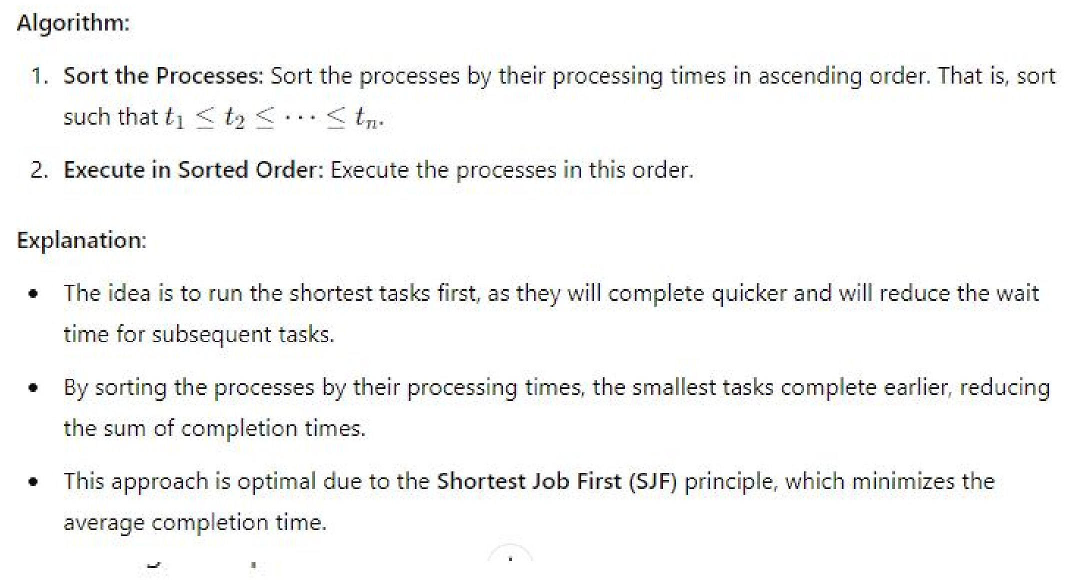
Question-8 Algorithm Answers

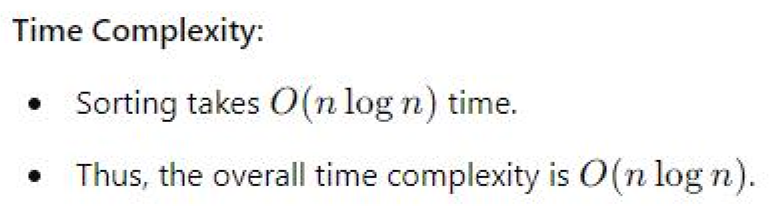


Algorithm:

1. **Sort the Processes:** Sort the processes by their processing times in ascending order. That is, sort such that $t_1 \leq t_2 \leq \cdots \leq t_n$.
2. **Execute in Sorted Order:** Execute the processes in this order.

Explanation:

- The idea is to run the shortest tasks first, as they will complete quicker and will reduce the wait time for subsequent tasks.
- By sorting the processes by their processing times, the smallest tasks complete earlier, reducing the sum of completion times.
- This approach is optimal due to the **Shortest Job First (SJF)** principle, which minimizes the average completion time.



Time Complexity:

- Sorting takes $O(n \log n)$ time.
- Thus, the overall time complexity is $O(n \log n)$.

```
import java.util.Arrays;
import java.util.Scanner;

public class MinimizeCompletionTime {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Input: number of processes
        System.out.print("Enter the number of processes: ");
        int n = scanner.nextInt();

        // Input: processing times of the processes
        int[] processingTimes = new int[n];
        System.out.println("Enter the processing times of the processes:");
        for (int i = 0; i < n; i++) {
            processingTimes[i] = scanner.nextInt();
        }

        // Sort the processing times in ascending order
        Arrays.sort(processingTimes);

        // Calculate the completion times and the sum of completion times
        int totalCompletionTime = 0;
        int currentCompletionTime = 0;

        for (int i = 0; i < n; i++) {
            currentCompletionTime += processingTimes[i];
            totalCompletionTime += currentCompletionTime;
        }

        // Calculate the average completion time
        double averageCompletionTime = (double) totalCompletionTime / n;

        // Output the result
        System.out.println("The average completion time is: " + averageCompletionTime);
    }
}
```

Explanation of the Code:
Input:

The number of processes n.
The processing times for each of these processes.

Sorting:

The processing times are sorted in ascending order using Arrays.sort().

Calculating Completion Times:

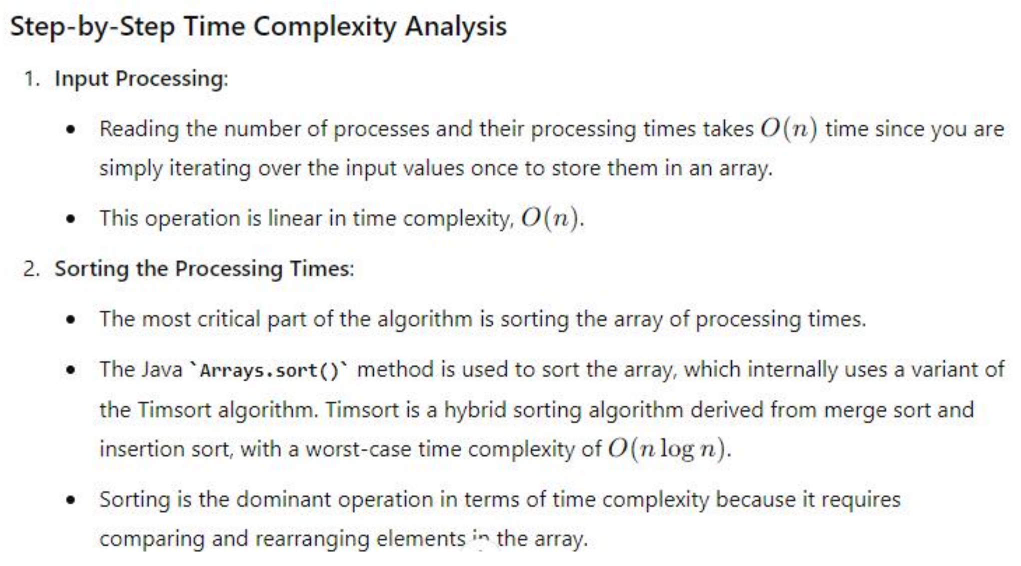
The algorithm calculates the cumulative completion time as it iterates through the sorted list of processing times.
currentCompletionTime keeps track of the sum of the times of the processes that have been executed so far.

