CSE431 Course Project

Algorithmic theory provides us with a solid framework for understanding how the complexity of an algorithm increases with the size of the input. In many cases, these techniques generate a solid expectation for which of two algorithms will be better to solve a problem. In practice, however, we can never be sure of an algorithm's speed on a particular problem size until we try it out.

In the four sections below, you will experimentally test how the speed of algorithms vary across input sizes; each section is worth 150 points toward your project grade for a total of 600 points.

For each of the four sections

- quicksort vs insertion sort
- hybrid sorting
- binary search trees vs hash tables
- multiset vs vectors

you will upload a write-up (as a pdf) with the following five sections:

- *Hypothesis*: Briefly describe what results you believe you will find. It is important to write your hypothesis before you start conducting experiments as a way of acknowledging your initial expectations. *Note*: you will not lose points for your hypothesis being incorrect; in fact, getting a result different from your hypothesis will often be more exciting and more of a learning experience.
- *Methods*: Describe step-by-step the experiments that you conduct. Provide the source code that you use, and details about which compiler you use, how you compile it (*e.g.*, optimization flags), and the range of inputs that you feed into your program. Your methods should accurately reflect how you actually generated your data such that someone reading them can replicate your experiment.
- *Results*: Present the data that your experiments produced. In most cases this will be a graph (as described in each question) and a brief explanation to make sure the reader understands the data in that graph. Graphs should be properly labeled (axis labels, etc)!
- *Discussion*: Provide a brief discussion of your data. Did anything about it surprise you? Were there any unexpected challenges in collecting the data? This section is where you will answer specific questions posed in each of these problems.
- *Conclusions*: Present a concise take-away for this experiment. For example, "Under the conditions tested, data structure A produces a faster algorithm for $n < 1000$, while data structure B is faster for $n > 1500$. For n between 1000 and 1500 the two data structures are indistinguishable."

Some advice:

- If you have trouble measuring short run times, put a loop around an experiment to do it 1000 times in a row (or more) so that you can tell the difference. For example, if Quicksort takes 0 seconds when n=10, you might find that running it 1000 times takes 0.013 seconds. Then you can estimate that each iteration took 0.000013 seconds.
- Most languages have a simple way to get the current time in milliseconds, so you can run this before and after the test and subtract to get the total time elapsed. For example, in C++ you can do:

```
#include <ctime>
std::clock_t start_time = std::clock();
the_function_you_want_to_time();
std::clock_t tot_time = std::clock() - start_time;
std::cout << "Time: "
          << ((double) tot_time) / (double) CLOCKS_PER_SEC
          << " seconds" << std::endl;
```

- Our main goal is to give you experience empirically testing algorithms or data structures against one another. If you have ideas for how to adjust the experiments above to more clearly test these effects, you should do so. For example, if you prefer to do parts 3 and 4 in a language other than C++, you may, but make sure to properly compare balanced binary trees, hash tables, and sorted arrays as containers. Furthermore, feel free to come up with your own extensions to these projects – especially thorough or clever studies will warrant **extra credit points**.

Remember, all work on the course project must be your own.

All coursework (including course project) MUST be manually submitted to Gradescope in order to complete submission! There is no auto-submit feature enabled for any Gradescope coursework. There will be no exemptions granted if you forget to manually submit your coursework. If time permits, you can contact the course staff and they can check on the Gradescope platform and confirm your coursework submission status.

IMPORTANT NOTE: the late day policy applies only to assignments and **NO LATE DAYS ARE ALLOWED FOR THE COURSE PROJECT!** Thus, the stated project deadline is a hard deadline.

Honors option: an honors option is available and takes the form of an add-on extension to the course project (see syllabus for more details). Contact the instructor for more details.

1. Quick sort vs. Insertion sort

(150 pts)

Quicksort has an expected runtime of $\Theta(n \log n)$; insertion sort has an expected run time of $\Theta(n^2)$. As such, we know that Quicksort will be faster for very large n. Insertion sort, however, turns out to be faster for small n. Your job is to figure out *how small*.

Compare implementations of Quicksort and Insertion sort, testing each with a range of values for n. Provide a graph of the results, clearly indicating the value on n where the lines cross.

You may use your own implementations, in any language you choose, or ones that you find elsewhere, as long as you cite your sources. You must use a wide enough range of values of n to provide a convincing argument of your answer.

Upload your write-up (see the assignment description for what should go into the write-up) as a pdf along with any associated supporting files (*e.g.*, source code, etc).

2. Hybrid sorting

(150 pts)

Given the results of part 1 (Quick sort vs. Insertion sort), build a Hybrid sorting algorithm. When you recurse in Quicksort, if a partition size is less than or equal to some constant *k*, you should use insertion sort, but if it is greater than *k* you should continue with Quicksort. Experimentally determine what value of *k* will optimize speed.

Is *k* the same as the crossover point for part 1? Why do you think this is the case? Generate a graph comparing this hybrid sort to both Insertion Sort and Quicksort. Make sure to test with a wide range of values for *k* (to be sure of your answer) as well as for *n* (in case the number of values sorted affects your answer).

Upload your write-up (see the assignment description for what should go into the write-up) as a pdf along with any associated supporting files (*e.g.,* source code, etc).

3. Binary Search Trees vs. Hash Tables

(150 pts)

In the C++ standard library, `std::multiset` can be used to easily store values in a balanced binary tree. Likewise, `std::unordered_multiset` can be used to store values in a hash table.

In theory, a hash table should be faster for insertions and deletions, but how much faster? Compare the two for insertion of $n = 10$, $n = 100$, $n = 1000$, ..., $n =$ as high as you need to go for at least one of the data structures to take more than 3 seconds to run.

Generate a graph comparing the performance. You may use binary tree and hash table implementations in another language if you prefer not to use C++.

Upload your write-up (see the assignment description for what should go into the write-up) as a pdf along with any associated supporting files (*e.g.*, source code, etc).

4. Multiset (or something equivalent) vs. Vectors

(150 pts)

In C++, `std::vector` is typically considered a poor choice for a sorted container because for every insertion, all subsequent values must be shifted to make room for it. The net effect should be linear-time insertions. Modern processors, however, can move whole blocks of memory at once.

How efficient can a sorted vector be? Implement one where you use binary search to quickly find the insertion position of a new value in the vector, and then use the vector member function insert (*iterator*, *value*) to place the new value into the vector (insert will efficiently adjust elements that need to be moved). Graph the time your sorted vector takes as compared to `std::multiset`, which you already looked at in problem 3. Again, you may use a language other than C++ as long as you pick equivalent data structures.

Upload your write-up (see the assignment description for what should go into the write-up) as a pdf along with any associated supporting files (*e.g.,* source code, etc).