Heinz Ulrich Boehmer Fiehn
CSCI 2270

Algorithm Analysis

For this experiment, I compared three array sorting algorithms and tested them on different sized arrays. I decided to compare how many loops and swaps were executed by each, as well as their runtime in milliseconds. The purpose of doing this is to find which algorithm was most efficient and also to see if their efficiency was compromised when the array size increased.

I chose to analyze Radix Sort, Shell Sort, and Quick Sort. I chose the first two because we hadn't seen them in class and thought it would be interesting to see how they worked. I also decided to use Quick Sort because –as its name suggests- it's pretty quick.

Radix sort is special because it does not compare the values of the integers in the array. Instead, it analyzes each digit individually and groups elements that have the same significance. The specific implementation of Radix Sort that I used uses the least significant digit approach, meaning that the end result is an array of integers sorted by their value and not in lexicographical order.

Shell sort is basically an extension of Bubble Sort. However, it has the ability to move elements several positions over, instead of simply swapping neighboring elements. This gives it an advantage over Bubble Sort, as there are less steps involved. Nonetheless, it is still not a very efficient algorithm.

Quick Sort is a pretty efficient sorting algorithm. However, in the worst case, it can take up to $O(n^2)$ operations to complete, which means that it can, in theory, be slower than other algorithms.

To actually compare the algorithms, I decided to carry out four different tests, each time increasing the size of the array by a factor of 10. The first test began with 1,000 elements and the last ended with 1,000,000. Of course, I needed a way to dynamically create unsorted arrays, as I wasn't all too eager about typing these out myself. To do this, I simply made a function that loops the same number of times as the size of the array and inserts a random number into every index. I also decided to limit the number to be between 0 and 999, as this made the data way more readable when debugging and didn't affect the performance of the algorithms in any way. Because these arrays were randomly generated every time the program was run, the elapsed time for the algorithms depended on how long this took. To get around this, I simply measured the time elapsed when generating the array, as well as for each of the three algorithms.

For my first test, I used an array size of 1,000 and ran my code. This produced the following output in the terminal:

```
HeinzBoehmer:Assignment 9 HeinzBoehmer$ ./Assignment9_1k

The array is being generated
Time elapsed: 0.1ms

When ordered with radixSort, the algorithm performs 10029 loops and  6000 swaps.
Time elapsed: 0.162ms

When ordered with shellSort, the algorithm performs 15647 loops and  7632 swaps.
Time elapsed: 0.259ms

When ordered with quickSort, the algorithm performs 8826 loops and  2699 swaps.
Time elapsed: 0.148ms
```

It looks like Quick Sort just barely outperformed Radix Sort. It looped about 1,000 times less, but swapped less than half as much as Radix Sort. Shell Sort performed pretty decently, but it was definitely slower than the other two.

The second test consisted of an array of size 10,000, which produced the following output:

```
HeinzBoehmer:Assignment 9 HeinzBoehmer$ ./Assignment9_10k

The array is being generated
Time elapsed: 0.227ms

When ordered with radixSort, the algorithm performs 100029 loops and  60000 swaps.
Time elapsed: 0.931ms

When ordered with shellSort, the algorithm performs 260228 loops and  140210 swaps.
Time elapsed: 3.101ms

When ordered with quickSort, the algorithm performs 119553 loops and  38901 swaps.
Time elapsed: 1.268ms
```

This time, Radix Sort was significantly faster than Quick Sort, taking about 300ms less. Interestingly, it performed more than 20,000 swaps more than Quick Sort, even though it was faster. Again, Shell Sort was the slowest. It performed pretty badly this time compared to the other two, taking more than 2ms more than Radix Sort to complete.

