CS/IT 200

Lab 10: Sorting

Recommended Due Date: Thursday, December 3, 11:59pm

Submission

- Submit all code and other materials in a single zip file to the Lab 10 assignment folder on Kodiak.

Goal: Analyze and report the running times of sorting algorithms.

Part I

The sorting module distributed for this lab contains code for Selection Sort, Insertion Sort, Bubble Sort, Merge Sort, and Quick Sort. Using that class, write a series of small programs to measure the efficiency of the various sorting algorithms for different sizes of N using lists (the built-in Python object).

(a) Gather measurements of the following for each of the five sorting algorithms in the `sorting` module, using the `timeit` module. For each subpart i-iii, collect data points for at least the following list sizes: N = 100; 1000; 10,000; 100,000. Place this code in `lab10.py`.

   i. Create a list of random integers and sort it. The range of random values should be between 0 and 10*N. (That is, if N=100, fill the list with random values between 0 and 1000.)
   ii. Create a list of integers that is already sorted and sort it.
   iii. Create a list of integers that is sorted backwards and sort it.

   If any condition produces a `RuntimeError` or `RecursionError`, report ERR instead of a time.

   Use the `default_timer()` function in the `timeit` module to time a block of code.

```
start_time = timeit.default_timer()
# YOUR CODE TO GET TIMED GOES HERE
end_time = timeit.default_timer()
time_elapsed = end_time - start_time
```

   Tips

- Make sure that you are starting with an appropriate list prior to each sorting test. Don't accidentally use a sorted list from the previous trial where you should be using a random list.
- Make sure you're only timing the sorting, not also the time required to build the list.
- Give yourself plenty of time to take these measurements. Execution of N=100,000 may take a long time to finish. Have other work handy, or write your programs so

they can run while you take a nap. In the past, some students ran this code overnight.

(b) Present your data in a table that makes it easy to compare the difference between sorting algorithms for each of the subparts.

Part II

The `_choose_pivot()` function selects a pivot point for the Quick Sort algorithm. Currently, the pivot is whatever is already in `mylist[b]`, where `a` and `b` are the edges of the section of the list that we are looking at.

A different way to choose a pivot is the **median-of-three** approach, using the median of the three values `mylist[a]`, `mylist[b]`, and `mylist[(a+b)//2]`. Modify `_choose_pivot()` to select a pivot in this way. The `_choose_pivot()` function should return <u>the index</u> where the median value is located.

Reminder: Determine the median based on the ***values*** in the array, not the indices themselves, but **return the index**. In the diagram below, the median value is 17, because 8 < 17 < 21, even though 21 is in the `(a+b)//2` slot. Since the median is 17, we would want `_choose_pivot()` to return `a`.

	a		(a+b)//2		b	
...	17	...	21	...	8	...

Collect and report the same data that you did for Part I (a) for this Modified Quick Sort and add it to your table from Part I (b).

Part III

We discussed (or will discuss) the Radix Sort algorithm, which is substantially different from the other sorting algorithms.

Add an implementation of radix sort to the sorting module that was distributed with this lab. Your implementation can assume a list of integers. Name the function `radix_sort()`. Its only parameter should be a list. Depending on your implementation, you may want your function to return the final sorted list.

Collect and report the same data that you did for Part I (a) for Radix Sort on lists of integers. Report the data in the same table as Part I and II.

Part IV

Place your table from Parts I-III in a document (Word or PDF). In the same document, answer the following questions:

- Compare and contrast the relative performance of the sorting algorithms for the different operations. Which one is faster/slower? Under which conditions? Do your observations match the complexities we have discussed in class?
- Which conditions, if any led to `RuntimeErrors`? What was the cause of these errors? Why did those conditions lead to these errors?
- How did Modified Quick Sort compare to the original quick sort? Did it ever succeed when the original algorithm failed, or vice versa?

What to Submit:

- Your code (`lab10.py` and `sorting.py`, plus any additional modules that you used)
- A document that includes your table of data and your answers to the questions in Part IV.

Rubric

Grade	Overall	Code	Table/Analysis
4	A 4 in one part and at least a 3 in the other part.	Correct implementation of median-of-three and radix sort. Experiment code structure is present and correct.	Table has entries for all cases (with reasonable values). Analysis is reasonable and correct.
3	At least a 3 in one part, and at least a 2 in the other part.	Minor flaw in median-of-three or radix sort.	Table is missing up to 3 entries or one question is not addressed.
2	At least a 2 in both parts; or at least a 3 in one part, and at least a 1 in the other part.	Minor flaw in experiment code, or minor flaws in both algorithms.	Table is missing up to 6 entries or two questions are not addressed or (table missing 3 entries and one question not addressed)
1	At least a 1 in both parts.	Major flaw in median-of-three or radix sort, but sorting still works.	Analysis is correct but table is missing many entries, or vice versa.
0		Median-of-three and/or radix sort crashes (when it shouldn't) or fails to sort the list; or experiment code crashes for reasons other than the sorting algorithms themselves.	No table or no analysis