CIS 103 - Homework Assignment 5

Topic: Searching, Sorting, and Complexity Analysis

Due: 10/04/2024

Objective:

The objective of this assignment is to introduce students to common searching and sorting algorithms, as well as time complexity analysis. Students will implement these algorithms in Python, analyze their performance, and answer theoretical questions on algorithmic complexity.

Problem 1: Implementing Linear Search and Binary Search

1. **Linear Search**: Implement a function that performs linear search. The function should take a list and a target element as input and return the index of the target element if found, otherwise return `-1`.

2. **Binary Search**: Implement a function that performs binary search on a sorted list. The function should return the index of the target element if found, otherwise return `-1`.

- **Hint**: Remember that binary search requires the list to be sorted in advance.

3. **Comparing Time Complexity**: Write a Python program that compares the performance of both algorithms on lists of varying sizes (e.g., 1000, 10,000, and 100,000 elements). Use Python's `time` module to measure the time taken by each algorithm.

Code Requirements:

- Implement both `linear_search()` and `binary_search()` functions.
- Write a driver code that tests both search algorithms on the same data sets and prints out the time taken for each.

Problem 2: Implementing Sorting Algorithms

1. **Bubble Sort**: Implement the bubble sort algorithm in Python. Your function should take an unsorted list and return the list sorted in ascending order.

2. **Merge Sort**: Implement the merge sort algorithm in Python. Your function should recursively divide the list, sort it, and then merge the sorted sublists.

3. **Comparing Time Complexity**: Test both sorting algorithms on lists of varying sizes (e.g., 1000, 10,000, and 100,000 elements) and use Python's `time` module to compare their performance.

Code Requirements:

Implement both `bubble_sort()` and `merge_sort()` functions.
Write a driver code that tests both sorting algorithms on the same data sets and prints out the time taken for each.

Problem 3: Complexity Analysis of Searching and Sorting Algorithms

Using what you have learned about time complexity, answer the following questions:

1. **Linear Search**:
- What is the worst-case time complexity of linear search?
- What factors determine the performance of linear search?

2. **Binary Search**:
- What is the time complexity of binary search in the worst case? Explain why binary search requires a sorted list.
- Can binary search be applied to unsorted lists? Why or why not?

3. **Bubble Sort**:
- What is the worst-case time complexity of bubble sort? How does it compare to merge sort in terms of performance?
- What are the advantages and disadvantages of bubble sort?

4. **Merge Sort**:
- What is the time complexity of merge sort in the best, worst, and average cases?
- How does merge sort's divide-and-conquer strategy help improve its performance compared to bubble sort?

Problem 4: Analyzing Python's Built-in Sorting Algorithm

Python's built-in `sort()` method and `sorted()` function use a highly optimized sorting algorithm called **Timsort**, which is a hybrid sorting algorithm derived from merge sort and insertion sort.

1. Use Python's `sorted()` function to sort a list of integers.
2. Measure the performance of `sorted()` and compare it with your implementation of merge sort.
3. Research and explain why Timsort is more efficient than merge sort in certain cases.

Problem 5: Theory of Time Complexity

Answer the following questions about time complexity:

1. **What is the difference between time complexity and space complexity?**

2. **Explain the Big O notation.** How is it used to describe the efficiency of an algorithm?

3. **Provide examples of algorithms with the following time complexities and explain what they

mean:**
- O(1)
- O(n)
- O(n^2)
- O(log n)
- O(n log n)

4. **Why is it important to consider both time and space complexity when designing algorithms?**

5. **Discuss the trade-offs between recursive algorithms and iterative algorithms in terms of time complexity.** Provide examples where recursion may lead to higher time complexity.

Submission Instructions:

Submit your Python scripts for each coding problem as separate `.py` files.
Answer the theoretical questions in a `.docx` or `.pdf` file.
Ensure your code is well-commented, formatted correctly, and follows best practices.
Submit your assignment via the Brightspace and GitHub repository by the due date.