ADS Assignment

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.**

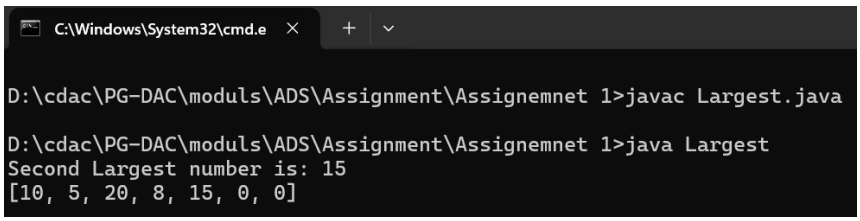
Ans:

Input:

```
import java.util.Arrays;
class Largest{
	public static void main(String[] args){
		int arr[]={10, 0, 5, 20, 0, 8, 15};
		int max=0;
		int max2=0;
		for(int i =0; i<arr.length;i++)
		{
			if(arr[i]>max)
			{
				max=arr[i];
			}
```

```
            }
            //System.out.println("largest:"+max);
            for(int i =0; i<arr.length;i++)
            {
                if(arr[i]>max2 && arr[i]<max)
                {
                    max2=arr[i];
                }
            }

            System.out.println("Second Largest number is: "+max2);
            int n = arr.length;
        int index = 0;  // Position to place non-zero elements
        for (int i = 0; i < n; i++) {
            if (arr[i] != 0) {
                int temp = arr[i];
                arr[i] = arr[index];
                arr[index] = temp;
                index++;
            }
        }
            System.out.println(Arrays.toString(arr));

        }
}
```

Output:



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

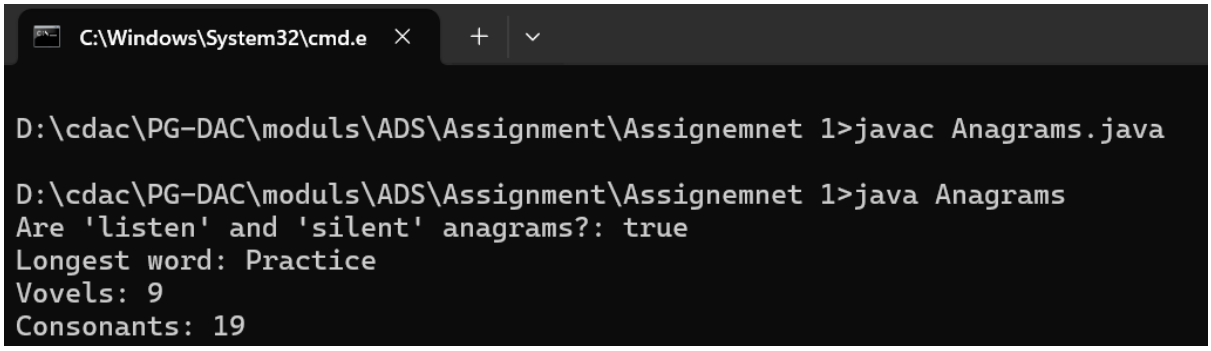
Ans:
Input:

```
import java.util.Arrays;

class Anagrams{

    static boolean isAnagram(String a, String b) {

    a = a.toLowerCase();

    b = b.toLowerCase();


    if (a.length() != b.length())

    {

       return false;

    }

    char[] arrA = a.toCharArray();

    char[] arrB = b.toCharArray();

    Arrays.sort(arrA);

    Arrays.sort(arrB);
```

```
    for (int i = 0; i < arrA.length; i++) {
        if (arrA[i] != arrB[i]) {
            return false;
        }
    }
    return true;
}
        static void isVowels(String s){
                char[] arrS = s.toCharArray();
                int count=0;
                int n=arrS.length;
                for (int i = 0; i < arrS.length; i++) {
        if (arrS[i]=='a'||arrS[i]=='e'||arrS[i]=='i'||arrS[i]=='o'||arrS[i]=='u'){
            count++;
        }
    }
                int Consonants= n-count;
                System.out.println("Vovels: "+count);
                System.out.println("Consonants: "+ Consonants);
        }
        static void longest(String s) {
    String[] words = s.split(" ");
    String longestWord = "";

    for (String word : words) {
        if (word.length() > longestWord.length()) {
            longestWord = word;
        }
```

```
    }
            System.out.println("Longest word: "+longestWord);
        }
        public static void main(String[] args){
            String a="listen";
            String b="silent";
            String s="Practice makes a man perfect";
            boolean result=isAnagram(a,b);
            System.out.println("Are 'listen' and 'silent' anagrams?: "+result);
            longest(s);
            isVowels(s);
        }
}
```

