TERM PAPER - CS350

RUSSELL MILLER AND BEN CARR

Introduction

This project is an assessment of several sorting algorithms. We chose to implement four sorting algorithms:

- Bubble, average $O(n^2)$, worst-case $O(n^2)$
- Insertion, average $O(n^2)$, worst-case $O(n^2)$
- Quick, average $O(n \lg n)$ worst-case $O(n^2)$
- Merge, average and worst-case $O(n \lg n)$

These four algorithms were implemented and tested in two languages: C++ and Java. We had initially planned on including Python and Haskell, but due to time constraints and the limitations of Python, we were only able to collect significant data and run analysis on our Java and C++ algorithms. Each program tested 900 completely random unsorted lists of integers, each of size $n = \{10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000\}$. This paper will discuss our findings, assess our results and communicate to the reader our understanding of how complexity theory comes up in practice. We found that quicksort ran exceptionally fast on these random lists, so additional data (lists size "n" up to 10,000,000 for Java and 100,000,000 for C++) was run on quicksort.

Date: March 10, 2011.

Background

We split up the main algorithms we would need to write so that each of us would be writing them in two languages. Russell worked in Java and Python while Ben worked in Haskell and C++. Russell has written a lot of small Python programs, and a few medium-scale ones. Java was a major refresher for him, as it has been a while. Ben was fairly new to Haskell and hadn't used C++ much since his freshman year of college, so Russell was helpful when he got stuck. Writing the algorithms themselves took very little time, especially in Python and Haskell. These languages are so straight-forward that if you know what you want to do you can have something working in very little time. Later in this paper, we'll discuss one of Python's weak points. We also used Python to generate our lists and write them to files.

One of the struggles we faced while working with Java was determining how to store the lists/arrays after reading them in from a file. At first we just wanted to have an array as a private class field. We then discovered that most of the algorithms we were working with sorted in place, meaning they would destructively change the values in the array. This would cause inconsistency for a class that has four different sort algorithms going in sequence and accessing the same array, and unfortunately we did not abstract the sorting out of the primary class. Rather we used four arrays, one dedicated to each algorithm.

The next tricky thing was figuring out how to run the stopwatch on the algorithms while they sorted the lists. The way the Java program was laid out, we were able to place the stopwatch calls right next to the sort method calls, and we used public methods to retrieve the times from the private fields they were stored in. We also had trouble with timing our C++ algorithms. Initially we tried using the C++ function `clock()` from the library `sys/time.h` which gives a value for the current clock cycle. We wanted the elapsed time in seconds, in order to compare the elapsed time to the other languages, so next tried to use `gettimeofday()`. This

is a function that returns seconds and the nanoseconds in a struct. We struggled a lot with getting readable results from the values in the struct. After researching more, we found a message board online with the following extremely helpful hint. Using `clock()` and a C macro, we were finally getting seconds as a decimal value.

```
// source: http://www.physicsforums.com/showthread.php?t=224989
double diffclock(clock_t t1, clock_t t2)
{
  double diffticks = t2 - t1;
  double diff = diffticks / (double)CLOCKS_PER_SEC;
  return diff;
}
```

The final struggle while working in Java, which carried over to the C++ implementation as well, was the merge sort. The way we understood the merge sort algorithm was that it split the list into two smaller lists and recursed down to those. Then the merge action put the smaller lists back together. Algorithms exist to do this in place with an array, but the easier implementation was to try the Java `List` class. In attempting to learn to use it, we also discovered the magic of the `ArrayList` class. Basically, it allows you to do list operations such as `cons` (Java calls it `add`), and there is also a `get` function that allows you to refer to a specific index. Both of these operations came in handy. C++ has lists in its standard library, and though it did not have the get method, it was not difficult to make the translation from the Java code to the C++ code.

