ADS Assignment:1

Ankita Durgude

**Problem 1:** Given an array of integers, perform the following operations: 1. Find the second largest element in the array. 2. Move all zeros to the end of the array while maintaining the order of non-zero elements. Input: arr = [10, 0, 5, 20, 0, 8, 15] Output: Second largest element: 15 Array after moving zeros: [10, 5, 20, 8, 15, 0, 0] Constraints: ● Do not use built-in sort functions. ● The array may contain duplicate elements or zeros at any position. ● Array length ≥ 2.

import java.util.Arrays;

public class ArrayOperations {

public static int findSecondLargest(int[] arr) {

int largest = Integer.MIN\_VALUE;

int secondLargest = Integer.MIN\_VALUE;

for (int num : arr) {

if (num > largest) {

secondLargest = largest;

largest = num;

} else if (num > secondLargest && num != largest) {

secondLargest = num;

}

}

return secondLargest;

}

public static void moveZerosToEnd(int[] arr) {

int nonZeroIndex = 0;

for (int num : arr) {

if (num != 0) {

arr[nonZeroIndex++] = num;

}

}

while (nonZeroIndex < arr.length) {

arr[nonZeroIndex++] = 0;

}

}

public static void main(String[] args) {

int[] arr = {10, 0, 5, 20, 0, 8, 15};

int secondLargest = findSecondLargest(arr);

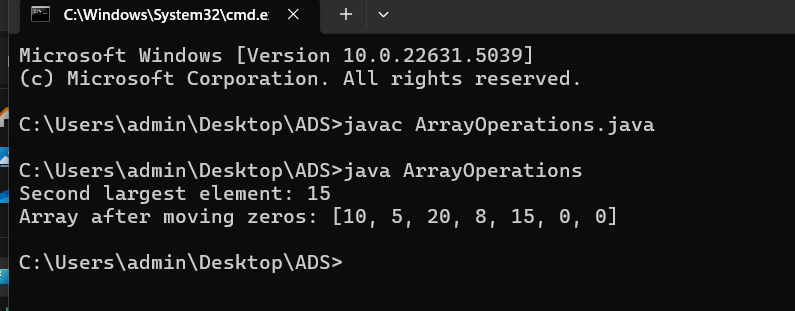
System.out.println("Second largest element: " + secondLargest);

moveZerosToEnd(arr);

System.out.println("Array after moving zeros: " + Arrays.toString(arr));

}

}



**Problem 2:** Write a program that performs the following operations on strings:

1. Check whether two given strings are anagrams of each other.

2. Identify the longest word in a given sentence.

3. Count the number of vowels and consonants in the same sentence.

**Input:**

String 1: listen

String 2: silent

Sentence: Practice makes a man perfect

**Output:**

Are 'listen' and 'silent' anagrams? true

Longest word: Practice

Vowels: 9, Consonants: 17

import java.util.Arrays;

public class StringOperations {

// Function to check if two strings are anagrams

public static boolean areAnagrams(String str1, String str2) {

char[] arr1 = str1.toCharArray();

char[] arr2 = str2.toCharArray();

Arrays.sort(arr1);

Arrays.sort(arr2);

return Arrays.equals(arr1, arr2);

}

// Function to find the longest word in a sentence

public static String findLongestWord(String sentence) {

String[] words = sentence.split(" ");

String longestWord = "";

for (String word : words) {

if (word.length() > longestWord.length()) {

longestWord = word;

}

}

return longestWord;

}

// Function to count vowels and consonants in a sentence

public static int[] countVowelsAndConsonants(String sentence) {

int vowels = 0, consonants = 0;

sentence = sentence.toLowerCase().replaceAll("[^a-z]", "");

for (char ch : sentence.toCharArray()) {

if ("aeiou".indexOf(ch) != -1) {

vowels++;

} else {

consonants++;

}

}

return new int[]{vowels, consonants};

}

public static void main(String[] args) {

String str1 = "listen";

String str2 = "silent";

String sentence = "Practice makes a man perfect";

System.out.println("Are '" + str1 + "' and '" + str2 + "' anagrams? " + areAnagrams(str1, str2));

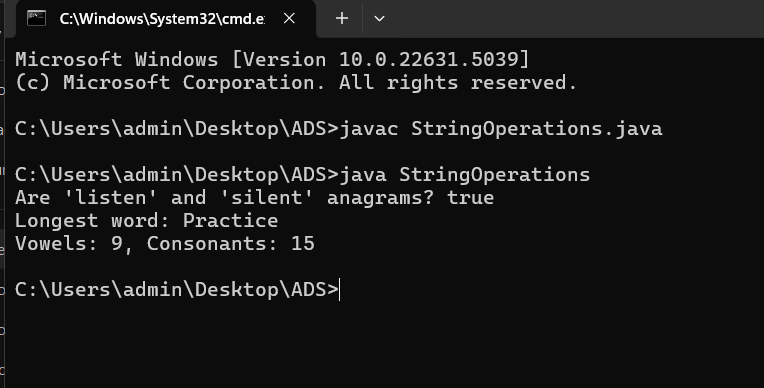
System.out.println("Longest word: " + findLongestWord(sentence));

int[] counts = countVowelsAndConsonants(sentence);