totalCompletionTime accumulates the completion times of all processes.
Average Completion Time:

The average completion time is then computed by dividing totalCompletionTime by the number of processes n.

Output:

The average completion time is printed as the final result



Step-by-Step Time Complexity Analysis

1. **Input Processing:**
   - Reading the number of processes and their processing times takes $O(n)$ time since you are simply iterating over the input values once to store them in an array.
   - This operation is linear in time complexity, $O(n)$.
2. **Sorting the Processing Times:**
   - The most critical part of the algorithm is sorting the array of processing times.
   - The Java `` `Arrays.sort()` `` method is used to sort the array, which internally uses a variant of the Timsort algorithm. Timsort is a hybrid sorting algorithm derived from merge sort and insertion sort, with a worst-case time complexity of $O(n \log n)$.
   - Sorting is the dominant operation in terms of time complexity because it requires comparing and rearranging elements in the array.

3. **Calculating Completion Times:**
    - After sorting, the algorithm iterates over the array to compute the cumulative completion times and the total completion time.
    - This involves a simple loop over the array, which is $O(n)$ in time complexity because it processes each element exactly once.
4. **Calculating the Average Completion Time:**
    - The average completion time is calculated by dividing the total completion time by the number of processes. This division operation is $O(1)$ or constant time, which is negligible in terms of complexity.

Combining the Complexities

- **Input Processing:** $O(n)$
- **Sorting:** $O(n \log n)$
- **Completion Time Calculation:** $O(n)$

Since $O(n \log n)$ dominates $O(n)$ and $O(1)$, the overall time complexity of the algorithm is $O(n \log n)$.

Conclusion

The $O(n \log n)$ time complexity arises because sorting is the most computationally expensive step in the algorithm. The sorting ensures that the processes are executed in an order that minimizes the total and average completion times, which is critical to the efficiency and correctness of the algorithm.

Algorithm (Selection Sort-Based Approach)

1. **Input:**
    - Let $t[1], t[2], \ldots, t[n]$ be the array of processing times.
2. **Selection Sort:**
    - For each i from 1 to $n - 1$:
        - Find the smallest element from $t[i]$ to $t[n]$.
        - Swap this smallest element with $t[i]$.
    - This ensures that after each iteration, the first i elements of the array are sorted in non-decreasing order.

3. **Compute Completion Times:**
   - Initialize `` `totalCompletionTime` `` to 0 and `` `currentCompletionTime` `` to 0.
   - For each i from 1 to n:
     - Set `` `currentCompletionTime = currentCompletionTime + t[i]` ``.
     - Add `` `currentCompletionTime` `` to `` `totalCompletionTime` ``.
4. **Calculate the Average Completion Time:**
   - Compute `` `averageCompletionTime = totalCompletionTime / n` ``.
5. **Output:**
   - Print `` `averageCompletionTime` ``.

```
import java.util.Scanner;

public class MinimizeCompletionTimeN2 {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Input: number of processes
        System.out.print("Enter the number of processes: ");
        int n = scanner.nextInt();

        // Input: processing times of the processes
        int[] processingTimes = new int[n];
        System.out.println("Enter the processing times of the processes:");
        for (int i = 0; i < n; i++) {
            processingTimes[i] = scanner.nextInt();
        }

        // Selection Sort - O(n^2)
        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (processingTimes[j] < processingTimes[minIndex]) {
                    minIndex = j;
                }
            }
            // Swap the found minimum element with the first element
            int temp = processingTimes[minIndex];
            processingTimes[minIndex] = processingTimes[i];
            processingTimes[i] = temp;
        }

        // Calculate the completion times and the sum of completion times
        int totalCompletionTime = 0;
        int currentCompletionTime = 0;

        for (int i = 0; i < n; i++) {
            currentCompletionTime += processingTimes[i];
            totalCompletionTime += currentCompletionTime;
        }
        // Calculate the average completion time
        double averageCompletionTime = (double) totalCompletionTime / n;

        // Output the result
        System.out.println("The average completion time is: " + averageCompletionTime);
    }
}
```

Explanation:

- **Selection Sort**: The main difference here is that instead of using a built-in sorting method like `` `Arrays.sort()` ``, which has $O(n \log n)$ complexity, we're using selection sort. Selection sort iteratively selects the smallest remaining element and swaps it into the correct position, resulting in $O(n^2)$ complexity.
- **Rest of the Algorithm**: The rest of the algorithm remains the same as before: it computes the completion times, adds them up, and calculates the average completion time.

Time Complexity:

- **Selection Sort**: The nested loops in selection sort result in $O(n^2)$ time complexity.
- **Completion Time Calculation**: The time to compute the completion times and their sum is $O(n)$.
- **Overall Complexity**: The overall time complexity is dominated by the selection sort, resulting in $O(n^2)$.

This implementation is less efficient than the $O(n \log n)$ version but is useful for understanding how a brute-force approach can solve the problem.