Once we verified that all of the sorts were in fact sorting, adjusted the output that would be written to the CSV file, tested the stopwatch, and did a dry-run on a small set of lists, it was time to ship it. The program ran for two days and still had not finished. At this point we became very worried, so we tried timing a single list of size 100,000. In java it only took about 30 seconds, but in C++ it took 3 minutes. We later discovered that this enormous difference in time for these very similar languages was due to leaving some extra asserts in the code and forgetting

to comment them out for the stopwatch trials. Still, at this point we decided to reduce our list size limit to 100,000 instead of one million.

Understanding Algorithms

While putting these sorting algorithms into our programs, one thing that helped us understand them was to go through them, line by line, with an example on a dry erase board. In addition, we made sure to test them all and verify that they were doing their job. The first sort we studied was the Quicksort. It turns out it is a relatively simple algorithm. When looking at an array of numbers, you simply pick a value that is present in the array - the pivot - and divide the rest of the array based on whether the values are less than or greater than the pivot. It's recursive so you then apply this to those two new arrays.

The Quicksort is a strange beast. When compared with the complexity of other sorting algorithms it doesn't seem to be so great because of the worst case $O(n^2)$ time that it can take. However, it is extremely unlikely for that case to take place. Its performance is also very good on lists that are not "nearly-sorted" which was the type of data we were feeding it. The behavior of Quicksort for smaller n seems to be much faster than that of the algorithms which do not have as high of a worst-case condition. The trickiest part is getting a good pivot value. By getting a lucky pivot value the divide portion of this divide-and-conquer algorithm is more effective. The reason $O(n^2)$ is possible is that the recursion only stops when the list being sorted has been reduced to size one. So if every pivot value is the worst possible pivot value, every divide will result in an empty list, and an n-1 sized list. Figure 1 is a hand-drawn example of Quicksort in action.

The complexity of Quicksort can vary considerably depending on the layout of the data being sorted and on how the pivot value is picked. Carefully picking a pivot value can be considered optimization and in our algorithms we didn't do anything special to pick ours. This is a divide and conquer algorithm, and on