Output:



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

Ans:
Input:

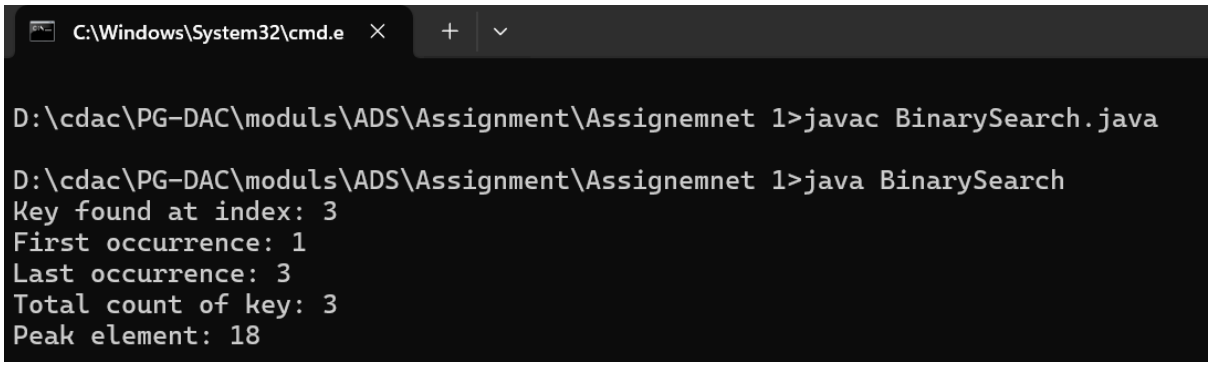
```
import java.util.Arrays;


class BinarySearch {
    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);


        // First and Last Occurrence
        int firstOccurrence = findFirstOccurrence(arr, key);
        int lastOccurrence = findLastOccurrence(arr, key);
        System.out.println("First occurrence: " + firstOccurrence);
        System.out.println("Last occurrence: " + lastOccurrence);


        // Total count of the key
        int totalCount = (firstOccurrence == -1) ? 0 : (lastOccurrence - firstOccurrence + 1);
        System.out.println("Total count of key: " + totalCount);


        // Input for Peak Element
```

```
    int[] peakArr = {1, 2, 18, 4, 5, 0};
    int peakElement = findPeakElement(peakArr);
    System.out.println("Peak element: " + peakElement);
}

public static int binarySearch(int[] arr, int key) {
    int low = 0, high = arr.length - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key) {
            return mid;
        } else if (arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1; // Key not found
}

// Find First Occurrence
public static int findFirstOccurrence(int[] arr, int key) {
    int low = 0, high = arr.length - 1, result = -1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key) {
            result = mid;
            high = mid - 1; // Search in the left half
        } else if (arr[mid] < key) {
```

```
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return result;
}

// Find Last Occurrence
public static int findLastOccurrence(int[] arr, int key) {
    int low = 0, high = arr.length - 1, result = -1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key) {
            result = mid;
            low = mid + 1; // Search in the right half
        } else if (arr[mid] < key) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return result;
}

// Find Peak Element
public static int findPeakElement(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        if ((i == 0 || arr[i] > arr[i - 1]) && (i == arr.length - 1 || arr[i] > arr[i + 1])) {
```

```
            return arr[i];
        }
    }
    return -1; // No peak element (for edge cases)
  }
}
```

Output:



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.

