Archana Goli

Data Structures tasks for week1

Introduction to Arrays:

Arrays are a collection of elements of similar data types stored in contiguous memory locations. Arrays have fixed memory size.

Arrays are of 2 types:

a) Static Arrays: Allocates memory at compile time and size of the arrays is fixed.

b) Dynamic Arrays: Arrays grow automatically when there is no space left to insert new item.

Array declaration:

datatype name = new datatype[size];

Time complexity to access an array: O(1)

Compiler can calculate the address of any element at ith index in constant amount of time.

Address of ith index: Base address + offset

Offset= i*(size of an element) for example size of element for int datatype is 4

Practice Arrays:

Array Creation, Insertion, Deletion and Traversal:

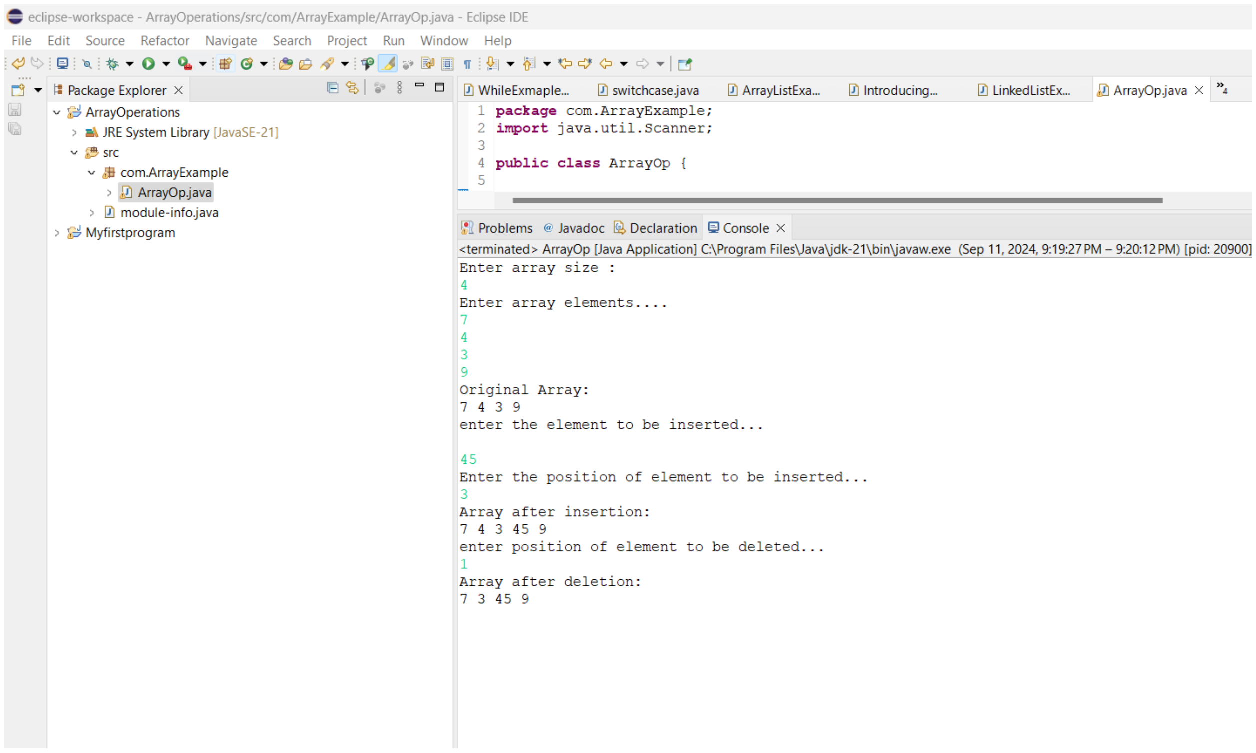
```
package com.ArrayExample;
```

```
import java.util.Scanner;
public class ArrayOp {

        public static void main(String args[])
        {
                System.out.println("Enter array size :");
                int size;
                Scanner ob=new Scanner(System.in);
                size=ob.nextInt();

                //creation of array
                int[] array = new int[size];

                System.out.println("Enter array elements....");
                for(int i=0;i<array.length;i++)
                {
                        array[i]=ob.nextInt();
                }
                ArrayOp a=new ArrayOp();

                //traversal
     System.out.println("Original Array:");
     a.traverseArray(array);
```

```
    //Insertion
    System.out.println("enter the element to be inserted...");
    int n=ob.nextInt();
    System.out.println("Enter the position of element to be inserted...");
    int position=ob.nextInt();
    array = a.insertElement(array, n, position); // non static method is accessed
with the object from static method
    System.out.println("Array after insertion:");
    a.traverseArray(array);


    //Deletion
    System.out.println("enter position of element to be deleted...");
    int del=ob.nextInt();
    array = deleteElement(array, del);  // static method accessed without need of
object from static method
    System.out.println("Array after deletion:");
    a.traverseArray(array);
     }
     public void traverseArray(int[] arr )
     {
         for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
         }
         System.out.println();
     }
```

```
   public int[] insertElement(int[] array, int new_el, int position)
   {
             if (position < 0 || position > array.length) {
    System.out.println("Position is invalid.Please enter correct position");
    return array;
}
int[] new_Arr = new int[array.length + 1];

for (int i = 0; i < position; i++) {
   new_Arr[i] = array[i];
}

new_Arr[position] = new_el;



for (int i = position; i < array.length; i++) {
   new_Arr[i + 1] = array[i];
}

return new_Arr;
   }
   public static int[] deleteElement(int[] array, int position) {
if (position < 0 || position >= array.length) {
    System.out.println("Position is invalid.Please enter correct position");
```

```
            return array;
        }
        int[] new_arr = new int[array.length - 1];


        for (int i = 0; i < position; i++) {
            new_arr[i] = array[i];
        }
        for (int i = position; i < array.length - 1; i++) {
            new_arr[i] = array[i + 1];
        }
        return new_arr;
            }
}
```



eclipse-workspace - ArrayOperations/src/com/ArrayExample/ArrayOp.java - Eclipse IDE

```
package com.ArrayExample;
import java.util.Scanner;

public class ArrayOp {

```

<terminated> ArrayOp [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe (Sep 11, 2024, 9:19:27 PM – 9:20:12 PM) [pid: 20900]

```
Enter array size :
4
Enter array elements....
7
4
3
9
Original Array:
7 4 3 9
enter the element to be inserted...

45
Enter the position of element to be inserted...
3
Array after insertion:
7 4 3 45 9
enter position of element to be deleted...
1
Array after deletion:
7 3 45 9
```

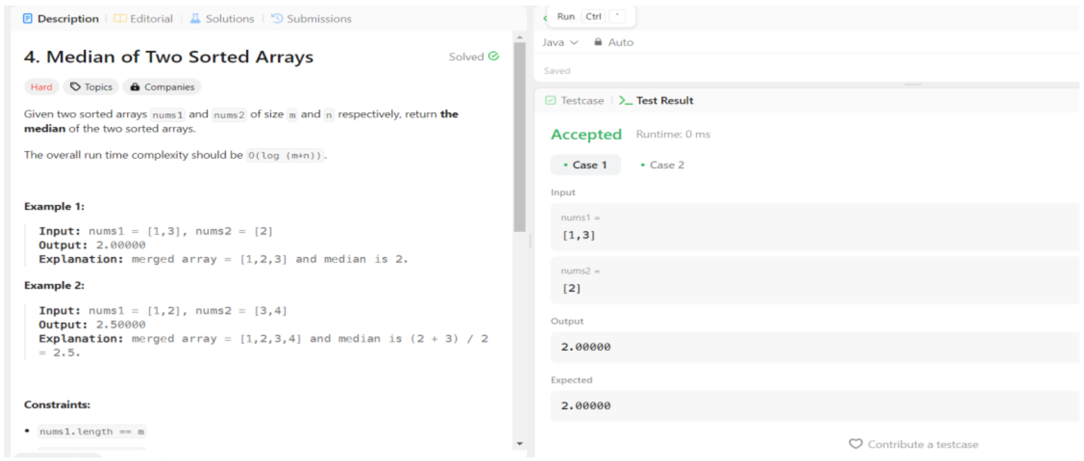
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
ArrayOperations
JRE System Library [JavaSE-21]
src
com.ArrayExample
ArrayOp.java
module-info.java
Myfirstprogram
WhileExmaple... switchcase.java ArrayListExa... Introducing... LinkedListEx... ArrayOp.java
Problems Javadoc Declaration Console

Leet-code problems:

Median of two sorted Arrays:

```
import java.util.Arrays;
class Solution {
    public double findMedianSortedArrays(int[] nums1, int[] nums2) {
        int a[]=new int[nums1.length+nums2.length];
        int j=0;
        for(int i=0;i<nums1.length;i++)
        {
            a[j]= nums1[i];
           // System.out.println(a[j]);
            j++;
        }
        for(int k=0;k<nums2.length;k++)
        {
            a[j]= nums2[k];
           // System.out.println(a[j]);
            j++;
        }
        Arrays.sort(a);
        double t=0.0;
       // System.out.println(a[0]+","+a[1]+","+a[2]);
```

```
        if(a.length%2==0)
        {
          double n1=a[(a.length/2)-1];
          double n2=a[(a.length/2)];
          t=(n1+n2)/2;
        }
        else
         t=a[(a.length/2)];
        return t;
    }
}
```



4. Median of Two Sorted Arrays

Given two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be `O(log (m+n))`.

