**Nishant Patil**

**PG-DAC**

**ADS**

**Assignment 1**

**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.

public class Main {

public static int findSecondLargest(int[] arr) {

int first = Integer.MIN\_VALUE, second = Integer.MIN\_VALUE;

for (int num : arr) {

if (num > first) {

second = first;

first = num;

} else if (num > second && num < first) {

second = num;

}

}

return second;

}

public static void moveZerosToEnd(int[] arr) {

int nonZero = 0;

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

if (arr[i] != 0) {

arr[nonZero++] = arr[i];

}

}

while (nonZero < arr.length) {

arr[nonZero++] = 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.print("Array after moving zeros: ");

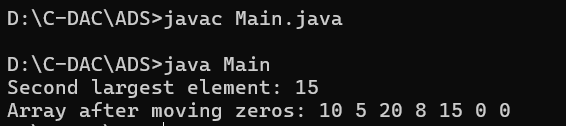
for (int num : arr) {

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

}

}

}



**Problem**

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.\*;

public class Main {

//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);

}

//longest word

public static String longestWord(String sentence) {

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

String longest = "";

for (String word : words) {

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

longest = word;

}

}

return longest;

}

// vowels and consonants

public static void countVowelsAndConsonants(String sentence) {

int vowels = 0, consonants = 0;

String low = sentence.toLowerCase();

for (char c : low.toCharArray()) {

if (Character.isLetter(c)) {

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

vowels++;

} else {

consonants++;

}

}

}

System.out.println("Vowels: " + vowels + ", Consonants: " + consonants);

}

public static void main(String[] args) {

String str1 = "listen";

String str2 = "silent";

String sentence = "Practice makes a man perfect";

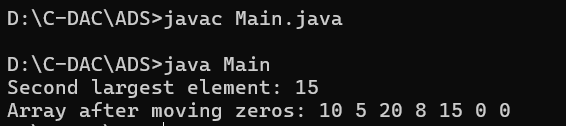
System.out.println("Are '" + str1 + "' and '" + str2 + "' anagrams? " + areAnagrams(str1, str2));

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

countVowelsAndConsonants(sentence);

}

}



**Problem**

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: 3

First occurrence: 1

Last occurrence: 3

Total count of key: 3

Peak element: 18

public class Main {

// Binary Search

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

int low = 0, high = arr.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (arr[mid] == key) {

return mid;

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

low = mid + 1;

} else {

high = mid - 1;

}

}

return -1;

}

//first occurrence

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

int low = 0, high = arr.length - 1, result = -1;

while (low <= high) {

int mid = (low + high) / 2;

if (arr[mid] == key) {

result = mid;

high = mid - 1;

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

low = mid + 1;

} else {

high = mid - 1;

}

}

return result;

}

// last occurrence

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

int low = 0, high = arr.length - 1, result = -1;

while (low <= high) {

int mid = (low + high) / 2;

if (arr[mid] == key) {

result = mid;

low = mid + 1;

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

low = mid + 1;

} else {

high = mid - 1;

}

}

return result;

}

// Count occurrences

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

int first = FirstOccurrence(arr, key);

if (first == -1) return 0;

int last = LastOccurrence(arr, key);

return last - first + 1;

}

//peak

public static int PeakElement(int[] arr) {

if (arr.length == 1) return arr[0];

if (arr[0] > arr[1]) return arr[0];

if (arr[arr.length - 1] > arr[arr.length - 2]) return arr[arr.length - 1];

for (int i = 1; i < arr.length - 1; i++) {

if (arr[i] > arr[i - 1] && arr[i] > arr[i + 1]) {

return arr[i];

}

}

return -1;

}

public static void main(String[] args) {

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

int key = 3;

// Binary search for key

int keyIndex = binarySearch(arr, key);

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

// First and last occurrence of key

int first = FirstOccurrence(arr, key);

int last = LastOccurrence(arr, key);

System.out.println("First occurrence: " + first);

System.out.println("Last occurrence: " + last);

// Total count of key

int totalCount = countOccurrences(arr, key);

System.out.println("Total count of key: " + totalCount);

// Find peak element

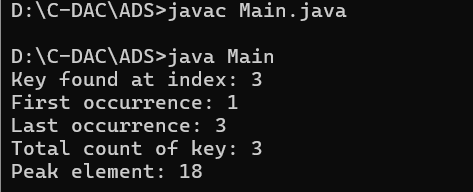
int[] peakArr = {1, 2, 18, 4, 5, 0};

int peak = PeakElement(peakArr);

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

}

}



**Problem**

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 Main {

//prime

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

if (n <= 1) return false;

if (divisor == 1) return true;

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

return isPrime(n, divisor - 1);

}

//palindrome

public static boolean isPalindrome(String s, int left, int right) {

if (left >= right) return true;

if (s.charAt(left) != s.charAt(right)) return false;

return isPalindrome(s, left + 1, right - 1);

}

//Sum

public static int sumOfDigits(int n) {

if (n == 0) return 0;