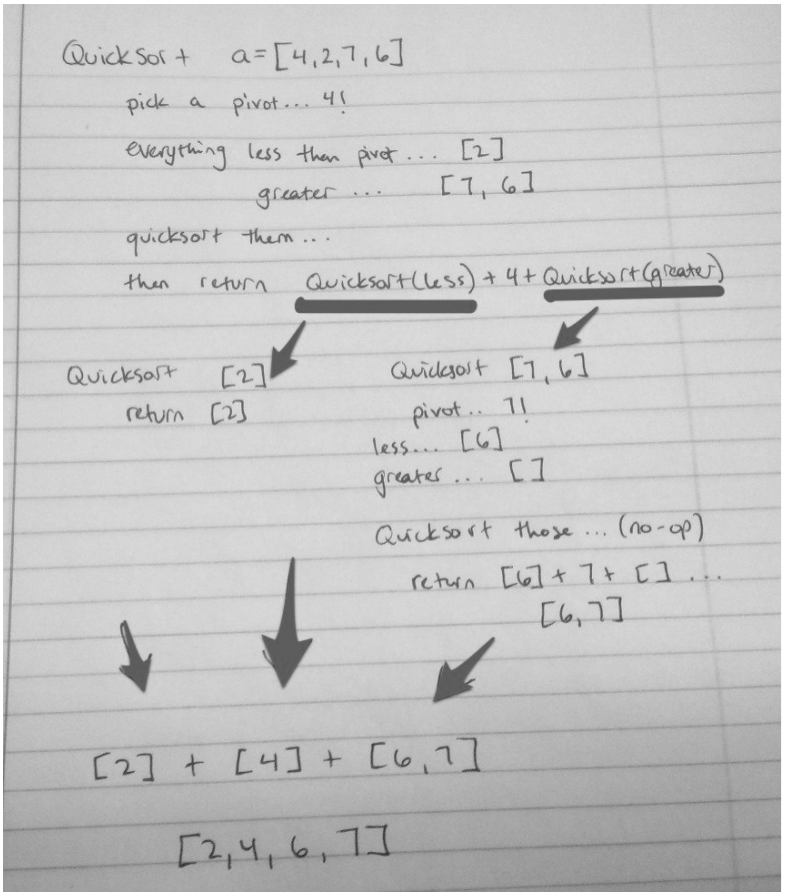

FIGURE 1. Walkthrough of quicksort

average can do O(n logn). The conquer part is a platform-specific complexity because sometimes concatenating lists or arrays together can be constant, but is usually a linear operation. The divide happens around a pivot value, and if the pivot value is greater than half of the array and less than the other half there will be a logarithmic complexity for the divide. If the pivot value causes one side of the division to be a lot larger than the other, the complexity moves toward quadratic. The nice thing about this algorithm, though, is it doesn't need to do as many comparisons as Merge Sort. Once it has done its full recursion it is simply concatenating results together. This may be the primary reason for Quicksort's victory.

The insertion sort is a particularly bad algorithm. What it does is iterate through the list, starting at the second element. This iteration is $O(n)$. For each iteration, There is also a while loop that starts at that point and goes backwards towards the beginning of the list swapping values that are out of order. The example I wrote shows how the "key" value is iterating across the array, and each time a new "key" is assigned a value, the while loop causes swaps towards the beginning. In asymptotic terms, the while loop is also $O(n)$. Most of the time, the while loop that happens after each new value is assigned to "key" does not actually traverse all the way to the beginning of the array, so the worst-case condition is not highly probable. On a sorted list, the condition to enter the while loop is always false and the complexity is $O(n)$ for the entire sort, because it is handled with just the for loop. In all other lists, however, the complexity is $O(n)$ for the for loop, times $O(n)$ for the while loop, resulting in $O(n^2)$.

The easiest algorithm to implement was bubble sort. The complexity is average case$O(n^2)$, as is shown in our comparison graphs. The reason that the complexity is so bad is as follows. It walks along the array/list (which is size n) and compares each item in order. When it gets through the list, it starts over, and on average has to walk through the list n times because it's only fixing (more or less) one thing at a time each time it walks through. I feel that detailed comments are the best way I can explain how bubble sort works:

```
void arrObj::bubbleSort()
{
    int swap;
    do{

        swap = 0; // swap is like a bool, it tells do-while
                  //to loop if the list is not sorted

        for (int n = 1; n < arrSize; n++) {
                 // each time we enter into the do-while loop, the
                 // for loop traverses the entire array, comparing
```

```
                // array[n-1] to array[n]. When it finds that
                // array[n-1] is greater than array[n],
                // it first sets swap to 1 (so we do-while again,
                // then it swaps. In my implementation, I use the
                // variable temp to store array[n-1]s value
                // copy the value in array[n] to array[n-1]
                // and then finally copy temps value to array[n].

            if (arrBubble[n-1] > arrBubble[n]) {
                swap = 1;
                int temp = arrBubble[n-1];
                arrBubble[n-1] = arrBubble[n];
                arrBubble[n] = temp;
            }
        }
    }while(swap == 1);
                // When we finally get here, if swap is set to 1, we have to
                // walk the entire array again, doing the for loop. Eventually
                // we will sort the list.
    //assert(isSorted(arrBubble));
}
```

Figure XXXXXXXXXXXX has a walk-through of bubble Sort on a small array, to show how it works.

Merge sort is much faster than bubble sort. We have included a walk through of merge sort in Figure XXXXXXXXXX. Merge sort takes a list and keeps splitting it into smaller lists until each list has one element in it (as seen in the top half of FIgure XXXXXXXXXXXX. This dividing takes $O(\log n)$ time, because you're dividing in half every time. Ss you recurse back up the tree, each merge of a size r list and a size s list takes $r + s - 1$ comparisons, which is $O(r + s)$ The final merge will be $r + s$, which is the size of the list, n. When you combine these you get $O(n \log n)$.

Findings and Analysis

Our C++ and Java programs took a total elapsed time of about 2 hours to sort all of the lists, sizes $n = \{10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000\}$.