Example 1:

```
Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2.
```

Example 2:

```
Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2
= 2.5.
```

Constraints:

- `nums1.length == m`

Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
nums1 =
[1,3]

nums2 =
[2]
```

Output

```
2.00000
```

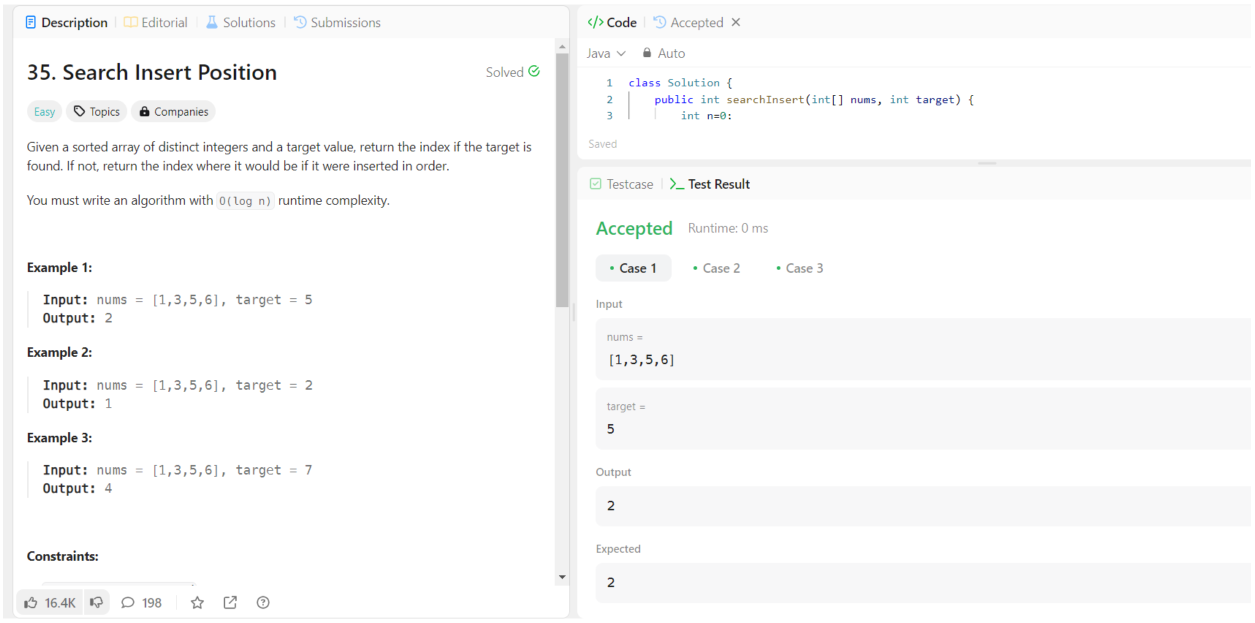
Expected

```
2.00000
```

Contribute a testcase

Search Insert Position:

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



Description | Editorial | Solutions | Submissions

35. Search Insert Position

Solved

Easy | Topics | Companies

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:

16.4K | 198

Code | Accepted

Java | Auto

```
class Solution {
    public int searchInsert(int[] nums, int target) {
        int n=0;
```

Saved

Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 | Case 2 | Case 3

Input

nums =
[1,3,5,6]

target =
5

Output

2

Expected

2

LinkedList:

A LinkedList is a type of linear data structure whose nodes are linked together via pointers. Every node is made up of two components:

Data: The actual value stored in a node.
A pointer or reference to the following node in the list.

A linked list does not store its elements in consecutive memory regions like an array does. Rather, the elements are dispersed throughout memory, but they form a chain since each element is aware of the address of the subsequent element.

Different Linked List Types:
Single-linked list: Every node has a reference to the node after it as well as data. There is only one direction of traversal (from head to tail).
Double-linked list: Every node has two references—one to the previous node and one to the next—as well as data. There are two possible directions for traversal.

Circular-linked list: The final node creates a circle by pointing back to the first one. It can be either singly or doubly linked.

creation, insertion, deletion, and traversal of Linked List

LinkedList Main class:

```
package Com.Example;


import java.util.Collection;

import java.util.Collections;

import java.util.Iterator;

import java.util.LinkedList;

import java.util.Scanner;


public class LinkedListmainClass {
	public static void main(String[] args) {
		LinkedListExample op=new LinkedListExample();
		Scanner obj= new Scanner(System.in);

		System.out.println("Enter number elements you want to add to 
list..");
		int num=obj.nextInt();
		System.out.println("Enter the elements to be inserted...");
		int data;
		for(int i=0;i<num;i++)
		{
			data=obj.nextInt();
			op.insert(data);
		}


		//Insertion at specified index
		System.out.println("enter the index at which element to be 
inserted...");
		int idx= obj.nextInt();

		System.out.println("enter the element to be inserted at the index...");
		int new_e=obj.nextInt();
		op.insertAtIndex(idx, new_e);
```

```
                    System.out.println("Result Linked List: ");
                    op.show();

                    System.out.println("\nEnter the index at which element to be 
deleted");
                    int index=obj.nextInt();
                    op.deleteAtIndex(index);

                    System.out.println("Result linked List is : ");
                    op.show();

          }

}
```

LinkedList Operations class

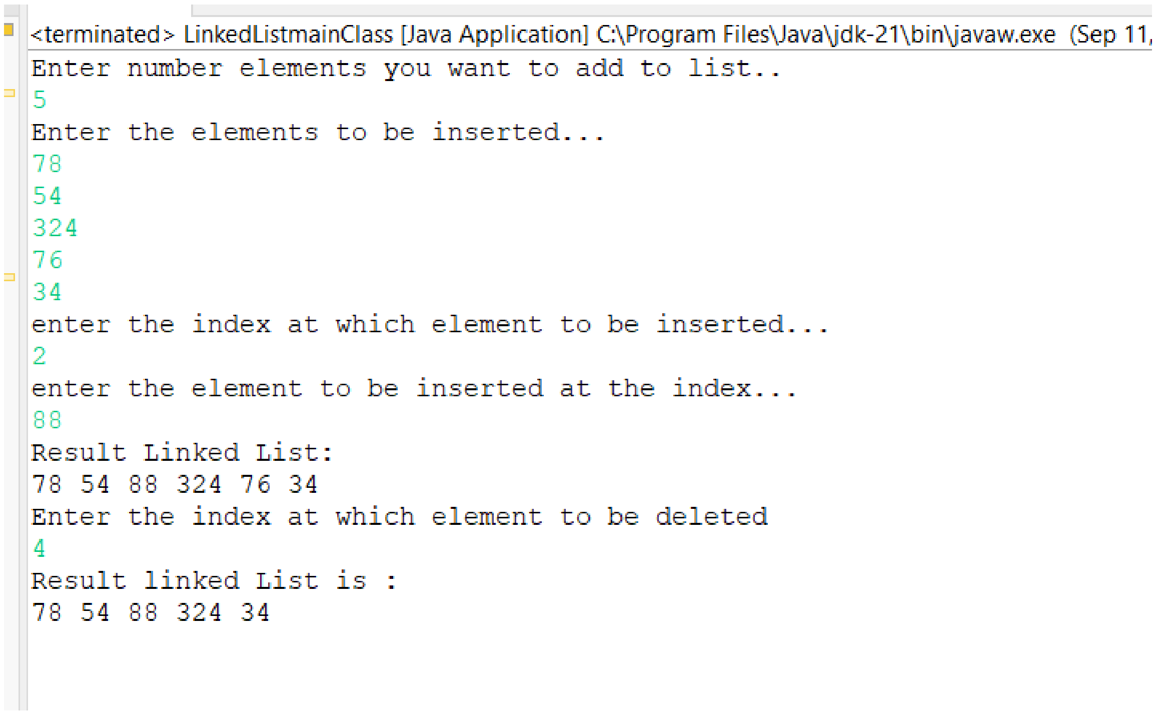
```
package Com.Example;

public class LinkedListExample {

                    LinkedListNodeClass head=null;  //refer to the first node of list

                    public void insert(int data)
                    {
```

```
			LinkedListNodeClass node=new LinkedListNodeClass();  // when we insert new
val, create new node
			node.data=data;
			node.next=null;

			if(head==null) {   // checking if the list is empty, if so making the node as first
node
				head=node;
			}
			else
			{
				LinkedListNodeClass n=head;
				while(n.next!=null)
				{
					n=n.next;
				}
				n.next=node;
			}
		}
		public void insertAtStart(int data)
		{
			LinkedListNodeClass node =new LinkedListNodeClass();
			node.data=data;
			node.next=null;
			node.next=head; //assigning previous head node as next node to the current
node
			head=node;     //making the current node as a head node
		}
		public void insertAtIndex(int index, int data)
		{
```

```
                LinkedListNodeClass node=new LinkedListNodeClass();  // when we insert new
val, create new node
                node.data=data;
                node.next=null;
                LinkedListNodeClass n=head;
                if(index==0)
                {
                        insertAtStart(data);
                }
                for(int i=0;i<index-1;i++)
                {
                        n=n.next;           //reaching the node before the index
                }
                node.next=n.next;  //assigning the reference of n.next to the new node
                n.next=node;                // assigning the node node reference to the n
        }
        public void show()
        {
                LinkedListNodeClass n=head;
                while(n.next!=null)
                {
                        System.out.print(n.data+" ");
                        n=n.next;
                }
                System.out.print(n.data+" ");
        }
        public void deleteAtStart() {
    if (head == null) {
      System.out.println("List is empty...");
```

```
        } else {
            head = head.next; // As we are removing the first node, move head to the next node
        }
    }

    public void deleteAtIndex(int index) {
        if (head == null) {
            System.out.println("List is empty...");
            return;
        }

        if (index == 0) {
            deleteAtStart(); //delete first element if specified index is 0
            return;
        }

        LinkedListNodeClass n = head;
        for (int i = 0; i < index - 1; i++) {
            if (n.next == null) {
                System.out.println("Index out of bounds.");
                return;
            }
            n = n.next;
        }

        if (n.next == null) {
            System.out.println("Index out of bounds.");
            return;
        }
```

```
            n.next = n.next.next; // point to next of next node to skip the node at specified index
        }
    }
```