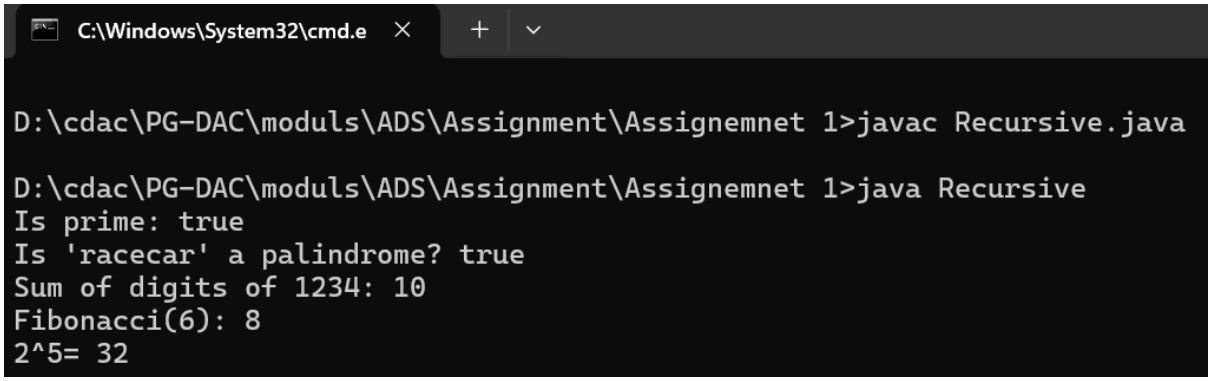
Ans:
Input:

```
class Recursive{
	static boolean isPrime(int n, int i){
		if(i==1)
		{
			return true;
		}
		if(n%i==0)
		{
			return false;
		}

		return isPrime(n,i-1);
   }

	static boolean isPalindrome(String str){
		int n = str.length();
		if (n == 0)
			return true;
		return isPalRec(str, 0, n - 1);
	}
```

```
static boolean isPalRec(String str, int s, int e){
        if (s == e)
        {
                return true;
        }
        if ((str.charAt(s)) != (str.charAt(e))){
                return false;
        }
        if (s < e + 1){
                return isPalRec(str, s + 1, e - 1);
        }
        return true;
}

static int SumDigit(int n){
        if(n==0){
                return 0;
        }
        return(n%10+ SumDigit(n/10));
}

static int Fibonacci(int n){
        if(n==0){
                return 0;
        }
        if(n==1){
           return 1;
        }
```

```
            return Fibonacci(n-1)+Fibonacci(n-2);
        }

        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;
            int n=1234;
            String str = "racecar" ;
            int index=6;
            int a=2;
            int b=5;

            System.out.println("Is prime: "+ isPrime(num, 2));
      System.out.println("Is 'racecar' a palindrome? "+isPalindrome(str));
            System.out.println("Sum of digits of 1234: "+ SumDigit(n));
            System.out.println("Fibonacci(6): "+Fibonacci(6));
            System.out.println("2^5= "+power(a,b));
        }
}
```

Output:



Problem 5:

Dry Run & Analyze: Time and Space Complexity

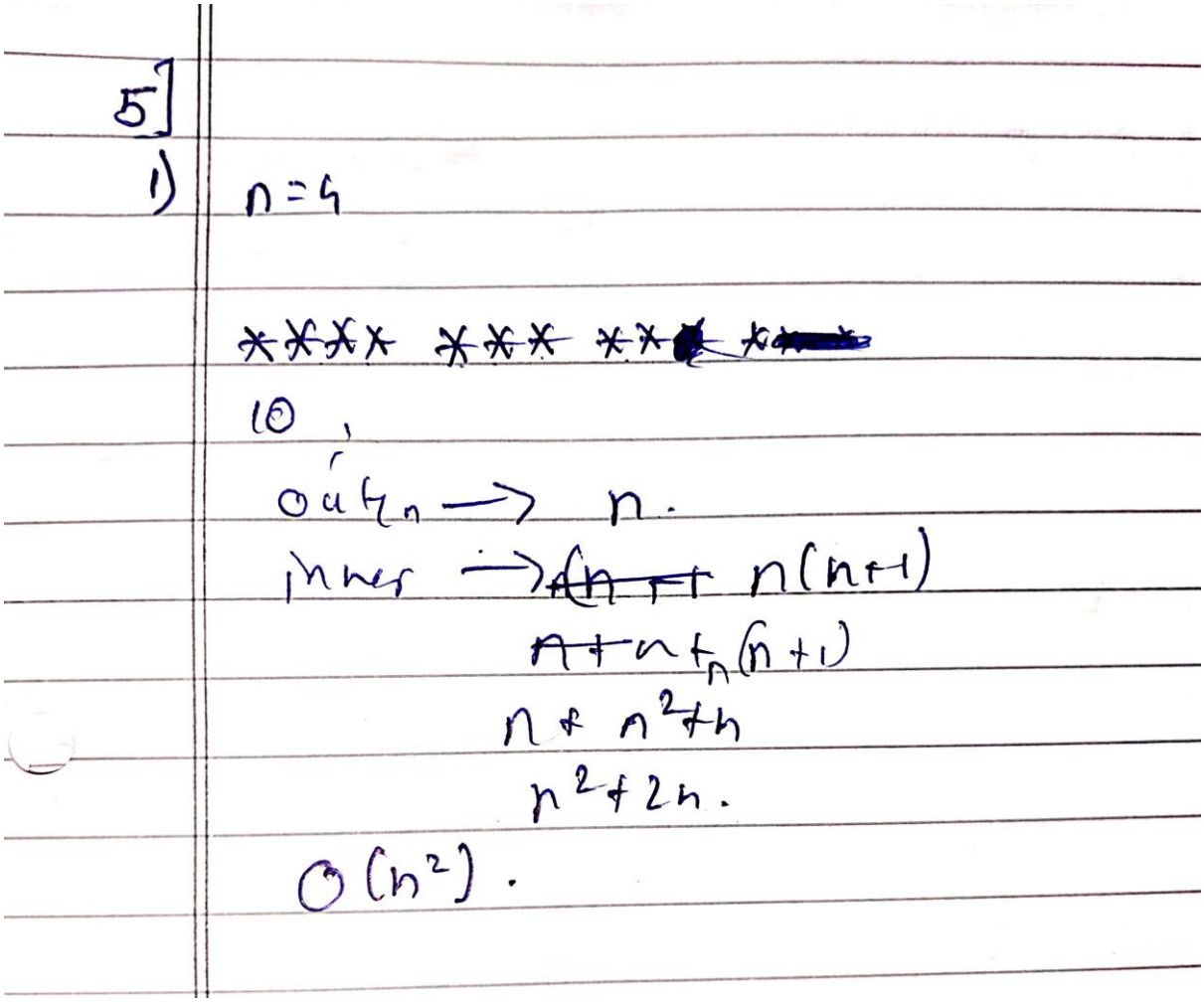
1. Dry run the code for n = 4. How many times is * printed? What is the time complexity?

```
void printTriangle(int n) {
for (int i = 0; i < n; i++)
for (int j = 0; j <= i; j++)
System.out.print("*");
}
```

Ans:

It will print ********** i.e * will print 10 times;

Time complexity is O(n^2)