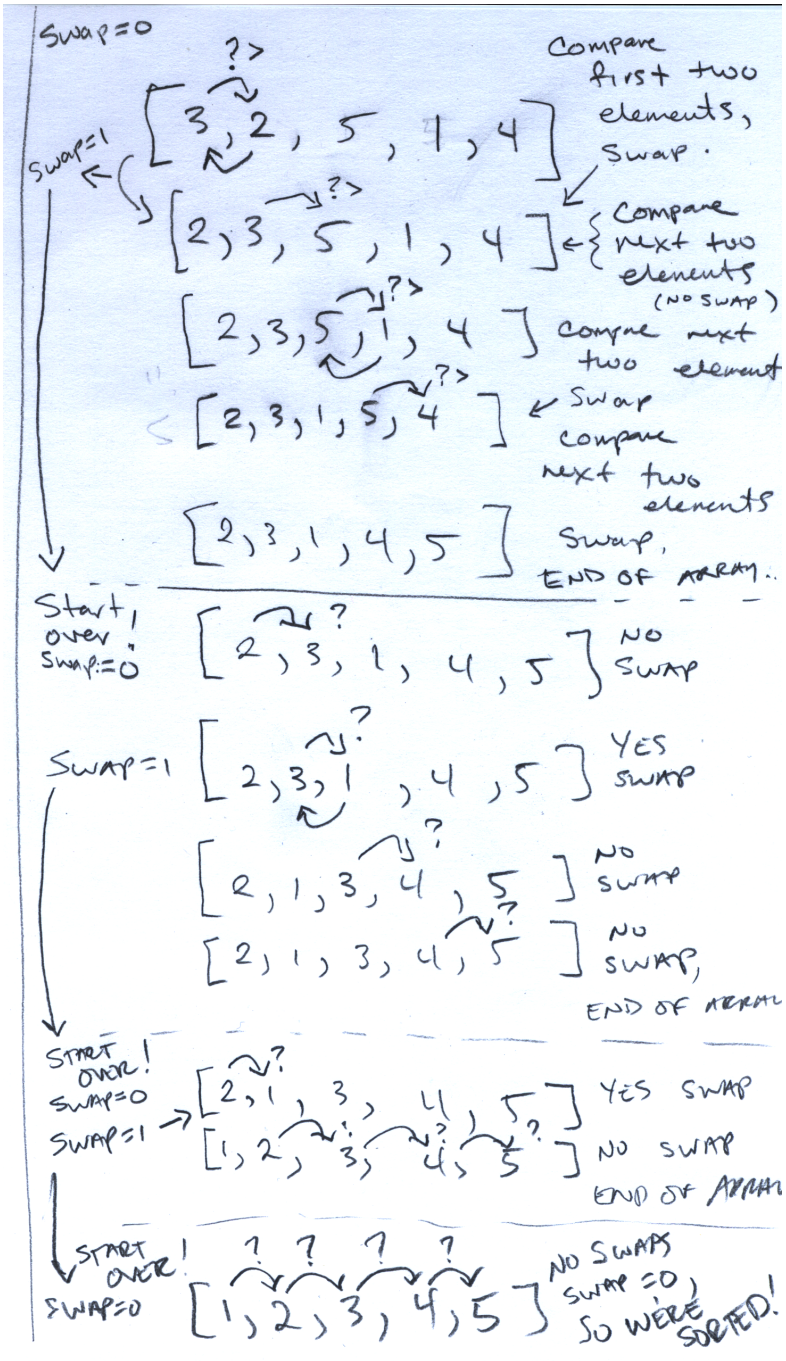

FIGURE 2. Walkthrough of bubbleSort

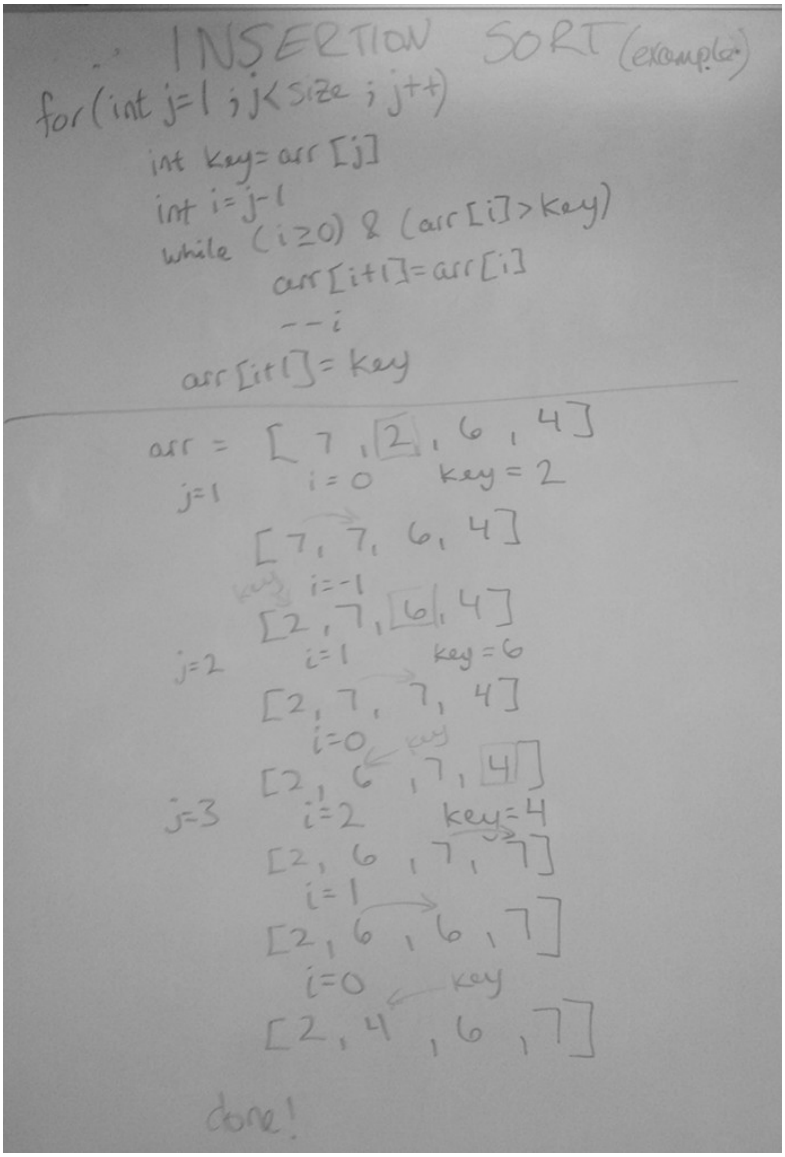

FIGURE 3. Walkthrough of bubbleSort

For each of these sizes, 100 different lists were sorted by each algorithm (the total number of lists the program sorted was $100 * |n| = 900$). The average time for each algorithm (in seconds) was recorded for Java and C++ and can be found in Table 1 and Table 2 respectively. Figure 1 and Figure 2 are the graphs of these averages. We wanted to see how consistent the algorithms were behaving; Figure 3 and Figure 4 show each trial's execution time for each algorithm. Quicksort ran