```
<terminated> LinkedListmainClass [Java Application] C:\Program Files\Java\jdk-21\bin\javaw.exe  (Sep 11,
Enter number elements you want to add to list..
5
Enter the elements to be inserted...
78
54
324
76
34
enter the index at which element to be inserted...
2
enter the element to be inserted at the index...
88
Result Linked List:
78 54 88 324 76 34
Enter the index at which element to be deleted
4
Result linked List is :
78 54 88 324 34
```

Compare and contrast arrays and linked lists in terms of performance and use cases.

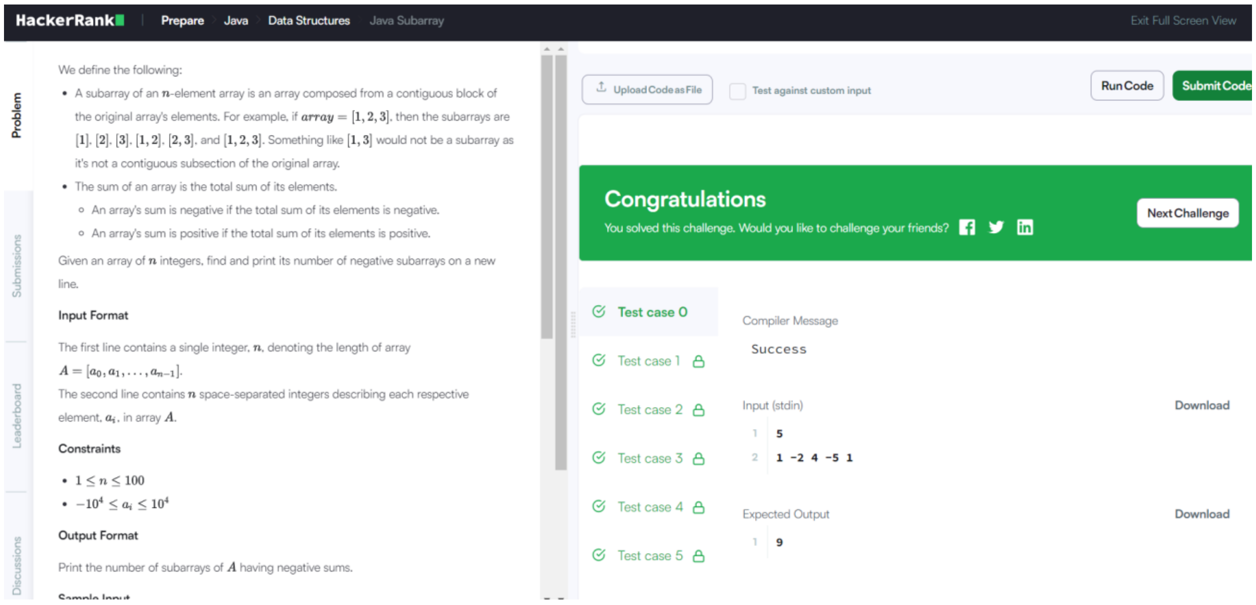
Aspect	Arrays	Linked List
Memory Allocation	Contiguous blocks of memory are allocated Size of Arrays is fixed, re-sizing is done when required	Non contiguous nodes linked by pointers are used to allocate memory. Size increases dynamically.
Access Time	Arrays uses indices to access the elements directly. So the random access time is O(1)	Since the nodes must be traversed from head to target node, the access time O(n)
Insertion/Deletion	Insertion/deletion at the end	Insertion at the beginning takes

	takes O(1) time complexity and Insertion/deletion at the arbitrary position takes O(n) time complexity due to shifting.	O(1) time complexity.Insertion/deletion at the end takes O(n) time complexity since traversal is required to access the last node. Insertion/deletion in the middle also takes O(n) time complexity.
Memory Efficiency	Efficient, Since the arrays use contiguous memory allocation, these does not require extra memory to store the pointers.	Less efficient,Each node requires extra memory to store the pointers to next or previous nodes.
Search Efficiency	O(log n) for binary search (in sorted arrays) O(n) for linear search in unsorted arrays	Binary search is not feasible due to lack of index based access. In case of linear search, each element need to be checked one by one so it takes O(n) time complexity.
Best Use case	Best in situations where the size is known in advance or varies seldom. Perfect for matrices, fixed-size buffers, lookup tables, and random access use cases.	Perfect for dynamic datasets that undergo frequent additions and removals (such as queues, stacks, and undo features). functions effectively in situations where sequential access is adequate and dynamic memory management is necessary.

Handson Practice: Arrays

Finding the number of subarrays having negative sums

```
import java.io.*;

import java.util.*;

import java.text.*;
import java.math.*;
import java.util.regex.*;
public class Solution {
    public static void main(String[] args) {
    Scanner sc=new Scanner(System.in);
    int n=sc.nextInt();
    if(n<=0)
    {
        System.out.println("Array size must be greater than 0");
        return;
    }
    long[] array1= new long[n];
    for(int i=0;i<n;i++){
        array1[i]=sc.nextInt();
    }
    int count=0;
    for(int j=0;j<n;j++){
        long sum=0;
        for(int k=j;k<n;k++){
            sum=array1[k]+sum;
            if(sum<0){
                count++;
            }
        }
        }
        System.out.println(count);
    }
}
```



We define the following:

- A subarray of an n-element array is an array composed from a contiguous block of the original array's elements. For example, if $array = [1, 2, 3]$, then the subarrays are $[1]$, $[2]$, $[3]$, $[1, 2]$, $[2, 3]$, and $[1, 2, 3]$. Something like $[1, 3]$ would not be a subarray as it's not a contiguous subsection of the original array.
- The sum of an array is the total sum of its elements.
  - An array's sum is negative if the total sum of its elements is negative.
  - An array's sum is positive if the total sum of its elements is positive.

Given an array of n integers, find and print its number of negative subarrays on a new line.

Input Format

The first line contains a single integer, n, denoting the length of array $A = [a_0, a_1, \ldots, a_{n-1}]$.
The second line contains n space-separated integers describing each respective element, a_i, in array A.

Constraints

- $1 \le n \le 100$
- $-10^4 \le a_i \le 10^4$

Output Format

Print the number of subarrays of A having negative sums.

Sample Input

Congratulations

You solved this challenge. Would you like to challenge your friends?

✓ Test case 0
✓ Test case 1
✓ Test case 2
✓ Test case 3
✓ Test case 4
✓ Test case 5

Compiler Message

```
Success
```

Input (stdin)

```
5
1 -2 4 -5 1
```

Expected Output

```
9
```

Approach:

- Used Scanner to read the input values
- If length of the array entered as 0 or negative value, then prompt is displayed as “Array size must be greater than 0”.
- Array is created with long data type to handle larger input values.
- Counter is initialized with zero initially which is used to store the count of number of subarrays have negative sum.
- Nested for loop is used to iterate over the array elements.
- Outer for loop starts from each index i of the array.
- Inner for loop calculates the sum of subarrays starting from index i to every subsequent index j.

- If the cumulative sum of current subarray is less than 0 then increment the count value.
- Print the count of subarrays having negative sum.

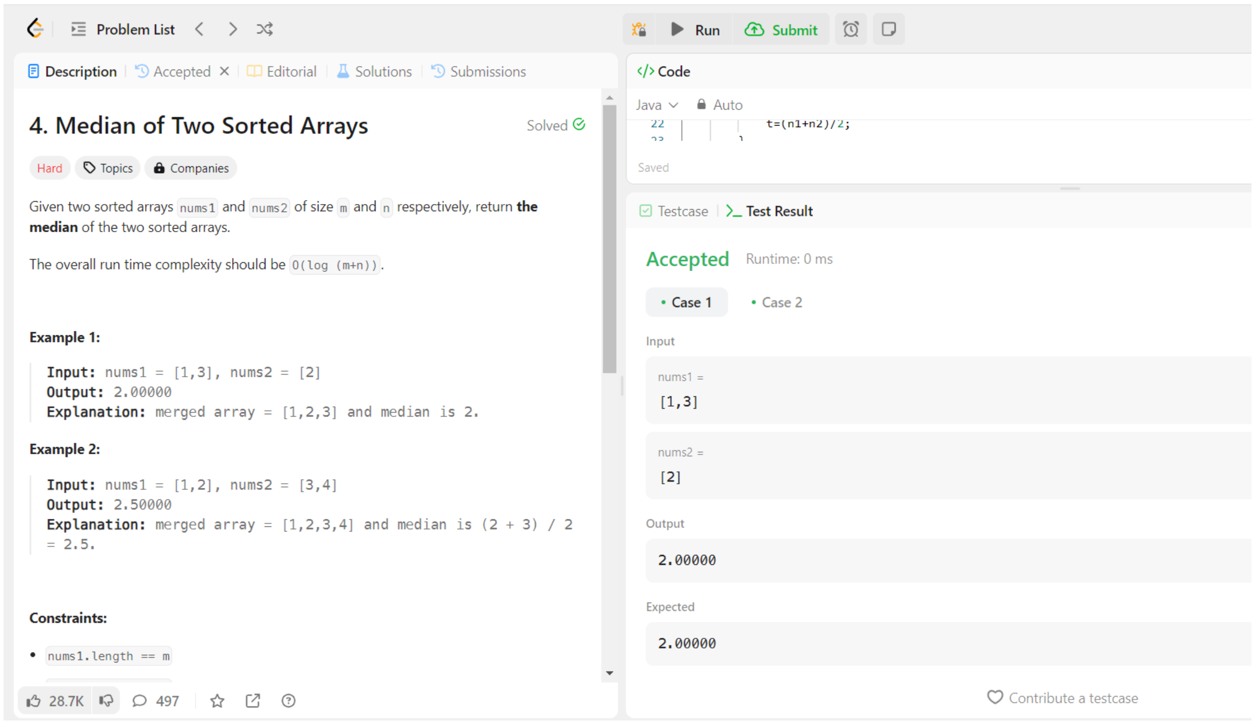
Challenges:

- Invalid input handling: Achieved by adding a condition that checks whether the size of array is less than or equal to 0. If yes, print a prompt and return.
- Performance is inefficient since we are using nested for loops which takes time complexity of O(n^2) to calculate the sum of subarrays.
- Input validation: Input entered must be a integer otherwise the program crashes.

Find median of two sorted Arrays:

```
import java.util.Arrays;

class Solution {

  public double findMedianSortedArrays(int[] nums1, int[] nums2) {

    int a[]=new int[nums1.length+nums2.length];

    int j=0;

    for(int i=0;i<nums1.length;i++)

    {

      a[j]= nums1[i];

      j++;

    }

    for(int k=0;k<nums2.length;k++)

    {

      a[j]= nums2[k];

      j++;
```

```
        }
        Arrays.sort(a);
        double t=0.0;
        if(a.length%2==0)
        {
            double n1=a[(a.length/2)-1];
            double n2=a[(a.length/2)];
            t=(n1+n2)/2;
        }
        else
          t=a[(a.length/2)];
        return t;
    }
}
```



4. Median of Two Sorted Arrays

Given two sorted arrays `nums1` and `nums2` of size `m` and `n` respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be `O(log (m+n))`.

Example 1:

```
Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2.
```

Example 2:

```
Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.
```

Constraints:

- `nums1.length == m`

