Question 1: MCQs

1. Which of the following statements is/are true?
   A. Binary search runs in O(log n) and faster than linear search
   B. Insertion sort runs in $O(n^2)$ and is thus faster than merge sort that runs in O(nlogn)
   C. Merge sort runs in $O(n^2)$ and is thus faster than insertion sort that runs in O(nlogn)
   D. Merger sort runs in O(nlogn) and is thus faster than insertion sort that runs in $O(n^2)$

2. The Ω notation in asymptotic evaluation represents
   A. Lower bound on a function
   B. Upper bound on a function
   C. Average bound on a function
   D. Worst bound on a function

3. If the array is already sorted, merge sort runs in
   A. O(1)
   B. O(logn)
   C. O(nlogn)
   D. O(n)
   E. $O(n^2)$

4. The problem is recursively divided into exactly 2 subproblems in Divide and Conquer
   A. True
   B. False

5. The main novelty in Strassen's algorithms comes from
   A. Having n^3 levels in recursion tree
   B. Having fewer subproblems at each level in the recursion tree
   C. Having n^2 levels in recursion tree
   D. Dividing the problem into 4 subproblems instead of 8

Question 2: Solve these recurrences using Master's theorem and indicate which CASE applies

$$\begin{cases} O(n^d) \text{ if } a<b^d \\ O(n^{\log_b(a)}) \text{ if } a>b^d \\ O(n^d \log(n)) \text{ if } a=b^d \end{cases}$$

1- T(n) = 4T (n/2) + n^2

So the solution becomes T(n) = O(n^2log n).

2- T(n) = 16T (n/4) + n

So the solution becomes T(n) = O(n^2).

3- T (n) = 64T (n/8) − n^2 log n

Explanation: The given recurrence cannot be solved by using the Master's theorem.

4- T (n) = 3T (n/3) + $\sqrt{n}$

T(n) = O(n).

5- What will be the recurrence relation of the following code?

```
int sum(int n)
{
   If(n==1)
        return 1;
   else
        return n+sum(n-1);
}
```

a) T(n) = T(n/2) + n
b) T(n) = T(n-1) + n **Recurrence relation: T(n) = _____**
c) T(n) = T(n-1) + O(1) **Case? _____**
d) T(n) = T(n/2) + O(1)

Answer: c

Explanation: As after every recursive call the integer up to which the sum is to be calculated decreases by 1. So the recurrence relation for the given code will be T(n) = T(n-1) + O(1).

Question 3: Runtime Analysis Techniques

Find the runtime of the following recurrence equation using the below three techniques:

$$\mathbf{T(n) = 9T(n/3) + n^2 \text{ for } n >= 2}$$

$$\mathbf{T(1) = 0}$$

a) Iterative Substitution

$T(n) = 9T(n/3) + n^2$

$= 9\,(9T(n/3^2) + (n/3)^3) + n^2$

$= 9^2\,(9T(n/3^3) + (n/3^2)^3) + 2n^2$

…

$= 9^k\,T(n/3^k) + kn^2$

For $k = \log_3 n = 9^{\log_3 n}\,T(3) + n^2 \log_3 n = O(n^2 \log_3 n)$

b) Recursion Tree

Level	Number of sub-problems	Input size	Amount of work at each level
0	1	n	n^2
1	9	n/3	n^2
2	9^2	$n/3^2$	n^2
3	9^3	$n/3^3$	n^2
…	…	…	…
$\log_3 n$	$9^{\log_3 n}$	3	n^2

So, T(n) is equivalent to the total amount of work which is $\underbrace{n^2 + n^2 + \ldots + n^2}_{\log_3 n \text{ times}} = O(n^2 \log_3 n)$

c) Master Theorem

$a = 9, b = 3, d = 2$

$9 = 2^3$ so $a = b^d$

using Master Theorem, the asymptotic runtime is $O(n^d \log n) = O(n^2 \log_3 n)$

Question 4: Algorithm with Pseudocode

You are given an array of integers. Your task is to find the second-largest element in the array.

a) Describe an algorithm to find the second-largest element in the array.

b) Analyze the time complexity of your algorithm.

c) Can you optimize your algorithm if you know that the array is sorted in descending order? If so, how?

a) **Algorithm to Find the Second-Largest Element:**

```
java
public static Integer findSecondLargest(int[] arr) {
    if (arr.length < 2) {
        return null;  // There's no second-largest element in an array with fewer
than two elements
    }

    int largest = Math.max(arr[0], arr[1]);
    int secondLargest = Math.min(arr[0], arr[1]);

    for (int i = 2; i < arr.length; i++) {
        if (arr[i] > largest) {
            secondLargest = largest;
            largest = arr[i];
        } else if (arr[i] > secondLargest && arr[i] != largest) {
            secondLargest = arr[i];
        }
    }

    return secondLargest;
}
```

b) **Time Complexity Analysis:**

The time complexity of this algorithm is O(n), where n is the size of the array.

c) **Optimizing for a Sorted Array (Descending Order):**

If you know that the array is sorted in descending order, you can optimize the algorithm by simply returning the second element of the array. In this case, there's no need to iterate through the entire array.

Optimized Java code for a sorted array in descending order:

```
java
public static Integer findSecondLargestSortedDescending(int[] arr) {
    if (arr.length < 2) {
        return null;  // There's no second-largest element in an array with fewer
than two elements
    }

    return arr[1];
}
```

This optimized version has a time complexity of O(1) since it directly returns the second element.