System.out.println("Vowels: " + counts[0] + ", Consonants: " + counts[1]);

}

}



**Problem 3:** Given a **sorted array of integers** (which may include duplicates), perform the following operations:

1. Search for a given key and return its index (if found) with Binary Search.

2. Find the first and last occurrence of the key in the array.

3. Count the total number of times the key appears.

4. Find any peak element in the array (an element greater than its neighbors).

**Input:**

arr = [1, 3, 3, 3, 5, 6, 8], key = 3

**Input for Peak Element:**

arr =[1, 2, 18, 4, 5, 0]

**Output:**

Key found at index: 2

First occurrence: 1

Last occurrence: 3

Total count of key: 3

Peak element: 18

class Solution {

// Binary Search to find any occurrence of key

public static int binarySearch(int[] arr, int key) {

int left = 0, right = arr.length - 1;

while (left <= right) {

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

if (arr[mid] == key)

return mid;

else if (arr[mid] < key)

left = mid + 1;

else

right = mid - 1;

}

return -1; // Key not found

}

// Find first occurrence of key

public static int firstOccurrence(int[] arr, int key) {

int left = 0, right = arr.length - 1, result = -1;

while (left <= right) {

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

if (arr[mid] == key) {

result = mid;

right = mid - 1; // Search left for first occurrence

} else if (arr[mid] < key) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return result;

}

// Find last occurrence of key

public static int lastOccurrence(int[] arr, int key) {

int left = 0, right = arr.length - 1, result = -1;

while (left <= right) {

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

if (arr[mid] == key) {

result = mid;

left = mid + 1; // Search right for last occurrence

} else if (arr[mid] < key) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return result;

}

// Count occurrences of key

public static int countOccurrences(int[] arr, int key) {

int first = firstOccurrence(arr, key);

if (first == -1) return 0; // Key not found

int last = lastOccurrence(arr, key);

return last - first + 1;

}

// Find any peak element

public static int findPeakElement(int[] arr) {

int left = 0, right = arr.length - 1;

while (left < right) {

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

if (arr[mid] > arr[mid + 1]) {

right = mid; // Move left to find peak

} else {

left = mid + 1; // Move right to find peak

}

}

return arr[left]; // Peak element

}

public static void main(String[] args) {

int[] arr = {1, 3, 3, 3, 5, 6, 8};

int key = 3;

int index = binarySearch(arr, key);

System.out.println("Key found at index: " + index);

System.out.println("First occurrence: " + firstOccurrence(arr, key));

System.out.println("Last occurrence: " + lastOccurrence(arr, key));

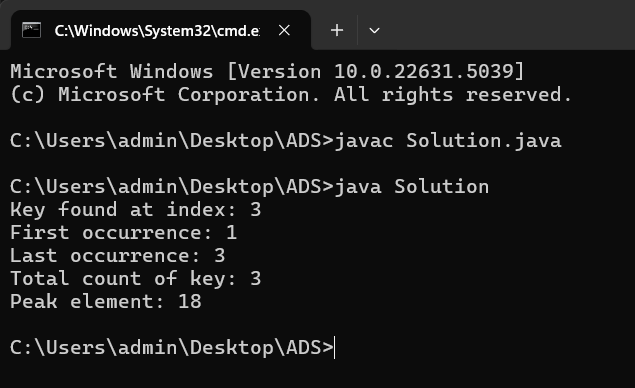
System.out.println("Total count of key: " + countOccurrences(arr, key));

int[] peakArr = {1, 2, 18, 4, 5, 0};

System.out.println("Peak element: " + findPeakElement(peakArr));

}

}



**Problem 4:**

Write a recursive program that performs the following operations:

1. **Check if a number is prime using recursion**.

2. Check whether a given string is a palindrome.

3. Find the sum of digits of a given number.

4. Calculate the nth Fibonacci number.

5. Calculate a raised to the power b

**Input:**

num = 7

str = "racecar"

num = 1234

fibIndex = 6

a = 2, b = 5

**Output:**

Is prime: true

Is 'racecar' a palindrome? true

Sum of digits of 1234: 10

Fibonacci(6): 8

2^5 = 32

**Constraints:**

● Do not use loops or built-in reverse methods.