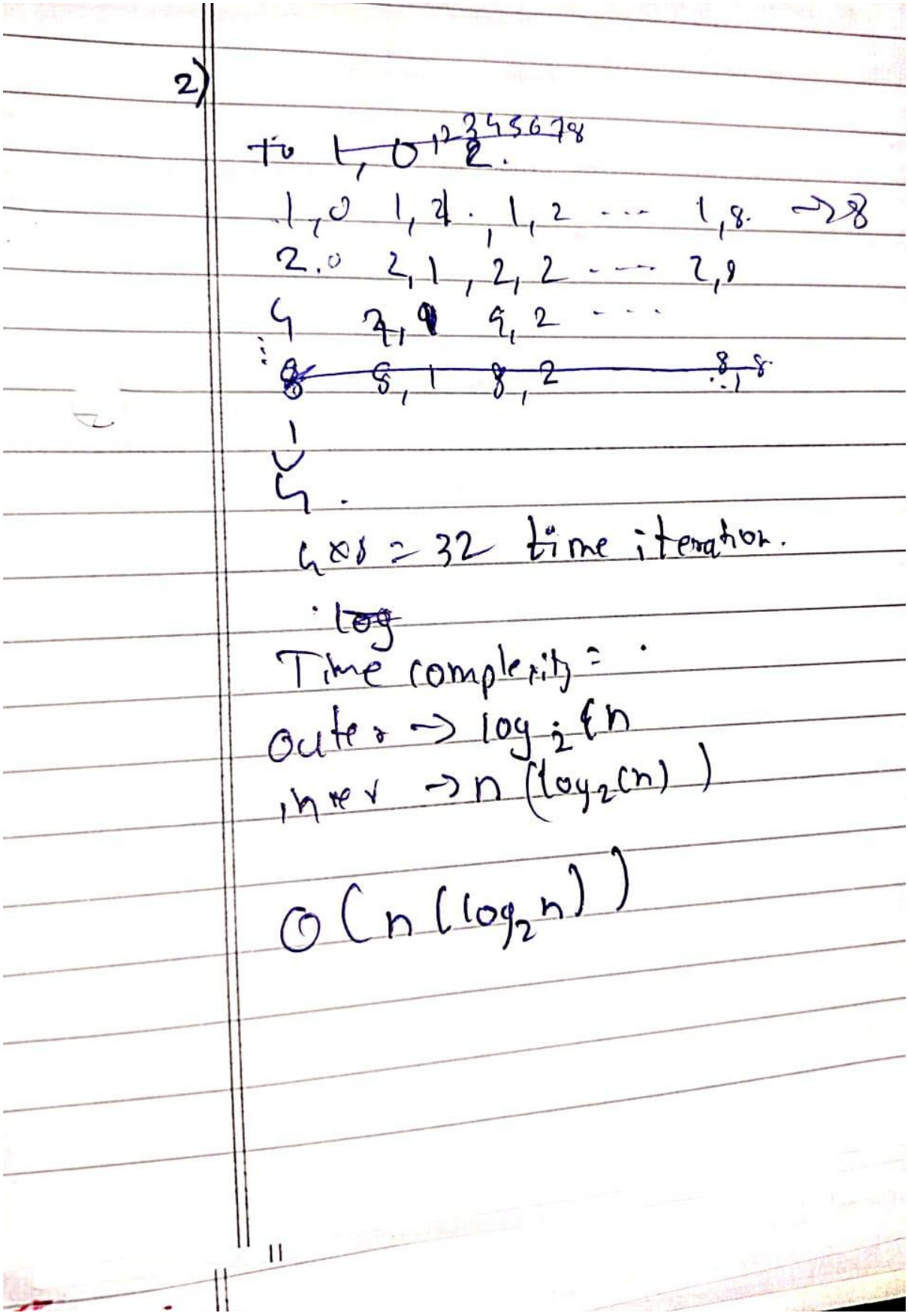


2. Dry run for n = 8. What's the number of iterations? Time complexity?

```
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:

32 iterations



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

void recHalf(int n) {

if (n <= 0) return;

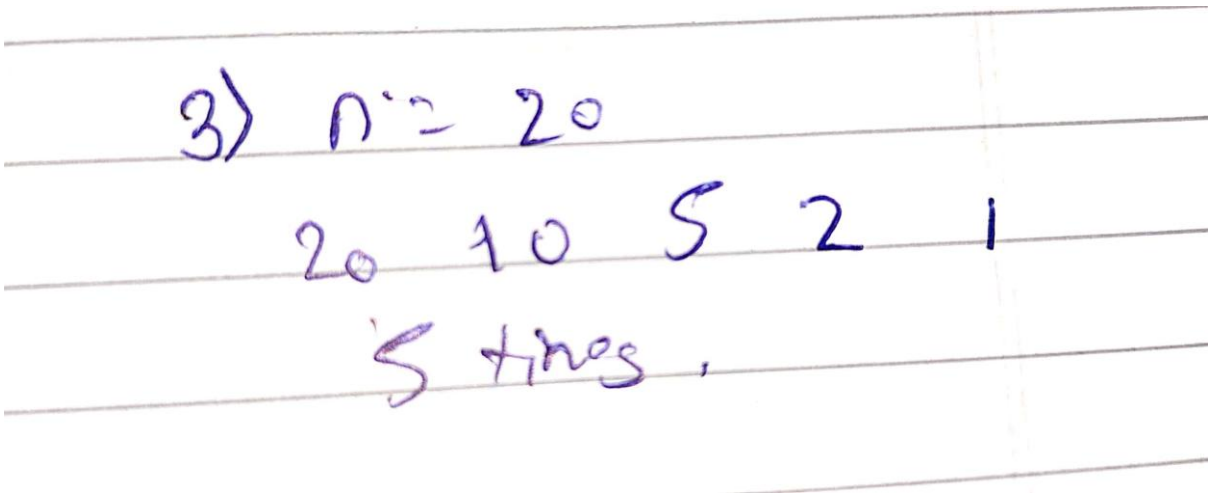
System.out.print(n + " ");

recHalf(n / 2);

}

