Assignment 1

**Sorting Algorithm Assignment**

**Step 1 - Implementing Sorting Algorithms**

**Task**

* Implement five sorting functions in Python that sort an unsorted list:
* def bubble\_sort(my\_list):
* def selection\_sort(my\_list):
* def insertion\_sort(my\_list):
* def quick\_sort(my\_list):
* def insertion\_sort(my\_list, left, right):
* Provide a main function that:
  + Uses each of the above sorting functions to sort a list of length 100.
  + Ensures the sorting functions work as expected.

**Questions**

1. Explain how each sorting algorithm works in your own words. What are the key steps involved in each?
2. Compare and contrast the five sorting algorithms implemented. What are their fundamental differences?
3. Which sorting algorithm would you choose for small datasets? Why?
4. Modify your implementation to sort a list of custom objects instead of integers. What changes were needed?

**Step 2 - Counting Steps (T(n))**

**Task**

* Modify your sorting functions so that they:
  + Sort the received list.
  + Count the number of operations (T(n)) required to perform the sorting.
  + Return T(n) along with the sorted list.
* **Testing:**
  + Send a best-case scenario, a worst-case scenario, and an average-case scenario to each sorting function.
  + Record the output values for T(n).

**Questions**

1. In your implementation, how do you count the exact number of operations? What assumptions did you make?
2. How does T(n) behave for the best, worst, and average cases in each sorting algorithm? Provide theoretical and empirical results.
3. How do the step counts (T(n)) you measured compare to the theoretical Big-O complexity of each algorithm? Are they aligned?
4. If two sorting algorithms have the same worst-case complexity, does that mean they will always take the same number of steps? Why or why not?

**Step 3 - Analyzing T(n) Graphs**

**Task**

* Use the sorting functions with the T(n) calculation feature to plot T(n) vs. n for different list sizes:
  + List sizes: **10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 1,000,000, 10,000,000**
  + Use a **worst-case scenario** for your input lists.

**Questions**

1. Describe the shape of the T(n) vs. n plots for each sorting algorithm. What patterns do you observe?
2. Which sorting algorithm exhibits the steepest growth in T(n)? What does this indicate about its efficiency?
3. How does the graph confirm or contradict your expectations based on Big-O notation?
4. What would you expect to happen if you added an optimized merge sort or heapsort to the comparison? Would the T(n) vs. n behavior be different?

**Step 4 - Timing Analysis**

**Task**

* Use Python’s time library to measure the execution time of each sorting algorithm.
* Run the sorting functions on the same list sizes as in Step 3.
* Plot **algorithm completion time vs. n** using the worst-case scenario input lists.
* Compare the actual execution time with the T(n) results.

**Questions**

1. Compare the runtime graphs (completion time vs. n) with the T(n) graphs. Do they follow similar trends? Why or why not?
2. Are there any discrepancies between the T(n) and actual runtime measurements? What factors might contribute to these differences?
3. For larger datasets (e.g., 1,000,000 elements), which algorithm performs the worst in actual execution time? Is this expected?
4. Why does quicksort perform better on average, even though its worst-case complexity is O(n²)?
5. What role does Python's built-in sorting algorithm (sorted()) play in real-world applications? How does it compare to the sorting algorithms implemented in this assignment?

**General Reflection and Optimization**

**Questions**

1. If you had to optimize one of your sorting algorithms, which one would it be and why?
2. How does memory usage compare for different sorting algorithms? Would memory constraints affect your choice of sorting algorithm?
3. Research an advanced sorting algorithm (e.g., radix sort, merge sort). How does it compare to the ones you implemented in terms of T(n) and runtime?