**Assignment1**

**Q1.**

**1. Study Merge Sort Algorithm:**

- **Time Complexity:** Merge Sort has a worst-case and average-case time complexity of O(nlogn). It uses the divide-and-conquer approach:
    - Divides the array into two halves.
    - Recursively sorts both halves.
    - Merges the two sorted halves.
- **Space Complexity:** Merge Sort requires additional space for the temporary arrays used during merging. This space complexity is O(n) in the traditional recursive implementation.

**2. Single Processor Consideration:**

- You are tasked with using a single processor, meaning parallelized versions of Merge Sort (such as parallelized merge) are not allowed. This setup will run standard Merge Sort sequentially.
- **Single Processor Usage:**
    - C++ programs, by default, run in a single thread unless you explicitly use threading libraries like `<thread>`, OpenMP, or other parallel programming techniques. The code here does not use any such libraries, so it operates on a single thread and processor core.
- **Merge Sort in the Code:**
    - Both the recursive and iterative versions of Merge Sort break the problem into smaller subproblems and solve them sequentially, without distributing the work across multiple threads or processors.

**3. Improving Running Time:**

- **Optimization Techniques for Merge Sort:**
    - **Switch to Insertion Sort for Small Subarrays:** For small subarrays (e.g., subarrays of size < 32), switching to Insertion Sort during the recursive calls can reduce the overhead of recursion. This can improve the running time in practice.
    - **Avoiding Auxiliary Arrays:** In place of using separate arrays for merging, you can use alternate strategies like alternating between two arrays to reduce memory usage.
    - **Iterative Merge Sort:** You can use an iterative version of Merge Sort (bottom-up merge sort) to avoid the overhead of recursion, which might reduce the constant factors impacting time performance.

**4. Comparing Running Times of Two Methods of Merge Sort:**

You could compare the recursive (top-down) and iterative (bottom-up) approaches:

- **Recursive Merge Sort:** Uses recursion and requires additional memory for stack calls.
- **Iterative Merge Sort:** Eliminates recursion but works similarly by merging subarrays of increasing sizes. It avoids the memory overhead of recursion and might perform slightly better on large input sizes.

**Q2.**

1. **Initial Implementation:**
    - Implement the Quicksort algorithm and test it using randomly synthesized lists of records.
    - Evaluate whether its time complexity is O(nlogn) based on experimental results.
2. **Pivot Selection:**
    - Modify the Quicksort algorithm to select pivots randomly (instead of using a fixed strategy like picking the first element).
    - Analyze the changes in time requirements and discuss whether the time complexity remains O(nlogn).

**Steps for the Implementation and Experimentation:**

1. **Standard Quicksort Implementation**:
    - Implement the recursive divide-and-conquer algorithm that uses a fixed pivot (e.g., first element).
    - Generate several random lists of varying sizes (e.g., 100, 1,000, 10,000 elements) to test the performance.
    - Measure the execution time and compare it against O(nlog⁡n)O(n \log n)O(nlogn) behavior.
2. **Random Pivot Quicksort**:
    - Modify the Quicksort algorithm to select a random pivot in each partitioning step.
    - Perform the same set of experiments with the same lists.
    - Compare the time taken and check for changes in performance, especially looking for average case O(nlogn) and whether worst-case behavior is improved (since random pivot helps avoid consistently poor performance).

**Experimental Justification:**

- The original Quicksort with a fixed pivot can have poor performance on some inputs (e.g., already sorted lists), leading to O(n^2) time complexity.
- Random pivot selection generally improves performance by ensuring a more balanced partitioning, leading to expected O(nlogn) time complexity in most cases.