Ans:

5 recursive calls



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

void fun(int n) {

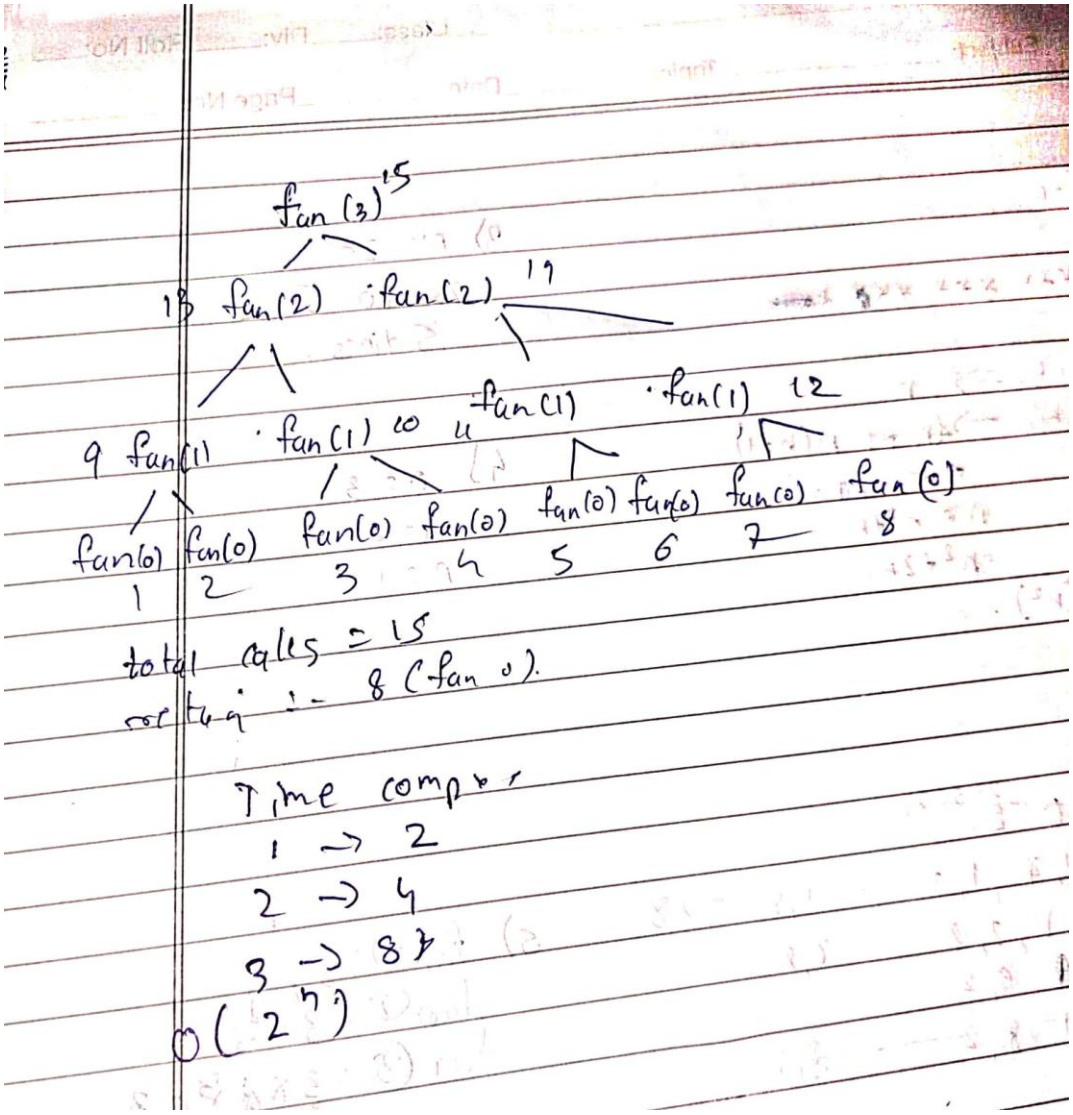
if (n == 0) return;

fun(n - 1);

fun(n - 1);

}

Ans:



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:

total 72 iterations

5) for ① = 3 (i)
for ② = 3 × 3 (i × j)
for ③ = 3 × 3 × 3 (i × j × k) = 27.

27 : iteration.

n × n × n

O(n3)