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

}

//Fibonacci

public static int fibonacci(int n) {

if (n <= 1) return n;

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

}

//Power

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 num = 7;

String str = "racecar";

int numForDigits = 1234;

int fibIndex = 6;

int a = 2, b = 5;

boolean primeCheck = isPrime(num, num / 2);

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

boolean palindromeCheck = isPalindrome(str, 0, str.length() - 1);

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

int digitSum = sumOfDigits(numForDigits);

System.out.println("Sum of digits of " + numForDigits + ": " + digitSum);

int fibResult = fibonacci(fibIndex);

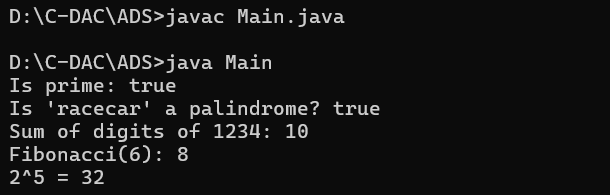
System.out.println("Fibonacci(" + fibIndex + "): " + fibResult);

int powerResult = power(a, b);

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

}

}



**Problem**

**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("\*");

}

**Dry Run for n = 4:**

* The outer loop (i) will iterate from 0 to 3 (since i < n).
* The inner loop (j) will iterate from 0 to i (i.e., j <= i).

**Iteration Breakdown:**

1. **i = 0:** \* is printed once.
2. **i = 1:** \* is printed twice.
3. **i = 2:** \* is printed three times.
4. **i = 3:** \* is printed four times.

**Output:**

* The total number of \* 1 + 2 + 3 + 4 = 10.

**Time Complexity:**

* Total number of iterations: 1+2+3+...+n
* sum is n(n+1)/2
* Therefore, the **time complexity** is **O(n^2)**.

**Space Complexity:**

* The space complexity is **O(1)** because we are using only a constant amount of space, regardless of the input size.

1. **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);

}

**Dry Run for n = 8:**

* The outer loop (i) starts at 1 and doubles each time (i \*= 2).
* The inner loop (j) runs n times for each value of i.

**Iteration Breakdown:**

1. **i = 1 (outer loop runs once):** (1,0), (1,1), (1,2), ..., (1,7) are printed (8 lines).
2. **i = 2 (outer loop runs once):** (2,0), (2,1), (2,2), ..., (2,7) are printed (8 lines).
3. **i = 4 (outer loop runs once):** (4,0), (4,1), (4,2), ...,( 4,7 )are printed (8 lines).
4. **i = 8 (outer loop runs once):** (8,0), (8,1), (8,2), ..., (8,7) are printed (8 lines).

* The outer loop will run until i exceeds n (i.e., i will take the values 1, 2, 4, and 8).

**Output:**

* The inner loop runs 8 times for each of 4 iterations of the outer loop, so the total number of printed lines is 4×8=32.

**Time Complexity:**

* The outer loop runs log n times.
* The inner loop runs n times for each iteration of the outer loop.
* Thus, the total number of iterations is n×log n.
* The **time complexity** is **O(n log n)**.

**Space Complexity:**

* The space complexity is **O(1)** since only a constant amount of space is used, regardless of the input size.

1. **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);

}

**Dry Run for n = 20:**

**Initial Call:**

* n = 20
* **First call (n = 20):**
  + Print 20.
  + Call recHalf(20 / 2) → recHalf(5).
* **Second call (n = 10):**
  + Print 10.
  + Call recHalf(10 / 2) → recHalf(5).

1. **Third call (n = 5):**
   * Print 5.
   * Call recHalf(5 / 2) → recHalf(2).
2. **Fourth call (n = 2):**
   * Print 2.
   * Call recHalf(2 / 2) → recHalf(1).
3. **Fifth call (n = 1):**
   * Print 1.
   * Call recHalf(1 / 2) → recHalf(0).
4. **Base case (n = 0):**
   * Since n <= 0, the function returns without printing anything.

**Output:**

* The printed values are:20 10 5 2 1.

**Time Complexity:**

* The function makes a recursive call with n / 2 at each step. This is logarithmic, specifically log n
* Thus, the **time complexity** is **O(log n)**.

**Space Complexity:**

* The space complexity is **O(log n)** because each recursive call adds a new frame to the call stack, and the depth of the recursion is proportional to log⁡2(n)\log\_2(n)log2​(n).