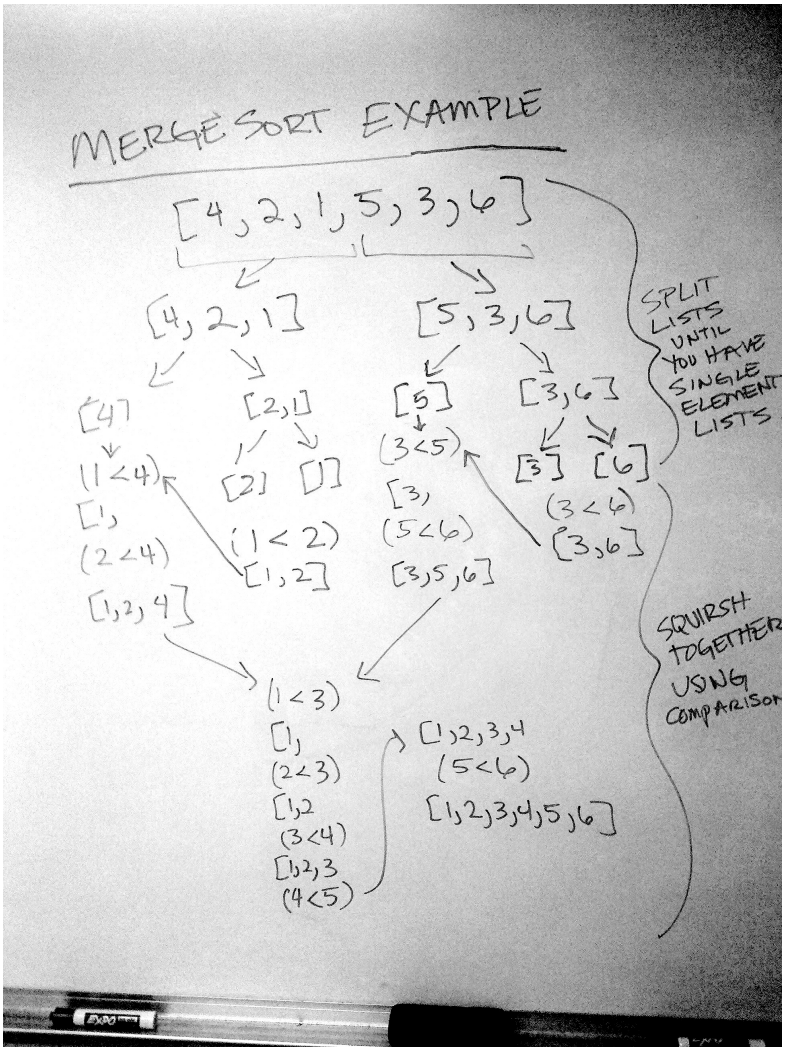

FIGURE 4. Walkthrough of merge sort

incredibly fast, so we decided to run it on larger lists (up to n = 100,000,000). Figure 5 shows the Quicksort against the calculated n log n plot. Note that in C++ our values of n were much higher. When attempting those values with Java, the virtual machine ran out of memory and threw an exception so a lower limit was used.

List Size	Quick	Bubble	Insertion	Merge
10	0	0	0	0
50	0	0	0	0
100	0	0	0	0
500	0	0	0	0
1000	0	0	0	0
5000	0	0.05	0	0.01
10000	0	0.19	0.01	0.03
50000	0	4.84	0.37	0.72
100000	0.01	19.4	1.51	3.03

TABLE 1. Java average run times (in seconds) of unordered lists.

