Lou George **Sorting Algorithms**
ligeorge@ucsc.edu

Motivation

Sorting algorithms are awesome. They were my favorite topic in my computer science education, and I always found the way they worked to be very satisfying to observe. Another topic I was very interested in is looking at different programming languages side to side and seeing the syntax at work. I figured that these two interests can overlap beautifully with this project.

Doing the project in Haskell, C and Java allowed me to program them in 3 different languages that are drastically different from one another. More importantly it combined a language I don't understand well, a language that I understand somewhat, and a language I know very well. Plus, on top of that I can have a little bank of these algorithms in on my computer with their run speeds for future reference.

The sorting algorithms I chose initially were all classics that are well known and have a good variety of speeds. Quick sort, merge sort, insertion sort, and bubble sort. I chose these because I feel they have different enough runtimes to give some interesting results.

Project Outline

This project was not in any way a fair to the different languages, since they work in unique ways, and instead of trying to write the same algorithm the same way in every language I wanted to find what makes every language unique. To find the truly best way to write an algorithm in a given language can take a very long time, if not impossible. There isn't really a way to prove that an algorithm is written as efficiently or in as few lines as possible for sure, after all there are an infinite way to write each of these algorithms, and each way comes with perhaps some advantages, and with some drawbacks. Looking into all of the possibilities of a given algorithm in even one language would be an overwhelming task for the hours allotted for this project.

The project is mostly based on being able to recreate how a piece of code would be implemented in a given language in the real world by a mid level programmer. Not trying to inch out every millisecond of performance out of it.

Project Breakdown:

First and foremost I needed to make a random number generator, because as good as it sounds to spend a majority of my hours allotted for this project, I really doubt I would have gotten a good grade, nor would it have been fun for anyone to read about how great I am at writing random number.

The other issue was pumping in my random numbers into each algorithm, and in Java and C I did that by parsing the file. I tried to set up a file I/O for Haskell but just ran out of time to do so. I ended up having to just paste arrays of data into an array within the code.

Quicksort:

The first algorithm up was quick sort. It's a simple algorithm that uses a pivot, and compares the next number to the pivot and determines if it should be before it or after it. It continuously does this for the ever increasingly smaller lists until it goes down to a completely sorted list.

Haskell:

In haskell, the main thing that made it very unique is the functional way of defining how to check if a number belongs in smallerSorted or biggerSorted side of the pivot. It was done with quicksorting a, given that a is less than or equal to the pivot. Haskell is also unique in being highly recursive and taking only 6 lines to implement.

Java:

In Java, it was done more functionally, with 2 functions. A swap function and a quicksort function. Quicksort would go through the list of numbers until it can narrow down which 2 numbers needed to be swapped. Then recursively call itself until finishes sorting. I was also able to use the fact that java allows declarations of variables at any point of the function so I could do things that would take 2 lines in C.

C:

In C, quicksort worked similarly, but this time I did it much more recursively than in java. It was still 2 functions, quicksort and split. Quicksort would find the pivot, and then recursively call itself for all numbers less than the pivot, and then for all numbers greater than the pivot. Split would do the bulk of work comparing numbers to the pivot and making sure they're in range, then it would swap them in that function. I felt I could have written a third function but I don't feel it would have made the code any clearer or saved any lines due to declarations being mandatory in C.

Mergesort:

Haskell:

In terms of complexity in haskell I feel this one was the one that took the longest to implement. The mergesort function actually does very little, it splits the list into 2 parts in the function split. Those are then passed over to merge. Merge then takes the two lists and compares the first element in each and combines them into one list as it recursively calls itself. Finally it returns that list back to mergesort which is the sorted list.

Java:

Due to mergesort being a bit older of an algorithm, and not being very complex besides the recursion it's built on I feel like this was the least java looking code I've

written in java in a while. Mergesort keeps calling mergesort recursively, by splitting the list into smaller and smaller halves until it ultimately is down to 1 item per list. Once mergesort gets down to single elements, merge finally gets called, and compares all possible nasty edge cases between the 2 numbers being compared at any given time, and they get put into an array until they reform a sorted array. Java has the advantage here that you have access to the .length option for arrays. This made declaring the array work for any number of items.

C:

The c code was almost identical to that of the java code, and it's because the logic behind it is very similar. On the other hand C is very iffy about using variables to declare the length of an array so I had to make an array of static size, and thus this algorithm would only work for 10,000 numbers. I probably could have done this better with some memory allocation, but it ended up being much more complicated than I expected.

Insertion Sort:

Haskell:

Insertion sort is super simple, which is why I chose it. It generally has a very poor runtime and wanted to see some awful runtime and processors overheating. Fortunately this was the quickest to implement. It works really simply, is takes the first element, and looks for where it belongs, and throws it in there.

Java:

Insertion sort is very simple and very slow in java. You can actually get it in just a few lines, and works exactly like it would in any other language. There really isn't anything too unique that I could add in java, so I guess this one was the real display of language efficiency.