The third test consisted of an array of size 100,000, which produced the following output:

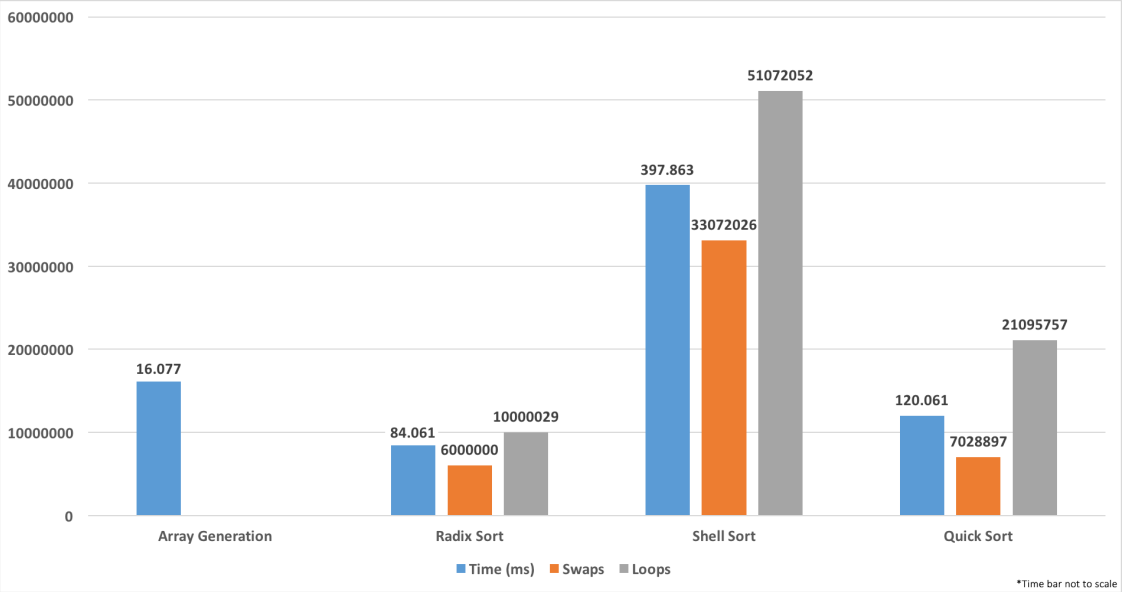
```
HeinzBoehmer:Assignment 9 HeinzBoehmer$ ./Assignment9_100k

The array is being generated
Time elapsed: 1.748ms

When ordered with radixSort, the algorithm performs 1000029 loops and  600000 swaps.
Time elapsed: 9.419ms

When ordered with shellSort, the algorithm performs 3504668 loops and  2004646 swaps.
Time elapsed: 30.073ms

When ordered with quickSort, the algorithm performs 1611672 loops and  535225 swaps.
Time elapsed: 10.346ms
```

This test definitely showed just how more efficient Radix Sort and Quick Sort are when compared to Shell Sort. It took approximately three times as long to complete when compared to the other two. Radix Sort was still more efficient than Quick Sort, but this time the time difference didn't seem to be as bad as last time. This goes to show how Quick Sort's efficiency can vary greatly depending on the array that it is sorting.

For the last test, I decided to graph the results in order to make the difference in efficiency clearer. This test was run with an array size of 1,000,000 and produced the following output:

```
HeinzBoehmer:Assignment 9 HeinzBoehmer$ ./Assignment9_1M

The array is being generated
Time elapsed: 16.077ms

When ordered with radixSort, the algorithm performs 10000029 loops and  6000000 swaps.
Time elapsed: 84.061ms

When ordered with shellSort, the algorithm performs 51072052 loops and  33072026 swaps.
Time elapsed: 397.863ms

When ordered with quickSort, the algorithm performs 21095757 loops and  7028897 swaps.
Time elapsed: 120.061ms
```

Which I then graphed to produce this:



It is clear that Radix Sort was the most efficient algorithm of the three, beating Shell Sort by more than 300ms and Quick Sort by almost 40ms. It also was the algorithm that performed the least number of swaps and loops.

After running this experiment, I found that for the specific sets of data that I used, Radix Sort tended to be the most efficient algorithm, only being beaten by Quick Sort when the array size was 1,000. Although Quick Sort seems to also be a very efficient algorithm, its efficiency seems to be easily affected by the array that is being sorted, while Radix Sort seems to be quite a bit more consistent, regardless of the array used.