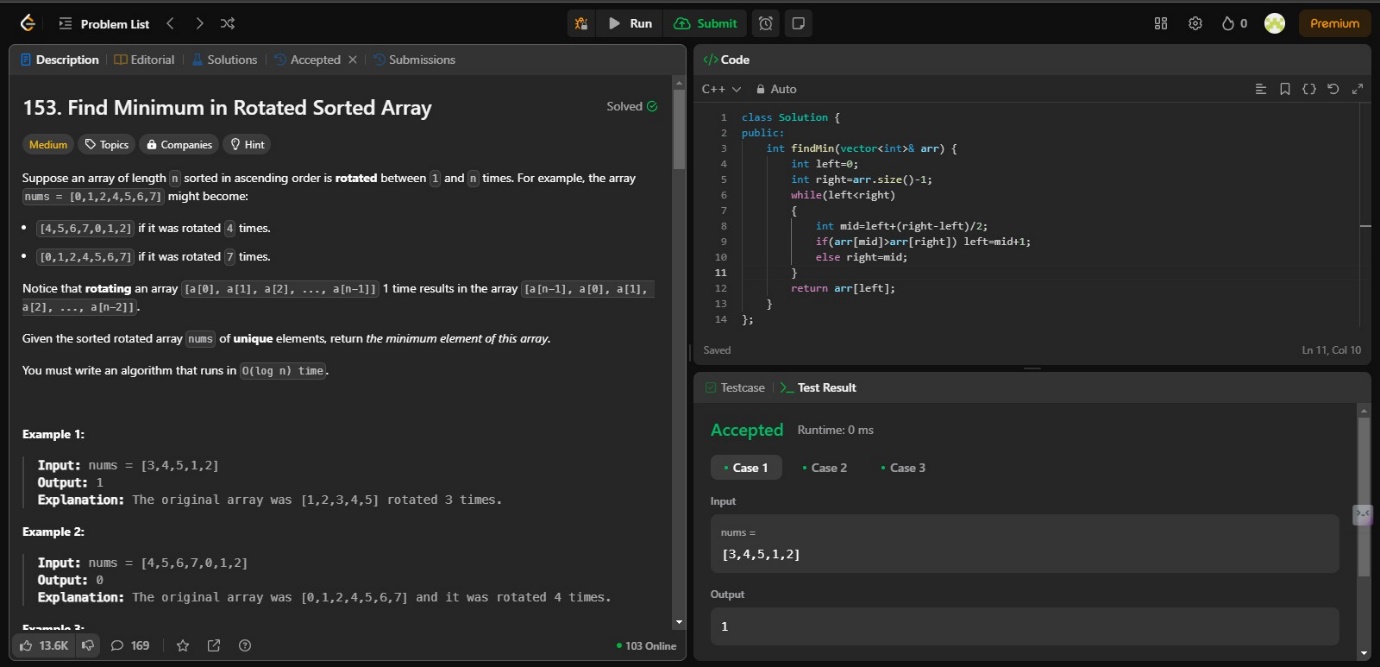
        }

        return arr[left];

    }

};

Output:



Time Complexity😊: O(log n)

Space Complexity : O(1)

**Sliding Window**

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 <= 109
* 1 <= nums.length <= 105
* 1 <= nums[i] <= 104

**Follow up:** If you have figured out the O(n) solution, try coding another solution of which the time complexity is O(n log(n)).

Program:

class Solution {

public:

    int minSubArrayLen(int target, vector<int>& nums) {

        int n=nums.size();

        int left=0;

        int right=0;

        int sum=0;

        int len=INT\_MAX;

        while(right<n)

        {

            sum+=nums[right];

            while(sum>=target)

            {

                len=min(len,right-left+1);

                sum-=nums[left];

                ++left;

            }

            ++right;

        }

        return len==INT\_MAX? 0:len;

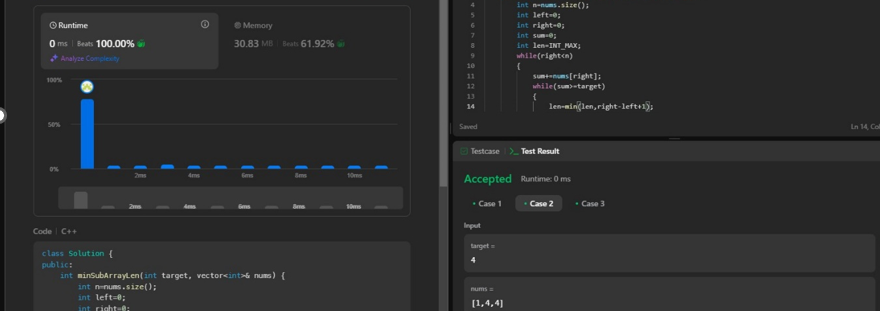
    }

};

Time Complexity: O(N)

Space Complexity: O(1)

Output:



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 \* 104
* s consists of English letters, digits, symbols and spaces.

Program:

class Solution {

public:

    int lengthOfLongestSubstring(string s) {

        int n = s.size();

        int i = 0, j = 0;

        int l = 0;

        unordered\_map<char, int> m;

        while (j < n) {

            if(m.find(s[j])!=m.end()){

                if(m[s[j]]>=i){

                  i=m[s[j]]+1;

                }

            }

            l=max(l,j-i+1);

            m[s[j]]=j;

            j+=1;

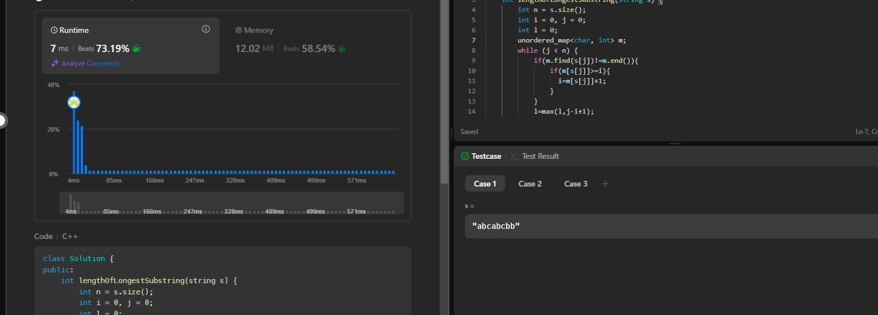
        }

        return l;

    }

};

Output:



Time Complexity: O(n)

Space Complexity: O(k=26)