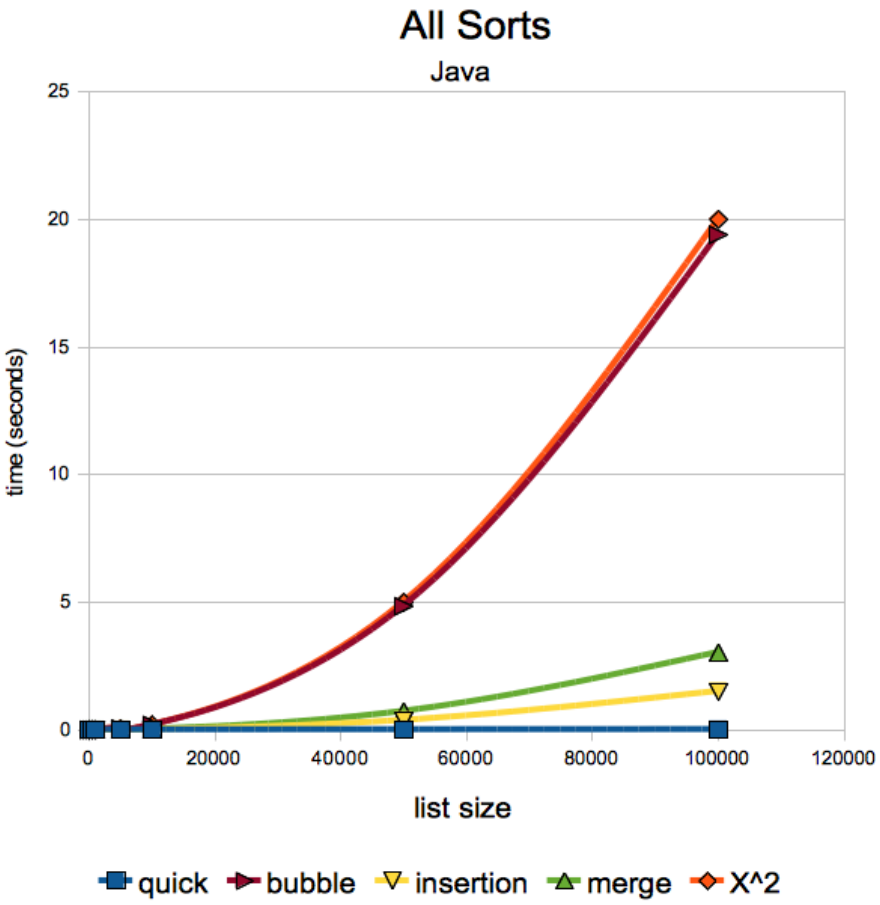

FIGURE 5. Behavior of sorts against each other

TESTING METHODS AND LIST GENERATION STRATEGIES

In order to verify that our algorithms were sound, we implemented a good deal of testing. The test set was usually the full 100 lists of a specific size, and in