● Use charAt() for string access.

● You can assume valid positive integer inputs.

public class RecursiveOperations {

// 1. Check if a number is prime

public static boolean isPrime(int num, int divisor) {

if (num <= 1) return false;

if (divisor == 1) return true;

if (num % divisor == 0) return false;

return isPrime(num, divisor - 1);

}

public static boolean isPrime(int num) {

return isPrime(num, num / 2);

}

// 2. Check if a string is a palindrome

public static boolean isPalindrome(String str, int start, int end) {

if (start >= end) return true;

if (str.charAt(start) != str.charAt(end)) return false;

return isPalindrome(str, start + 1, end - 1);

}

public static boolean isPalindrome(String str) {

return isPalindrome(str, 0, str.length() - 1);

}

// 3. Find the sum of digits of a number

public static int sumOfDigits(int num) {

if (num == 0) return 0;

return (num % 10) + sumOfDigits(num / 10);

}

// 4. Calculate the nth Fibonacci number

public static int fibonacci(int n) {

if (n <= 1) return n;

return fibonacci(n - 1) + fibonacci(n - 2);

}

// 5. Calculate a raised to the power b

public static int power(int a, int b) {

if (b == 0) return 1;

return a \* power(a, b - 1);

}

public static void main(String[] args) {

int num1 = 7;

String str = "racecar";

int num2 = 1234;

int fibIndex = 6;

int a = 2, b = 5;

System.out.println("Is prime: " + isPrime(num1));

System.out.println("Is '" + str + "' a palindrome? " + isPalindrome(str));

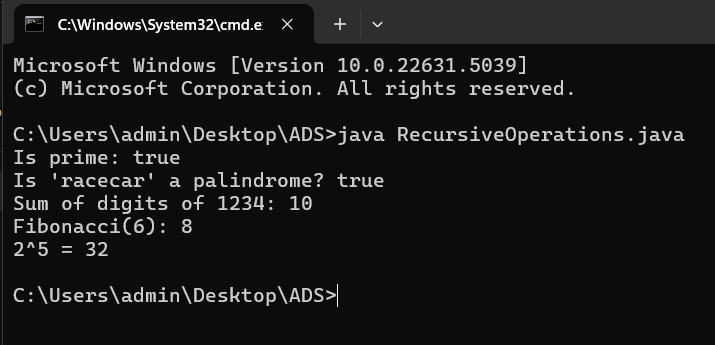
System.out.println("Sum of digits of " + num2 + ": " + sumOfDigits(num2));

System.out.println("Fibonacci(" + fibIndex + "): " + fibonacci(fibIndex));

System.out.println(a + "^" + b + " = " + power(a, b));

}

}



**Problem 5:**

**Dry Run & Analyze: Time and Space Complexity**

**1. Dry run the code for n = 4. How many times is \* printed? What is the time complexity?**

void printTriangle(int n) {

for (int i = 0; i < n; i++)

for (int j = 0; j <= i; j++)

System.out.print("\*");

}

### Ans:

1. **Total stars (\*) printed for n=4n = 4n=4**: **10**
2. **Time Complexity**: **O(n2)O(n^2)O(n2)**
3. **Space Complexity**: **O(1)O(1)O(1)**

**2. Dry run for n = 8. What’s the number of iterations? Time complexity?**

void printPattern(int n) {

for (int i = 1; i <= n; i \*= 2)

for (int j = 0; j < n; j++)

System.out.println(i + "," + j);

}

Ans:

1 **Total number of iterations for n=8n = 8n=8**: **32**

2 **Time Complexity**: **O(nlog⁡n)O(n \log n)O(nlogn)**

3 **Space Complexity**: **O(1)O(1)O(1)**

**3. Dry run for n = 20. How many recursive calls? What values are printed?**

void recHalf(int n) {

if (n <= 0) return;

System.out.print(n + " ");

recHalf(n / 2);

}

1 **Total recursive calls for n=20n = 20n=20**: **6**

2 **Printed Values**: **20 10 5 2 1**

**4. Dry run for n = 3. How many total calls are made? What’s the time complexity?**

void fun(int n) {

if (n == 0) return;

fun(n - 1);

fun(n - 1);

}

Ans:

1 total number of calls is **15**

2 the **time complexity** is: O(2n)O(2^n)O(2n)

**5. Dry run for n = 3. How many total iterations? Time complexity?**

void tripleNested(int n) {

for (int i = 0; i < n; i++)

for (int j = 0; j < n; j++)

for (int k = 0; k < n; k++)

System.out.println(i + j + k);

}

Ans:

1. the **total number of iterations is 27**.
2. **Time complexity** O(n3)