```
t=(n1+n2)/2;
```

Accepted Runtime: 0 ms

Case 1 Case 2

Input

nums1 =
```
[1,3]
```

nums2 =
```
[2]
```

Output
```
2.00000
```

Expected
```
2.00000
```

Approach:

Create an array (a) with size equal to the sum of sizes of the first array (nums1) and the second array(nums2).

Iterate through the first array(nums1) and store nums1 array elements in array a.

Followed by nums2 array elements. Iterate through the second array (num2) and store second array elements in array a.

Next sort the array “a” using sort method which is the method of Arrays class.

Median conditions:

a) If number of array elements is even, then median = ((value at index (length of array/2)-1) +(value at index (length of array/2)))/2
b) If number of array elements is add, then median = (value at index (length of array/2))

Challenges:

- Performance Concern:
  Problem: The method consists of combining two arrays into one and sorting the combined array; this results in an O((m+n) log(m+n)) time complexity, where m and n are the lengths of nums1 and nums2, respectively. When sorting huge arrays, sorting can be ineffective.
  Impact: This method is less effective than other methods for really big arrays since the sorting process may become a performance bottleneck.

- Space complexity:
  Problem: The technique has a space complexity of O(m+n) since it consumes extra space for the merged array a.
  Impact: When working with very big arrays or in contexts with restricted memory, this extra space utilization might be a problem.

Stack:

Stack is a linear data structure that follows LIFO strategy. The last added element is removed first. Stacks are used in many applications like function calls, expression evaluation and backtracking algorithms.

Stack operations:

Push: add element to the top of the stack. Time Complexity O(1)

Pop: remove and returns the element from the top of the stack. Time Complexity O(1)

Peek: retrieves element from the top of the stack without removing it from the stack. Time Complexity O(1)

isEmpty: checks if the stack is empty. Time Complexity O(1)

size: returns number of elements in the stack. Time Complexity O(1)

Queues:

Queue is a linear data structure that follows FIFO strategy. The First added element will be fetched first. Queues are used in scenarios such as task scheduling, managing requests and buffering.

Queue operations:

Enqueue: add element to the end of the queue

Dequeue: remove and return element from front of the queue

Peek: retrieve the element from front of the queue without removing it from the queue.

isEmpty: checks if the queue is empty

size: returns number of elements in the queue.

Hash Tables:

Hash table is a data structure which uses hash function to compute an index into an array of buckets to make an efficient insertion, deletion and retrieval of elements. It is used for implementing associative arrays or dictionary-like data structures

HashTable operations:

Insert(put): Adds key-value pair to the hash table

Search(get): retrieves the value associated with the given key

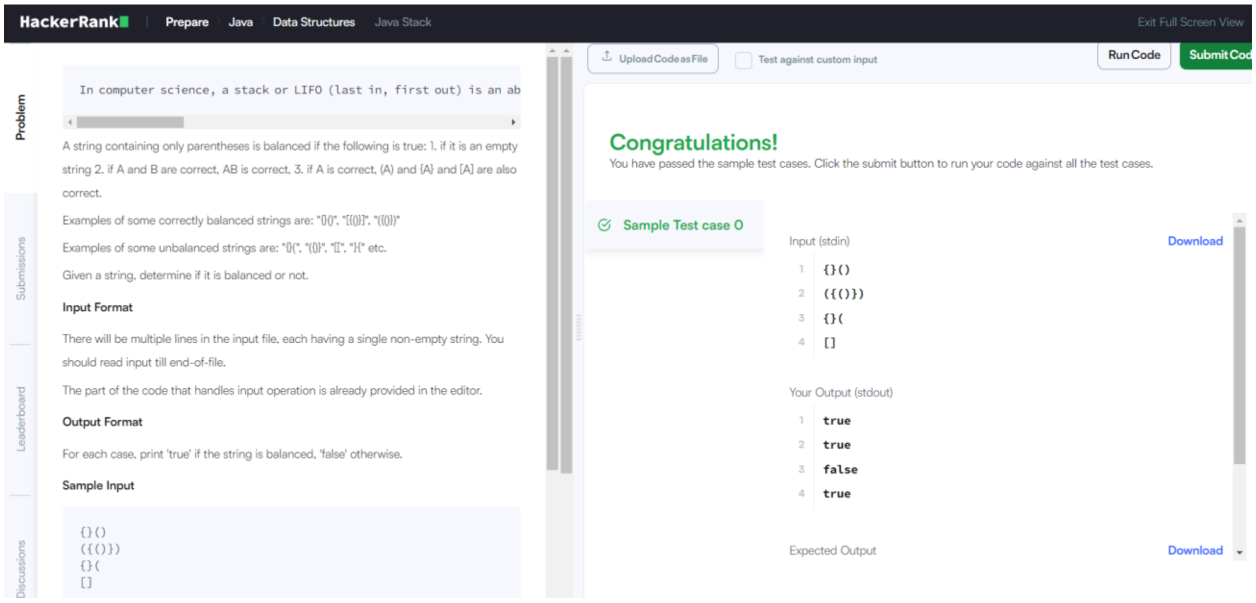
Delete(remove): removes key-value pair associated with the given key

Contains: checks if key-value pair is present in the hash table

Leet Code practice:

Stack: Program to check whether given string is balanced string or not.

```
import java.io.*;

import java.util.*;

import java.text.*;

import java.math.*;
import java.util.regex.*;

public class Solution {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        while(scanner.hasNextLine()) {
            Stack<Character> stack = new Stack<>();
            String line = scanner.nextLine();
            for(char c : line.toCharArray()) {
                if(c == '{' || c == '(' || c == '[') {
                   stack.push(c);
                   continue;
                }
                if(c == '}' && !stack.isEmpty() && stack.peek() == '{') {
                    stack.pop();
                    continue;
                }
                if(c == ')' && !stack.isEmpty() && stack.peek() == '('){
                    stack.pop();
                    continue;
                }
                if(c == ']' && !stack.isEmpty() && stack.peek() == '['){
                    stack.pop();
                    continue;
                }
                if(c == '}' || c == ')' || c == ']') {
                    stack.push(c);
                break;
              }
            }
            System.out.println(stack.isEmpty());
          }
        }
      }
```



In computer science, a stack or LIFO (last in, first out) is an ab

A string containing only parentheses is balanced if the following is true: 1. if it is an empty string 2. if A and B are correct, AB is correct, 3. if A is correct, (A) and {A} and [A] are also correct.

Examples of some correctly balanced strings are: "{}()", "[{()}]", "({()})"

Examples of some unbalanced strings are: "{}(", "({)}", "[[", "}{" etc.

Given a string, determine if it is balanced or not.

Input Format

There will be multiple lines in the input file, each having a single non-empty string. You should read input till end-of-file.

The part of the code that handles input operation is already provided in the editor.

Output Format

For each case, print 'true' if the string is balanced, 'false' otherwise.

Sample Input

```
{}()
({()})
{}(
[]
```

Congratulations!

You have passed the sample test cases. Click the submit button to run your code against all the test cases.

Sample Test case 0

Input (stdin)

```
{}()
({()})
{}(
[]
```

Your Output (stdout)

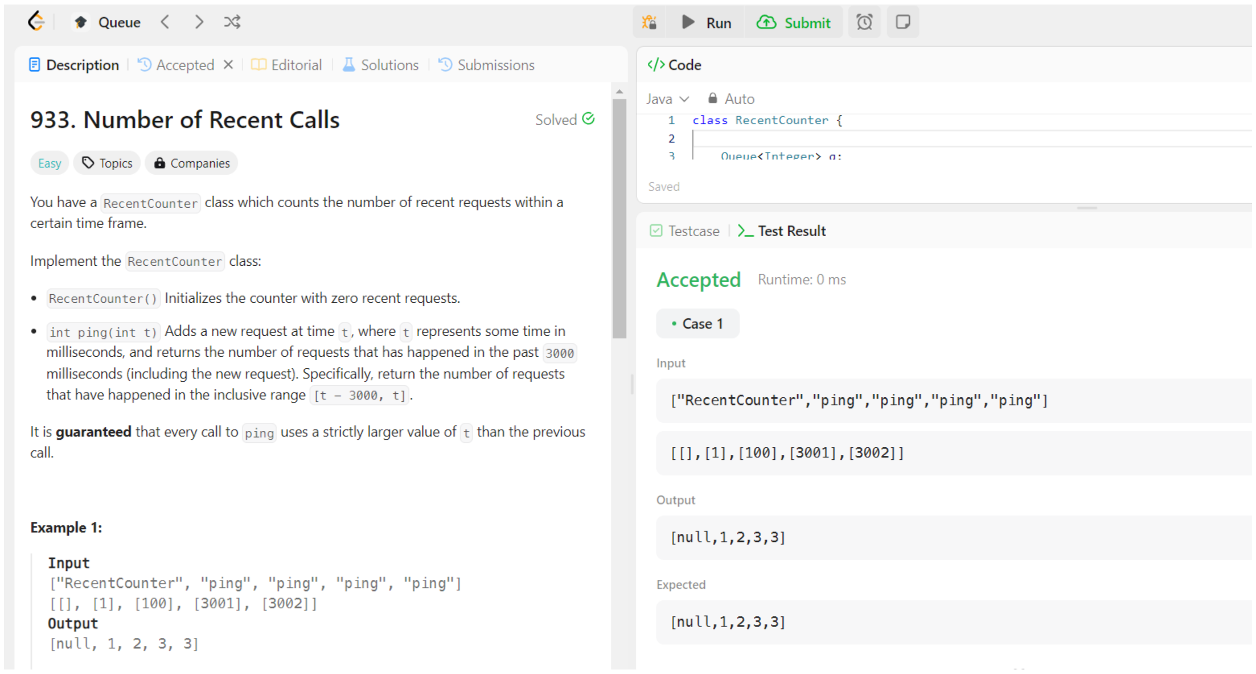
```
true
true
false
true
```

Expected Output

Queue: Number of recent calls

```
class RecentCounter {

  Queue<Integer> q;

  public RecentCounter() {
    q = new LinkedList<>();
  }
```

```
    public int ping(int t) {
        q.add(t);
        while (q.peek() < t - 3000) {
            q.poll();
        }
        return q.size();
    }
}
```



933. Number of Recent Calls

Solved

Easy

You have a `RecentCounter` class which counts the number of recent requests within a certain time frame.

Implement the `RecentCounter` class:

- `RecentCounter()` Initializes the counter with zero recent requests.
- `int ping(int t)` Adds a new request at time `t`, where `t` represents some time in milliseconds, and returns the number of requests that has happened in the past `3000` milliseconds (including the new request). Specifically, return the number of requests that have happened in the inclusive range `[t - 3000, t]`.

It is **guaranteed** that every call to `ping` uses a strictly larger value of `t` than the previous call.

Example 1:

```
Input
["RecentCounter", "ping", "ping", "ping", "ping"]
[[], [1], [100], [3001], [3002]]
Output
[null, 1, 2, 3, 3]
```