1. **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); }

**Dry Run for n = 3:**

1. **Call 1: fun(3)**
   * n != 0, so it proceeds to call fun(2) twice.
2. **Call 2: fun(2)** (first call from fun(3))
   * n != 0, so it proceeds to call fun(1) twice.
3. **Call 3: fun(1)** (first call from fun(2))
   * n != 0, so it proceeds to call fun(0) twice.
4. **Call 4: fun(0)** (first call from fun(1))
   * n == 0, so it returns immediately (base case).
5. **Call 5: fun(0)** (second call from fun(1))
   * n == 0, so it returns immediately (base case).
6. **Return to Call 3: fun(1)** (second call from fun(2))
   * n != 0, so it proceeds to call fun(0) twice.
7. **Call 6: fun(0)** (first call from fun(1))
   * n == 0, so it returns immediately (base case).
8. **Call 7: fun(0)** (second call from fun(1))
   * n == 0, so it returns immediately (base case).
9. **Return to Call 2: fun(2)** (second call from fun(3))
   * n != 0, so it proceeds to call fun(1) twice.
10. **Call 8: fun(1)** (first call from fun(2))
    * n != 0, so it proceeds to call fun(0) twice.
11. **Call 9: fun(0)** (first call from fun(1))
    * n == 0, so it returns immediately (base case).
12. **Call 10: fun(0)** (second call from fun(1))
    * n == 0, so it returns immediately (base case).
13. **Return to Call 8: fun(1)** (second call from fun(2))
    * n != 0, so it proceeds to call fun(0) twice.
14. **Call 11: fun(0)** (first call from fun(1))
    * n == 0, so it returns immediately (base case).
15. **Call 12: fun(0)** (second call from fun(1))
    * n == 0, so it returns immediately (base case).

**Total Number of Calls:**

* **1 call** for fun(3)
* **2 calls** for fun(2)
* **4 calls** for fun(1)
* **8 calls** for fun(0)

In total, there are **15 calls** made.

**General Time Complexity:**

* The recurrence relation for this function is:

T(n)=2T(n−1)+ 1

The solution to this recurrence is T(n)=2n−1, so the **time complexity** is **O(2^n)**.

**Space Complexity:**

* The space complexity depends on the maximum depth of the recursion stack.
* In the worst case, the recursion goes to depth n (when n is not zero), so the **space complexity** is **O(n)**.

1. **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);

}

**Dry Run for n = 3:**

This code consists of three nested loops, each running from 0 to n-1 (i.e., 0 to 2 for n = 3).

**Iteration Breakdown:**

* The outer loop (with i) runs 3 times (i = 0, 1, 2).
* The middle loop (with j) runs 3 times for each iteration of i (j = 0, 1, 2).
* The inner loop (with k) runs 3 times for each iteration of j (k = 0, 1, 2).

For each combination of i, j, and k, the statement System.out.println(i + j + k) is executed.

**Iterations:**

1. **i = 0:**
   * **j = 0:**
     + **k = 0**: i + j + k = 0 + 0 + 0 = 0
     + **k = 1**: i + j + k = 0 + 0 + 1 = 1
     + **k = 2**: i + j + k = 0 + 0 + 2 = 2
   * **j = 1:**
     + **k = 0**: i + j + k = 0 + 1 + 0 = 1
     + **k = 1**: i + j + k = 0 + 1 + 1 = 2
     + **k = 2**: i + j + k = 0 + 1 + 2 = 3
   * **j = 2:**
     + **k = 0**: i + j + k = 0 + 2 + 0 = 2
     + **k = 1**: i + j + k = 0 + 2 + 1 = 3
     + **k = 2**: i + j + k = 0 + 2 + 2 = 4
2. **i = 1:**
   * **j = 0:**
     + **k = 0**: i + j + k = 1 + 0 + 0 = 1
     + **k = 1**: i + j + k = 1 + 0 + 1 = 2
     + **k = 2**: i + j + k = 1 + 0 + 2 = 3
   * **j = 1:**
     + **k = 0**: i + j + k = 1 + 1 + 0 = 2
     + **k = 1**: i + j + k = 1 + 1 + 1 = 3
     + **k = 2**: i + j + k = 1 + 1 + 2 = 4
   * **j = 2:**
     + **k = 0**: i + j + k = 1 + 2 + 0 = 3
     + **k = 1**: i + j + k = 1 + 2 + 1 = 4
     + **k = 2**: i + j + k = 1 + 2 + 2 = 5
3. **i = 2:**
   * **j = 0:**
     + **k = 0**: i + j + k = 2 + 0 + 0 = 2
     + **k = 1**: i + j + k = 2 + 0 + 1 = 3
     + **k = 2**: i + j + k = 2 + 0 + 2 = 4
   * **j = 1:**
     + **k = 0**: i + j + k = 2 + 1 + 0 = 3
     + **k = 1**: i + j + k = 2 + 1 + 1 = 4
     + **k = 2**: i + j + k = 2 + 1 + 2 = 5
   * **j = 2:**
     + **k = 0**: i + j + k = 2 + 2 + 0 = 4
     + **k = 1**: i + j + k = 2 + 2 + 1 = 5
     + **k = 2**: i + j + k = 2 + 2 + 2 = 6

**Total Number of Iterations:**

* The total number of iterations is n×n×n=n^3
* For n = 3, the number of iterations is 3×3×3=27
* **Time Complexity:**
* The time complexity is the number of iterations in all the loops, which is O(n^3)

**Space Complexity:**

* The space complexity is **O(1)** since no extra space is used beyond the loop variables.