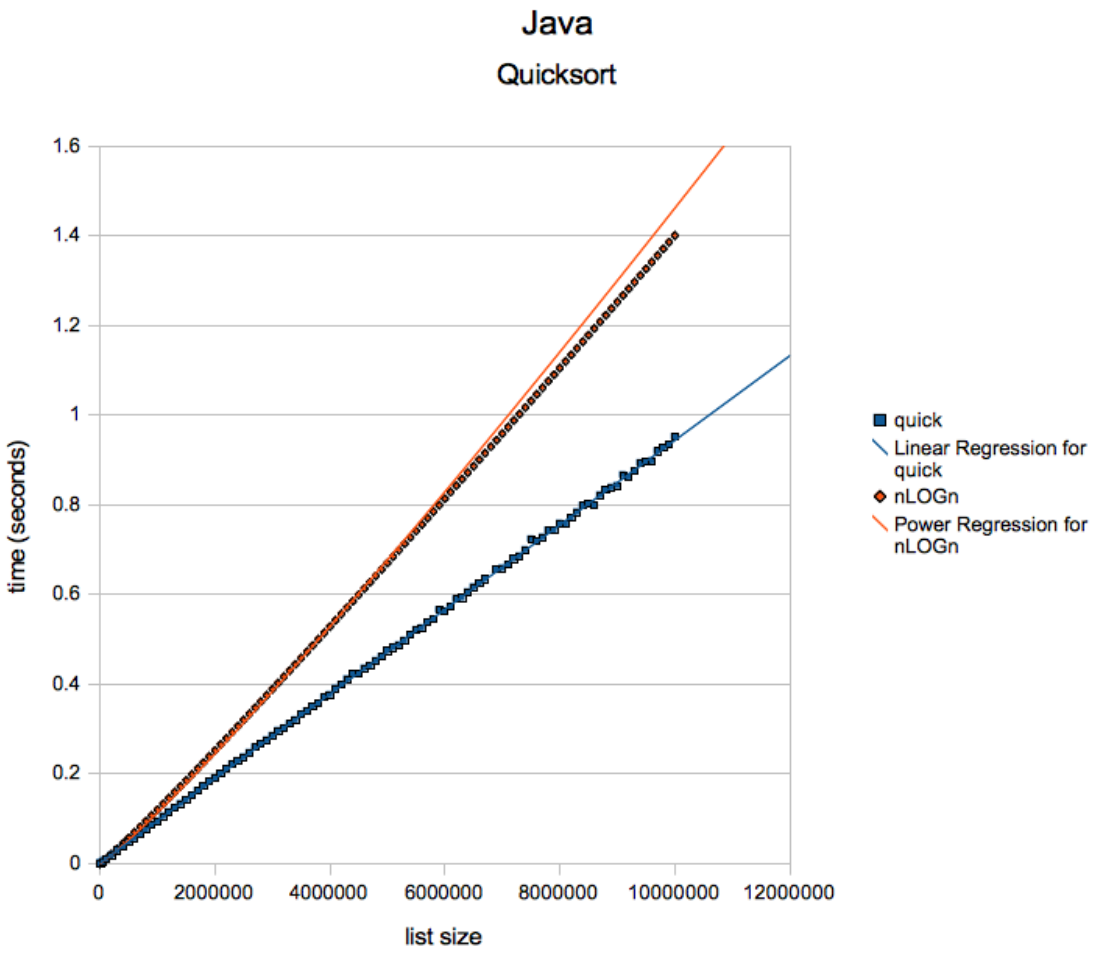

FIGURE 6. Behavior of quick sort for n up to 10,000,000

order to verify that the list in question is actually sorted afterwards, we did a few different things, depending on the language we were working with. Python has a built in function called sort, so we simply checked that the list we sorted was the same as the one Python sorted. In Java there was also a built-in sort function call. In C++ we iterated through the array, making sure that every value was less than its successor.

We decided that it would be good to have a consistent set of lists that are randomly generated to use across all of the sorting algorithms and all of the languages. The reason behind this was our plan of comparing the implementations

across languages. That is, if we wanted to compare one language to another, it would be unfair to compare their performance if they were operating on different data. There were many struggles when we were developing this list generator. It took several tries to figure out how to build up the right amount of data in memory before writing it to a file, because we did not want to spend extra time doing more File I/O than necessary.

At first we started with a very simple program that when given an argument, would create 1000 new files, each containing a list of the size specified in its argument. This worked fine for the first few sizes, so we ran the script in separate shells, with different inputs, to get all of our lists ready. However, the system ran out of memory before it could finish.

Next we decided to write each list to a file, one by one, rather than storing all of the information in memory and attempting to write all of the files at the end. This worked, but it took quite a considerable amount of time. We also had difficulty with the disk space all of these lists took up. We eventually reduced the number of each size of list from 1000 to 100.

Conclusions/Reflections

As the data has clearly shown, and was discussed earlier, Quicksort was the reigning champion of our small collection of algorithms. While we were in class we learned about Merge Sort and how it does not have a worst-case complexity as high as Quicksort does. This led us to believe that Merge Sort would be the fastest of the four, however our results showed that not to be the case in practice. The Merge Sorts we wrote in C++ and Java were actually a different style than all other sorts. They used List objects that constantly changed size and were passed in recursive calls. All of the other algorithms were sorting in place on arrays. If we had taken the time to implement an in place Merge Sort we would have probably seen it show up right next to Quicksort on the graph. Due to time constraints, we

had to accept our results and spend time reflecting on what we wish we had time to improve.

Another thing that we didn't expect was to see Java getting faster times than C++. Because Java runs in a virtual machine, it has a reputation of not running as fast as its hardware-intimate cousin C++. We made sure to run the C++ program with the compiler's optimization flag turned on, and used the latest GNU C Compiler. Most of the algorithms are very similar, and some are identical. It is not fully clear why it didn't perform as well. If we had more time we possibly could have verified the algorithms to be sound by going through several examples together.

As discussed earlier, we had to take our number-of-list size down to 100 which is an example of how complexity theory shows up in practice. We also set out to sort lists of size up to 1,000,000. After running our algorithms for several hours, it became apparent that the programs were not going to terminate in time, most likely due to the worst sorting algorithm we implemented, bubble sort with an average $O(n^2)$ behavior. So, in practice, to prepare usable data, we had to consider the complexity an alter our process so we would have usable data in enough time to run analysis for this project.