```
class RecentCounter {

    Queue<Integer> q;
```

Accepted Runtime: 0 ms

Case 1

Input

```
["RecentCounter","ping","ping","ping","ping"]
[[],[1],[100],[3001],[3002]]
```

Output

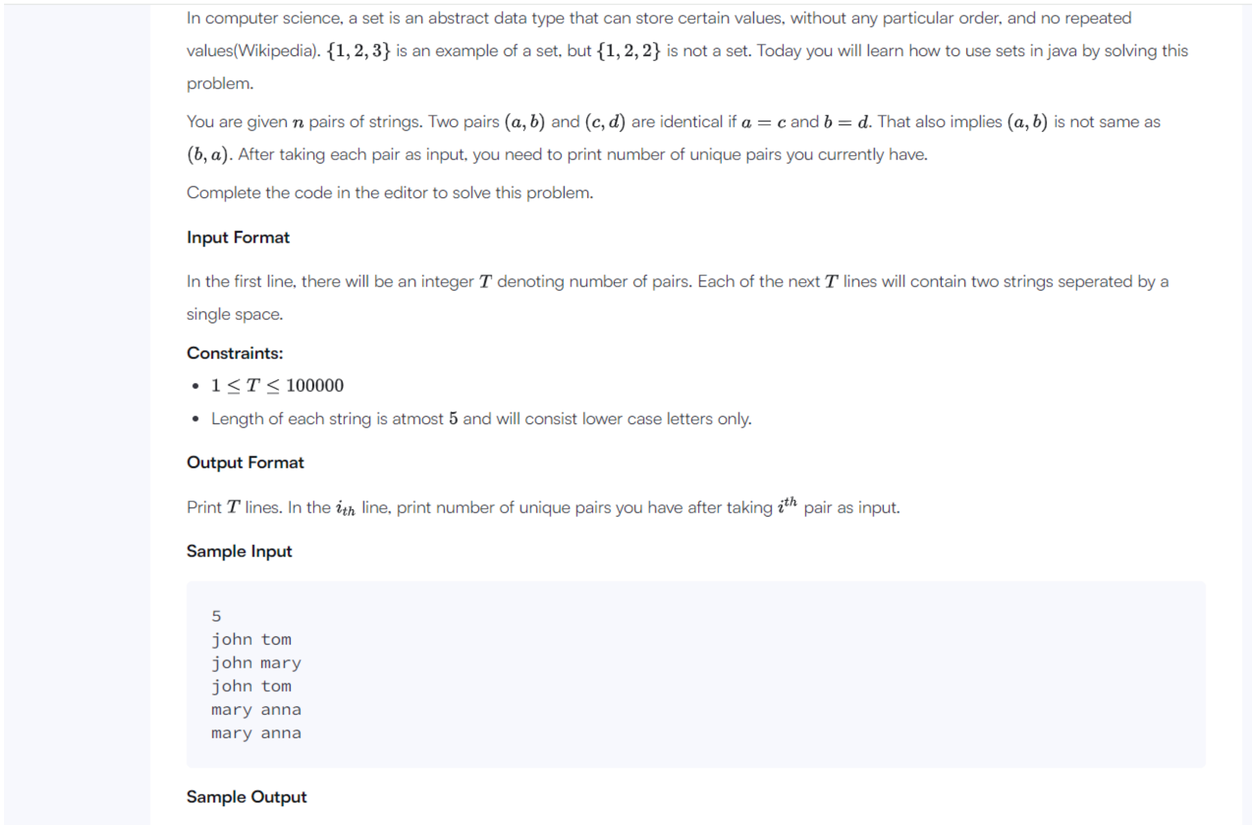
```
[null,1,2,3,3]
```

Expected

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

HashSet: number of unique pairs of Strings after each input

Problem statement:



In computer science, a set is an abstract data type that can store certain values, without any particular order, and no repeated values(Wikipedia). $\{1, 2, 3\}$ is an example of a set, but $\{1, 2, 2\}$ is not a set. Today you will learn how to use sets in java by solving this problem.

You are given n pairs of strings. Two pairs (a, b) and (c, d) are identical if $a = c$ and $b = d$. That also implies (a, b) is not same as (b, a). After taking each pair as input, you need to print number of unique pairs you currently have.

Complete the code in the editor to solve this problem.

Input Format

In the first line, there will be an integer T denoting number of pairs. Each of the next T lines will contain two strings seperated by a single space.

Constraints:

- $1 \le T \le 100000$
- Length of each string is atmost 5 and will consist lower case letters only.

Output Format

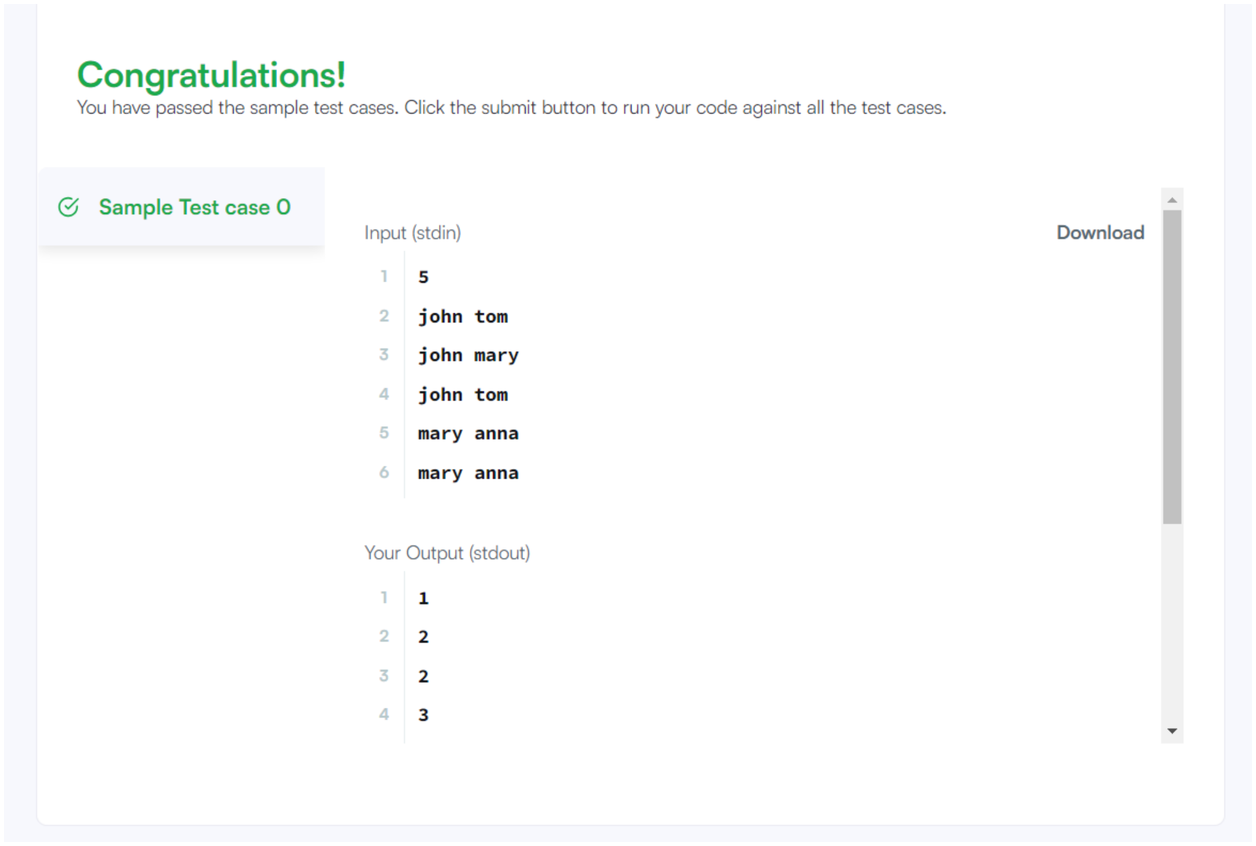
Print T lines. In the i_{th} line, print number of unique pairs you have after taking i^{th} pair as input.

Sample Input

```
5
john tom
john mary
john tom
mary anna
mary anna
```

Sample Output

```
import java.io.*;

import java.util.*;

import java.text.*;

import java.math.*;

import java.util.regex.*;


public class Solution {


 public static void main(String[] args) {

        Scanner s = new Scanner(System.in);

        int t = s.nextInt();

        String [] pair_left = new String[t];

        String [] pair_right = new String[t];
        for (int i = 0; i < t; i++) {
            pair_left[i] = s.next();
            pair_right[i] = s.next();
        }
        Set<String> st=new HashSet<String>();
        String [] str = new String[t];
        for (int i = 0; i < t; i++) {
          str[i]= pair_left[i]+" "+pair_right[i];
          st.add(str[i]);
          System.out.println(st.size());
        }
    }
}
```



Congratulations!

You have passed the sample test cases. Click the submit button to run your code against all the test cases.

Sample Test case 0

Input (stdin)

Download

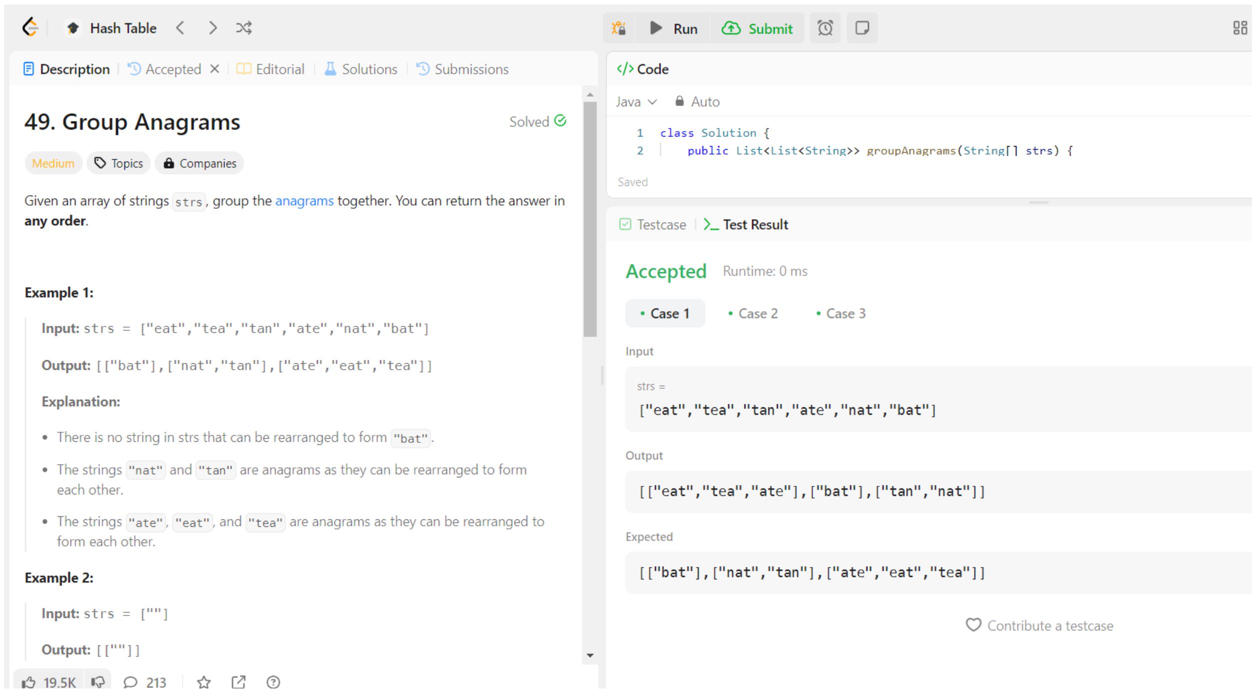
```
5
john tom
john mary
john tom
mary anna
mary anna
```

Your Output (stdout)

```
1
2
2
3
```

HashMap: Program to Group Anagrams

```
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
       HashMap<String, List<String>> h=new HashMap<>();
       for(String s:strs)
       {
           char[] c=s.toCharArray();
           Arrays.sort(c);
           String key=new String(c);
           if(!h.containsKey(key))
           {
               h.put(key,new ArrayList<>());
           }
           h.get(key).add(s);
       }
       //System.out.println(h.values());
       return new ArrayList<>(h.values());
    }
}
```



49. Group Anagrams

Given an array of strings `strs`, group the anagrams together. You can return the answer in **any order**.

Example 1:

Input: `strs = ["eat","tea","tan","ate","nat","bat"]`

Output: `[["bat"],["nat","tan"],["ate","eat","tea"]]`

Explanation:

- There is no string in strs that can be rearranged to form `"bat"`.
- The strings `"nat"` and `"tan"` are anagrams as they can be rearranged to form each other.
- The strings `"ate"`, `"eat"`, and `"tea"` are anagrams as they can be rearranged to form each other.

Example 2:

Input: `strs = [""]`

Output: `[[""]]`