C:

Insertion sort was the same thing that it was in java, and it was quick to implement as well. Again there was nothing I could do to make this unique to C so this one was mostly a performance comparison algorithm.

Bubble Sort

Haskell:

Bubble sort was actually surprisingly difficult to implement. This algorithm was here mostly for fun, because I knew haskell would struggle with it, and it did. But I also had to write it in a very non-haskell esque way. I literally had to implement a loop function which was basically a for loop. The issue here was that I couldn't break down the algorithm to simple end conditions since it's constantly swapping 2 arguments.

Java:

Unlike in haskell, this algorithm was really satisfying to write. It's just 3 super simple functions. The first is just the main bubblesort, it just checks that we went through the entire list, if we haven't bubble up. Bubbleup just checks if the number after the current one is less than it, then swap em. Which is done in swap. This algorithm is very nice because it's mostly just function calls and they're very descriptive and easy to follow what's going on.

C:

For C, it's very similar to java. I actually ended up writing this in C before I did it in haskell, and it's what I used to write the algorithm in haskell. I found that it was very helpful in that respect. It works the same way where bubblesort just goes through the entire list calling bubbleup for each element determining whether swaps are needed or not.

Results

So the upcoming result are a mix of opinion based and some are just raw data. Again the data needs to be taken with a grain of salt, these are algorithms as written by me in what I consider an average level of difficulty. This is in no way the best way to write any of these algorithms, these results are just based on my own abilities and in some cases my own opinion.

Readability:

So I'll include my what I consider most "readable" algorithms from all 3 languages. This comes down to opinion, but to me Java was the most readable, followed by C, and Haskell. As someone who has a strong background in java, I'm obviously biased to this, but at the same time I feel you can be a lot more descriptive and the syntax is far less cryptic.

Bubble sort in Java:

```java
public static void bubbleSort(int[] numbersInt, int total){
    int current = 0;
    while (current < total){
        bubbleUp(numbersInt, current, total);
        current++;
    }
}

public static void bubbleUp(int[] numbersInt, int start, int end){
    for (int index = end; index > start; index--){
        if (numbersInt[index] < numbersInt[index - 1])
            swap(numbersInt, index, index-1);
    }
}

public static void swap(int array[], int index1, int index2){
    int temp = array[index1];
    array[index1] = array[index2];
    array[index2] = temp;
}
```

Quicksort in Haskell

```haskell
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

Merge Sort in C

```c
void mergeSort(int numbers[], int first, int last){
    int middle;

    if (first < last){
        middle = (first + last) / 2;
        mergeSort(numbers, first, middle);
        mergeSort(numbers, middle + 1, last);
        merge(numbers, first, middle, middle + 1, last);
    }
}

void merge(int values[], int leftFirst, int leftLast, int rightFirst, int rightLast)

    int tempArray[10000];
    int index = leftFirst;
    int saveFirst = leftFirst;