```java
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
```

Accepted Runtime: 0 ms

Case 1 · Case 2 · Case 3

Input

strs =

```
["eat","tea","tan","ate","nat","bat"]
```

Output

```
[["eat","tea","ate"],["bat"],["tan","nat"]]
```

Expected

```
[["bat"],["nat","tan"],["ate","eat","tea"]]
```

Sorting Algorithms:

Bubble sort:

It is a simple sorting algorithm that iterates over the list repeatedly and compares two adjacent elements and swaps them if they are out of order. The process is repeated until the list is sorted.

Best case time complexity: O(n) if the list is already sorted

Average case time complexity: O(n^2)

Worst case time complexity: O(n^2) where n is the number of array elements

Example:

For the array [5, 3, 8, 4, 2]:

First Pass:

Compare 5 and 3, swap to get [3, 5, 8, 4, 2].

Compare 5 and 8, no swap needed.

Compare 8 and 4, swap to get [3, 5, 4, 8, 2].

Compare 8 and 2, swap to get [3, 5, 4, 2, 8].

Second Pass:

Compare 3 and 5, no swap needed.

Compare 5 and 4, swap to get [3, 4, 5, 2, 8].

Compare 5 and 2, swap to get [3, 4, 2, 5, 8].

Third Pass:

Compare 3 and 4, no swap needed.

Compare 4 and 2, swap to get [3, 2, 4, 5, 8].

Final Pass:

Compare 3 and 2, swap to get [2, 3, 4, 5, 8].

The array is now sorted

Selection sort:

Selection Sort works by dividing the list into a sorted and an unsorted region. It repeatedly selects the smallest element from the unsorted region and places it in the correct position in the sorted region.

Best case time complexity: O(n^2) even if the array already sorted

Average case time complexity: O(n^2)

Worst case time complexity: O(n^2) where n is the number of array elements

Example:

For the array [64, 25, 12, 22, 11]:

Initial array: [64, 25, 12, 22, 11]

Find the minimum element in the unsorted portion ([64, 25, 12, 22, 11]):

Minimum is 11.

Swap 11 with 64.

Array becomes [11, 25, 12, 22, 64].

Find the minimum element in the unsorted portion ([25, 12, 22, 64]):

Minimum is 12.

Swap 12 with 25.

Array becomes [11, 12, 25, 22, 64].

Find the minimum element in the unsorted portion ([25, 22, 64]):

Minimum is 22.

Swap 22 with 25.

Array becomes [11, 12, 22, 25, 64].

Find the minimum element in the unsorted portion ([25, 64]):

Minimum is 25.

No swap needed.

Array remains [11, 12, 22, 25, 64].

Insertion sort:

Insertion Sort operates by splitting the list into regions that are sorted and unsorted. Elements from the unsorted region are regularly taken and inserted into the sorted region at the appropriate location.

Best case time complexity: O(n) when the list is already sorted

Average case time complexity: O(n^2)

Worst case time complexity: O(n^2) where n is the number of array elements

Example:

For the array [5, 3, 8, 4, 2]:

Initial array: [5, 3, 8, 4, 2]

Insert 3:

Compare 3 with 5 (element before it). Since 3 < 5, move 5 to the right.

Insert 3 in the position of 5.

Array becomes [3, 5, 8, 4, 2].

Insert 8:

Compare 8 with 5. Since 8 > 5, it stays in its position.

Array remains [3, 5, 8, 4, 2].

Insert 4:

Compare 4 with 8. Since 4 < 8, move 8 to the right.

Compare 4 with 5. Since 4 < 5, move 5 to the right.

Insert 4 in the position of 5.

Array becomes [3, 4, 5, 8, 2].

Insert 2:

Compare 2 with 8. Since 2 < 8, move 8 to the right.

Compare 2 with 5. Since 2 < 5, move 5 to the right.

Compare 2 with 4. Since 2 < 4, move 4 to the right.

Compare 2 with 3. Since 2 < 3, move 3 to the right.

Insert 2 in the position of 3.

Array becomes [2, 3, 4, 5, 8]

Quick sort:

Quick Sort operates through:

- choosing a pivotal component from the collection.
- dividing the array into two sub-arrays: ones containing elements smaller than the pivot and ones larger than it.
- Applying the same procedure recursively to the two sub-arrays

Best case time complexity: O(nlogn), when the array is always divided into two equal halves by the pivot.

Average case time complexity: O(nlogn), when the array is divided into approximately equal halves by the pivot.

worst case time complexity: O(n^2), in situations when the pivot is poorly chosen (e.g., the largest or smallest element as the pivot).

Example:

For the array [10, 7, 8, 9, 1, 5], let's use the last element as the pivot:

Initial array: [10, 7, 8, 9, 1, 5]

Pivot: 5

Partitioning:

Move elements smaller than 5 to the left and those greater to the right.

Array becomes [1, 7, 8, 9, 10, 5] after partitioning around the pivot 5.

Recursively apply Quick Sort to the left ([1]) and right ([7, 8, 9, 10]) sub-arrays.

For the right sub-array [7, 8, 9, 10]:

Pivot: 10

Since 10 is already in its correct place, no change is needed for this pivot.

Further partition of the remaining elements if necessary and continue recursively.

Merge sort:

Using the Merge Sort algorithm, an array can be sorted by first splitting it up into smaller sub-arrays, sorting those sub-arrays, and then merging them back together. Up until the array is completely sorted, this operation is repeated recursively.

Best case time complexity: O(nlogn) even if the array is already sorted

Average case time complexity: O(nlogn)

Worst case time complexity: O(nlogn)

Example:

Example:

For an array [38, 27, 43, 3, 9, 82, 10]:

Initial Array: [38, 27, 43, 3, 9, 82, 10]

Divide:

Left half: [38, 27, 43]

Right half: [3, 9, 82, 10]

Conquer:

Sort [38, 27, 43]:

Divide further into [38] and [27, 43]

Sort [27, 43] into [27] and [43] and merge into [27, 43]

Merge [38] and [27, 43] into [27, 38, 43]

Sort [3, 9, 82, 10]:

Divide into [3, 9] and [82, 10]

Sort [3, 9] into [3] and [9] and merge into [3, 9]

Sort [82, 10] into [10] and [82] and merge into [10, 82]

Merge [3, 9] and [10, 82] into [3, 9, 10, 82]

Combine:

Merge [27, 38, 43] and [3, 9, 10, 82] into [3, 9, 10, 27, 38, 43, 82]

Searching Algorithms

Linear Search:

Definition: A list's elements are checked one after the other by a linear search algorithm until the requested element is located or the list is exhausted.

Algorithm:

Begin at the top of the list.
One by one, compare the goal value with each item in the list.
Give back the element's index if a match is discovered.
Return -1 if the list's conclusion is reached without the target being found.

Best case time complexity: O(1) when the element is at the first position

Average case time complexity: O(n)

Worst case time complexity: O(n)

Binary Search:

Definition: Binary search divides the search interval in half repeatedly in order to find an entry in a sorted list.

Algorithm:

Start with the sorted list's middle element.
Return the index of the middle element if it is equal to the target.
Limit the search to the left half of the list if the target is smaller than the center element.
Limit the search to the right half of the list if the target exceeds the center element.
Continue until the search interval is empty or the target is located.

Best case time complexity: O(1) when element is at the middle position

Average case time complexity: O(logn)

Worst case time complexity: O(logn)

Terinary Search:

Definition: In order to find an element, the ternary search algorithm divides the list into three parts rather than two.

Algorithm

The list should be divided into three equal (or nearly equal) sections.
Compare the two division points with the aim.
Return the index if the target equals any of the division points.
Search the left third if the objective is less than the first division point.
If the target is between the two dividing points, search the middle third.
Search the right third if the target exceeds the second division point.
Continue until the search interval is empty or the target is located.

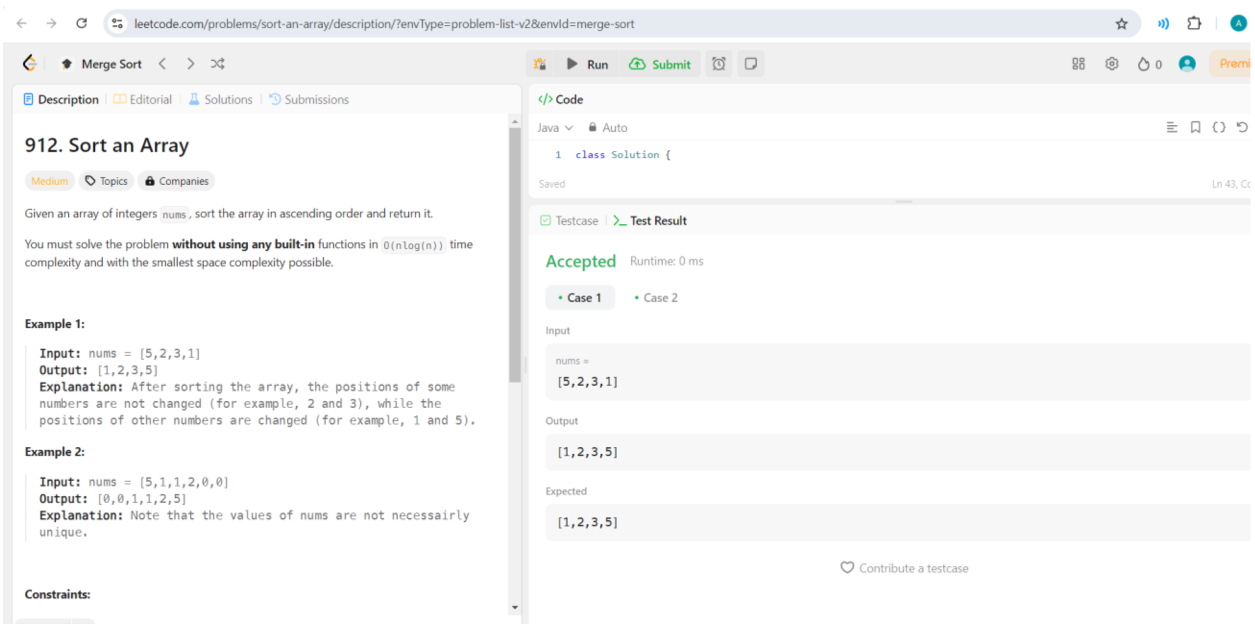
Best case time complexity: O(1)

Average case time complexity: O(logn base 3) better than Binary Search but with a higher constant factor

Worst case time complexity: O(logn base 3)

Leetcode Practice:

Sort Array using Merge Sort:



leetcode.com/problems/sort-an-array/description/?envType=problem-list-v2&envId=merge-sort

912. Sort an Array

Given an array of integers `nums`, sort the array in ascending order and return it.

You must solve the problem **without using any built-in** functions in `O(nlog(n))` time complexity and with the smallest space complexity possible.

Example 1:

```
Input: nums = [5,2,3,1]
Output: [1,2,3,5]
Explanation: After sorting the array, the positions of some
numbers are not changed (for example, 2 and 3), while the
positions of other numbers are changed (for example, 1 and 5).
```

Example 2:

```
Input: nums = [5,1,1,2,0,0]
Output: [0,0,1,1,2,5]
Explanation: Note that the values of nums are not necessarily
unique.
```

Constraints:

```
class Solution {
```

Accepted Runtime: 0 ms

Case 1 · Case 2

Input

```
nums =
[5,2,3,1]
```

Output

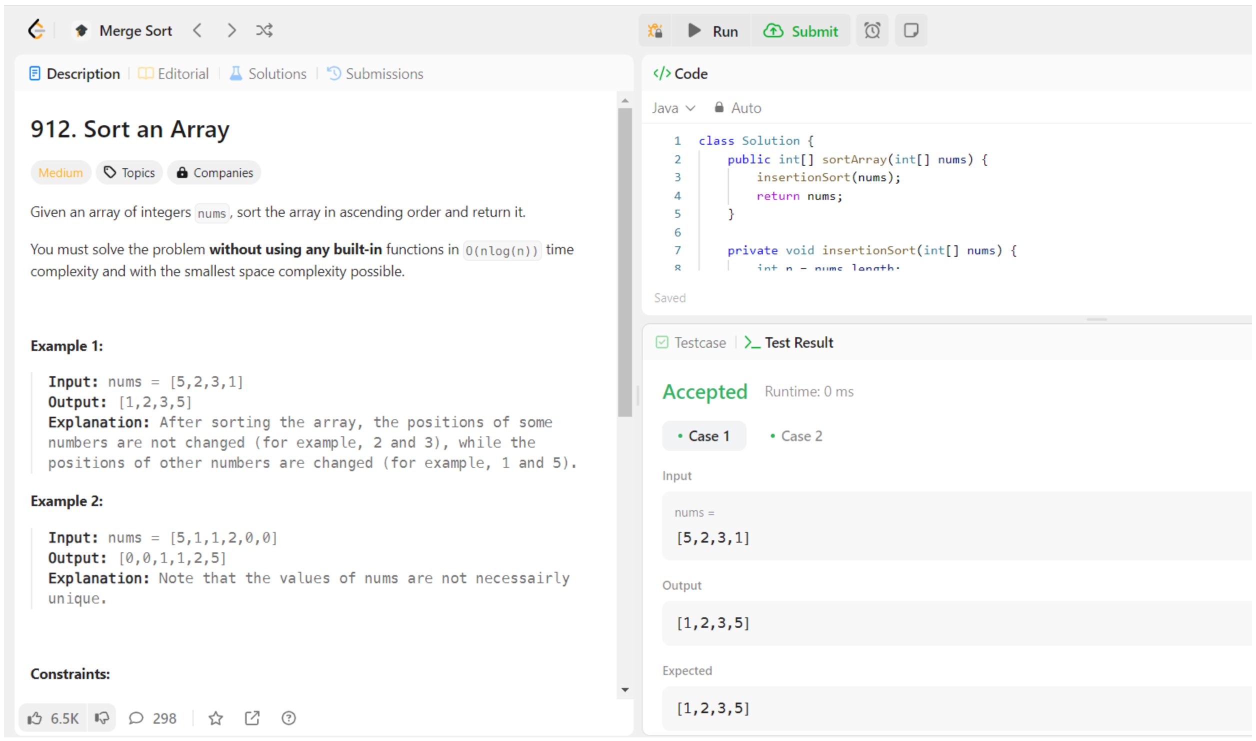
```
[1,2,3,5]
```

Expected

```
[1,2,3,5]
```

```
class Solution {
    public int[] sortArray(int[] nums) {
        mergeSort(nums, 0, nums.length - 1);
        return nums;
    }
    public void mergeSort(int[] nums, int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;
            mergeSort(nums, left, mid);
            mergeSort(nums, mid + 1, right);
            merge(nums, left, mid, right);
        }
    }
    public void merge(int[] nums, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;
        int[] l = new int[n1];
        int[] r = new int[n2];
        for (int i = 0; i < n1; i++) {
            l[i] = nums[left + i];
        }
        for (int j = 0; j < n2; j++) {
            r[j] = nums[mid + 1 + j];
        }
        int i = 0, j = 0;
        int k = left;
        while (i < n1 && j < n2) {
            if (l[i] <= r[j]) {
                nums[k] = l[i];
                i++;
            } else {
                nums[k] = r[j];
                j++;
            }
            k++;
        }
        while (i < n1) {
            nums[k] = l[i];
            i++;
            k++;
        }
        while (j < n2) {
            nums[k] = r[j];
            j++;
            k++;
        }
    }
}
```

Insertion Sort:



Merge Sort

Run Submit

Description | Editorial | Solutions | Submissions

912. Sort an Array

Medium Topics Companies

Given an array of integers `nums`, sort the array in ascending order and return it.

You must solve the problem **without using any built-in** functions in `O(nlog(n))` time complexity and with the smallest space complexity possible.

Example 1:

```
Input: nums = [5,2,3,1]
Output: [1,2,3,5]
Explanation: After sorting the array, the positions of some
numbers are not changed (for example, 2 and 3), while the
positions of other numbers are changed (for example, 1 and 5).
```

Example 2:

```
Input: nums = [5,1,1,2,0,0]
Output: [0,0,1,1,2,5]
Explanation: Note that the values of nums are not necessairly
unique.
```

Constraints:

6.5K 298

Code

Java Auto

```
class Solution {
    public int[] sortArray(int[] nums) {
        insertionSort(nums);
        return nums;
    }

    private void insertionSort(int[] nums) {
        int n = nums.length;
```

Saved

Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
nums =
[5,2,3,1]
```

Output

```
[1,2,3,5]
```

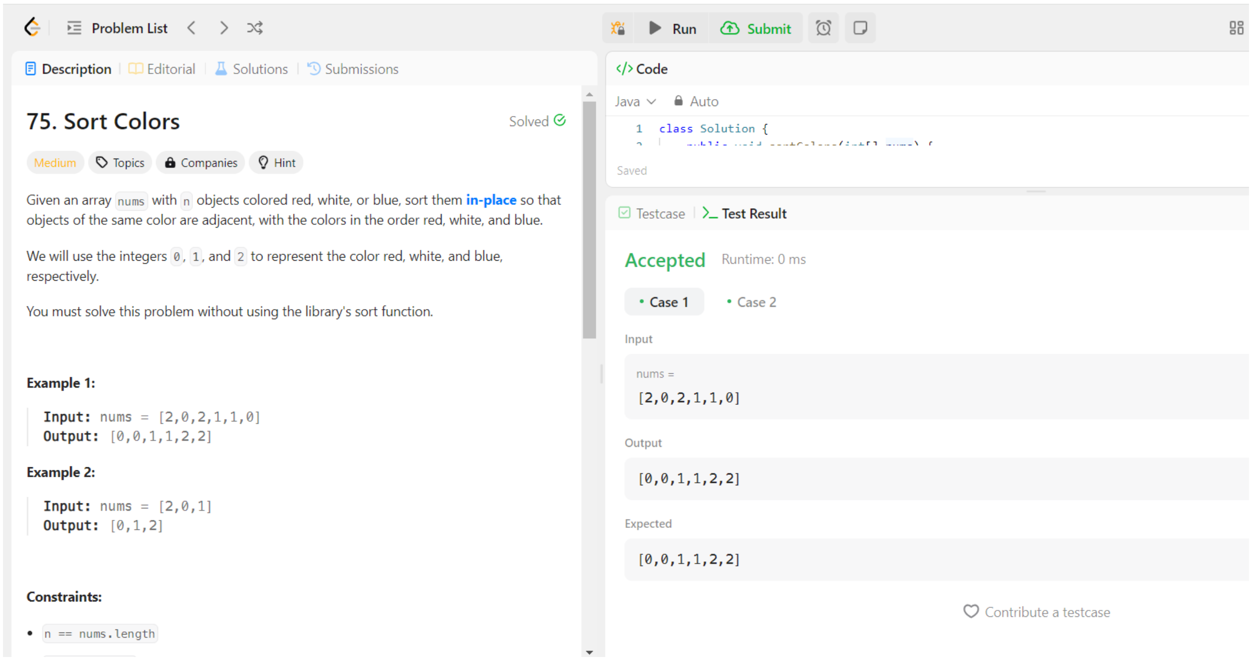
Expected

```
[1,2,3,5]
```

```
class Solution {
    public int[] sortArray(int[] nums) {
        insertionSort(nums);
        return nums;
    }

    private void insertionSort(int[] nums) {
        int n = nums.length;
        for (int i = 1; i < n; i++) {
            int key = nums[i];
            int j = i - 1;
            while (j >= 0 && nums[j] > key) {
                nums[j + 1] = nums[j];
```

```java
                j = j - 1;
            }
            nums[j + 1] = key;
        }
    }
}
```

Sort Colors using Selection Sort



Problem List

Description | Editorial | Solutions | Submissions

75. Sort Colors Solved

Medium · Topics · Companies · Hint

Given an array `nums` with `n` objects colored red, white, or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Example 1:

```
Input: nums = [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
```

Example 2:

```
Input: nums = [2,0,1]
Output: [0,1,2]
```

Constraints:

- `n == nums.length`

Run · Submit

Code · Java · Auto

```java
class Solution {
```

Saved

Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 · Case 2

Input

nums =

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

Output

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

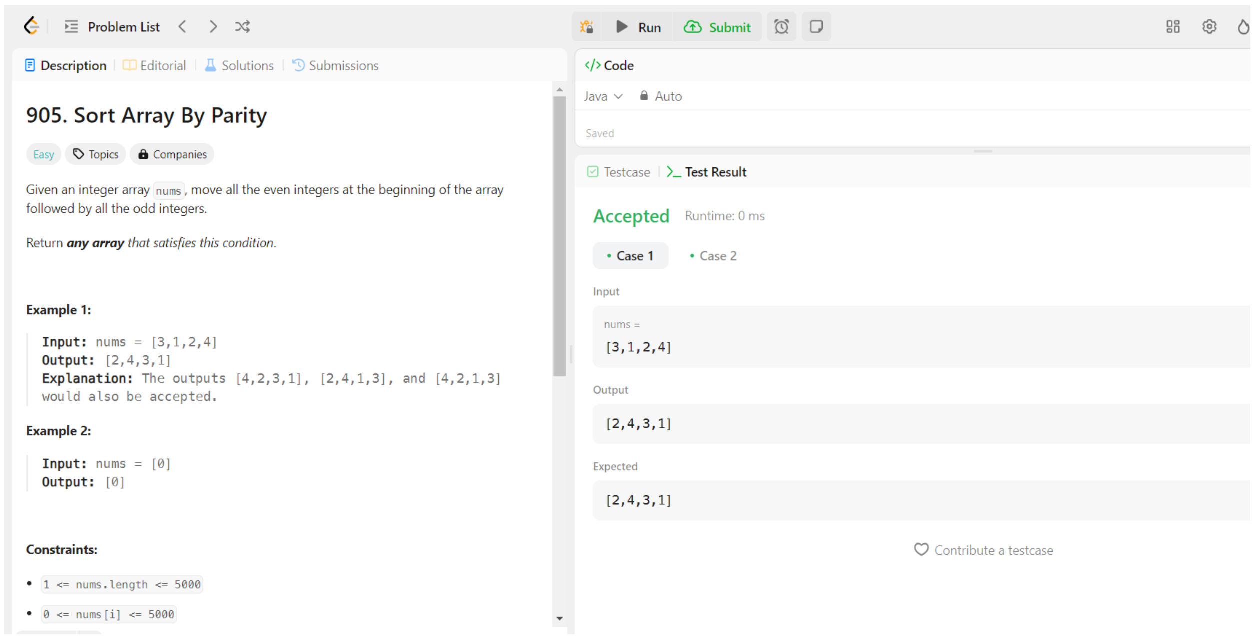
Expected

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

Contribute a testcase

```java
class Solution {
    public void sortColors(int[] nums) {
        selectionSort(nums);
    }

    public void selectionSort(int[] nums) {
        int n = nums.length;

        for (int i = 0; i < n - 1; i++) {
            int min_Index = i;
            for (int j = i + 1; j < n; j++) {
                if (nums[j] < nums[min_Index]) {
                    min_Index = j;
                }
            }
            int t = nums[min_Index];
            nums[min_Index] = nums[i];
            nums[i] = t;
        }
    }
}
```

Sort Array by Parity: Bubble Sort



905. Sort Array By Parity

Given an integer array `nums`, move all the even integers at the beginning of the array followed by all the odd integers.

Return ***any array*** *that satisfies this condition.*

Example 1:

```
Input: nums = [3,1,2,4]
Output: [2,4,3,1]
Explanation: The outputs [4,2,3,1], [2,4,1,3], and [4,2,1,3]
would also be accepted.
```

Example 2:

```
Input: nums = [0]
Output: [0]
```

Constraints:

- `1 <= nums.length <= 5000`
- `0 <= nums[i] <= 5000`

Accepted Runtime: 0 ms

Case 1 Case 2

Input

```
nums =
[3,1,2,4]
```

Output

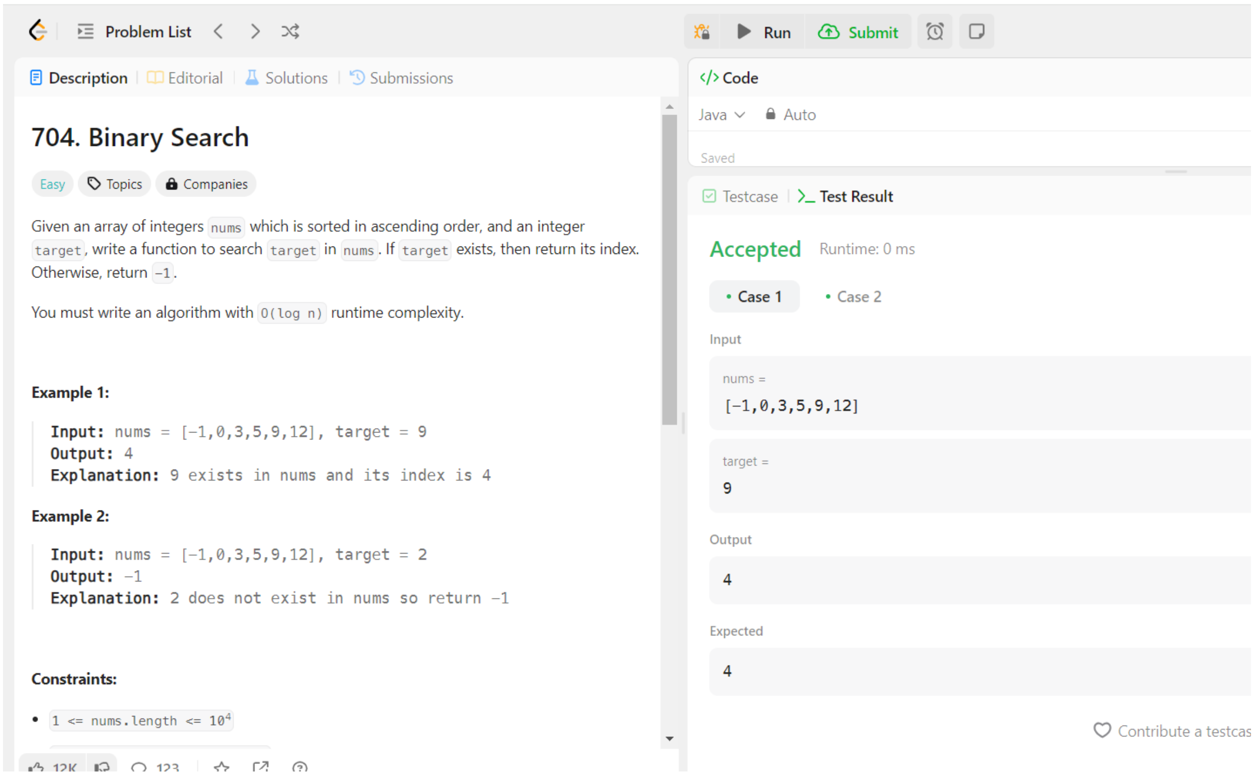
```
[2,4,3,1]
```

Expected

```
[2,4,3,1]
```

```
class Solution {
    public int[] sortArrayByParity(int[] nums) {
        bubbleSortByParity(nums);
        return nums;
    }

    public void bubbleSortByParity(int[] nums) {
        int n = nums.length;
        for (int i = 0; i < n - 1; i++) {
            for (int j = 0; j < n - i - 1; j++) {
                if (nums[j] % 2 != 0 && nums[j + 1] % 2 == 0) {
                    int t = nums[j];
                    nums[j] = nums[j + 1];
                    nums[j + 1] = t;
                }
            }
        }
    }
}
```

Binary Search:



Problem List

Description | Editorial | Solutions | Submissions

704. Binary Search

Easy · Topics · Companies

Given an array of integers `nums` which is sorted in ascending order, and an integer `target`, write a function to search `target` in `nums`. If `target` exists, then return its index. Otherwise, return `-1`.

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

Example 1:

```
Input: nums = [-1,0,3,5,9,12], target = 9
Output: 4
Explanation: 9 exists in nums and its index is 4
```

Example 2:

```
Input: nums = [-1,0,3,5,9,12], target = 2
Output: -1
Explanation: 2 does not exist in nums so return -1
```

Constraints:

- `1 <= nums.length <= 10^4`

Run · Submit

Code

Java · Auto

Saved

Testcase | Test Result

Accepted Runtime: 0 ms

Case 1 · Case 2

Input

```
nums =
[-1,0,3,5,9,12]
target =
9
```

Output

```
4
```

Expected

```
4
```

Contribute a testcas

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

            if (nums[mid] == target) {
                return mid;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return -1;
    }
}
```

1. Dijkstra's Algorithm:

The goal of the Dijkstra algorithm is to determine the shortest path in a weighted graph (with non-negative weights) between a source node and every other node.

Key steps:

Starting at the source node, mark all other nodes with an unlimited distance at first, and give it a tentative distance of 0.

Go to the node that has not been visited yet and has the shortest known distance. Update the estimated separations between its nearby nodes.
After updating all neighbors, designate the current node as visited, indicating that its shortest path has been verified.
Until every node has been visited or the shortest path to the destination node has been discovered, repeat the procedure for the next unexplored node with the minimum distance.

The time complexity is either O(V^2) when utilizing an adjacency matrix or O((V + E) log V) when utilizing a priority queue, such as a min-heap.

Use cases include network routing methods and navigation systems.

2. Bellman-Ford Algorithm:

Goal: The Bellman-Ford approach can also be used to determine the shortest path between a source node and every other node in a graph; however, it is only effective in graphs with negative edge weights.

Key Steps: Set the distances to all other nodes to infinity and the source node to 0 at initialization.
If a shorter path can be taken via the current edge than the known distance, update the destination node's distance for each edge.
V is the number of vertices. Repeat step (V-1) above a number of times.
Determine whether any more updates are possible in the last iteration and use that information to look for negative weight cycles. If so, a negative cycle can be seen in the graph.
Time Complexity: O(V * E), where V is the number of vertices and E is the number of edges.

3. DFS and BFS Graph Traversals: Breadth-First Search (BFS):

Level by level, BFS investigates the graph, beginning at the source node and examining each of its neighbors before proceeding to the next level.
It keeps track of which nodes to visit next via a queue.
BFS is the best method for locating the shortest path in graphs that are not weighted.
O(V + E) is the time complexity, where V is the number of vertices and E is the number of edges.
Depth-First Search (DFS):

Before turning around, DFS investigates a branch as far as it can.
Either directly or by recursion, it makes use of a stack.
DFS is helpful in solving tasks like graph cycle detection and topological sorting.
Time Complexity: O(V + E).

4. Divide and Conquer:

Goal: This algorithmic paradigm divides a problem into smaller subproblems, solves each one recursively, and then combines the solutions to solve the main problem.

Important Stages:

Divide: Divide the issue into more manageable subissues.
Conquer: Use recursive methods to solve the subproblems.
Combine: To solve the main problem, combine the answers to the subproblems.
Illustrations:

Quick Sort/Merge Sort
Use cases for binary searches: sorting, searching, and optimization challenges are examples of problems that can be solved effectively by

simplifying them.

5. The Fractional Knapsack Problem:

Goal: The objective of the greedy algorithm known as the Fractional Knapsack problem is to maximize the value of objects placed in a knapsack with a finite capacity while allowing fractions of items to be taken.

Important Stages:

Determine the value-to-weight ratio for every item.
This ratio will be used to sort the items in descending order.
Until the knapsack is filled, add stuff to it starting with the item with the highest ratio. Add as much of the thing (a fraction of the item) as feasible if it doesn't fit.
Due to item sorting, the time complexity is O(N log N), where N is the number of items.

Use Cases: Stock investments and container loading are two examples of resource allocation challenges where partial quantities of resources can be used.