    while ((leftFirst <= leftLast) && (rightFirst <= rightLast)){
        if (values[leftFirst] < values[rightFirst]){
            tempArray[index] = values[leftFirst];
            leftFirst++;
        }else{
            tempArray[index] = values[rightFirst];
            rightFirst++;
        }
        index++;
    }
    while (leftFirst <= leftLast){
        tempArray[index] = values[leftFirst];
        leftFirst++;
        index++;
    }
    while (rightFirst <= rightLast){
        tempArray[index] = values[rightFirst];
        rightFirst++;
        index++;
    }
    for (index = saveFirst; index <= rightLast; index++)
        values[index] = tempArray[index];
}
```

They're very clearly very different languages, and that leads to different ways of coding. In haskell you don't need your variables to be very descriptive, because in general you use very few of them, and more happens in fewer lines of code, thus less of them are needed.

On the other hand, in C you should have very descriptive variables, since it can look like a very cryptic language due to the amount of variables you generally end up using. You would want to make the code easy to follow with good variable names.

Lastly Java tends to make itself very Object Oriented, and even though I don't use classes I feel that bubble sort is a great example how powerful functions can be when it comes to clearing code up for someone reading it.

Data:

Lastly, the most important thing is the data. How did these algorithms compare, and here are my results based on 5 sets of 10,000 random integers. Each set was run through the algorithm in all 3 languages. For formatting purposes data is available on the last two pages, but from the data a few interesting points came up.

- First and foremost, C wins. C beat the other languages, and did so consistently. It was always the fastest and had very little difference in runtime between the runs.
- Java was very consistent as well, the run times were never too different. It's initial run time is so high that the algorithm didn't make too big of a difference.
- Haskell was hit or miss. It was either blazingly fast, or tremendously slow. This was mostly because of the nature of the language. Recursion in some cases is actually an awful idea and it can take much longer than a loop. Bubble and insertion sort clearly showed that.
- Bubble sort is awful. Don't bubble sort.

Lines of code:

As we can see, Haskell tends to do way more with way fewer lines of code. I'm sure java I could lower the line count by function chaining, but that would take away from the readability. On the other hand I feel the readability of Haskell doesn't change too much from adding extra lines. C I doubt I could have shortened it up too much without making it impossible to read.

Lines of code

Language	Quick Sort	Merge Sort	Insertion Sort	Bubble Sort
Haskell	6	16	15	20
Java	30	37	15	20
C	32	37	14	19

File Size:

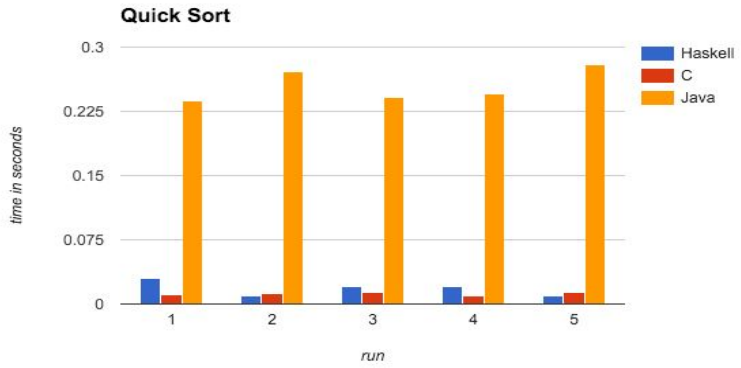
Lastly I looked at file size, which genuinely surprised me. Haskell's files were massive! I don't really understand why that is, I understand that java's files would be small since most of the executable would be in the java VM, but I didn't expect these result for Haskell. C on the other hand came out to roughly what I imagined the results would be:

Language	Quick Sort	Merge Sort	Insertion Sort	Bubble Sort
Haskell	2.4MB	2.5MB	2.4MB	2.6MB
Java	2KB	2KB	2KB	2KB
C	45KB	9KB	9KB	9KB

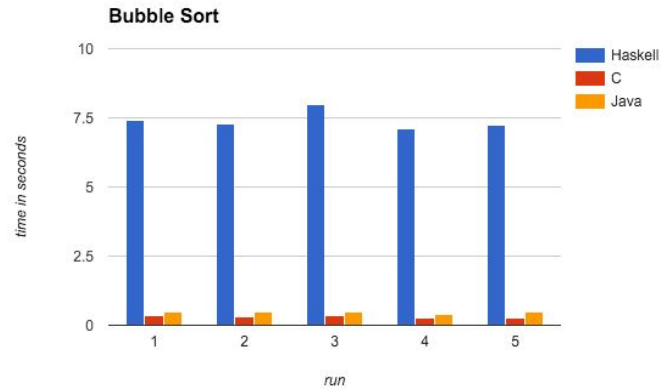
Conclusion:

Sorting algorithms are very strange and their performance widely depends on not just the programmer and what they're sorting, but also the language they choose. This project mostly just showed, that you should pick the right tool for the job. Every language has weaknesses and strengths, and sorting algorithms are just complex enough to make those points shine. But, they're also simple enough to be read by an average programmer and have them understand what's going on within a few minutes. Just another reason sorting algorithms are a great part of computer science to understand and play with.

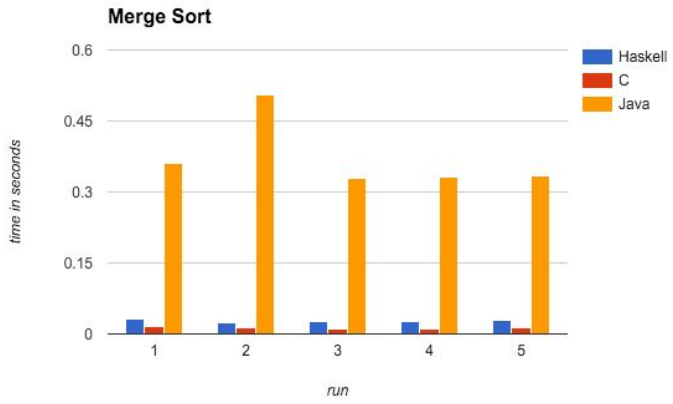
QuickSort			
run	Haskell	C	Java
1	0.03	0.011	0.237
2	0.01	0.012	0.272
3	0.02	0.013	0.242
4	0.02	0.01	0.245
5	0.01	0.014	0.28



BubbleSort			
run	Haskell	C	Java
1	7.393	0.338	0.494
2	7.276	0.325	0.504
3	7.982	0.331	0.485
4	7.112	0.282	0.411
5	7.255	0.277	0.495



MergeSort			
run	Haskell	C	Java
1	0.031	0.015	0.36
2	0.025	0.013	0.505
3	0.026	0.011	0.328
4	0.026	0.01	0.331
5	0.029	0.012	0.333



InsertionSort			
Run	Haskell	C	Java
1	1.879	0.096	0.241
2	1.649	0.09	0.39
3	1.921	0.088	0.252
4	1.901	0.08	0.25
5	1.889	0.093	0.366