Bob Laskowski

Professor Shana Watters

CSC 320

25 April 2016

Analysis of Sorting Algorithms

Introduction

Imagine you have a few thousand dollars in your safe in all different denominations. The bills are all jumbled together but you want to arrange the bills so that you can count it easily. You want the one dollar bills at the front, followed by fives then tens, twenties, fifties and finally hundreds so you can count it easily. You have so much money that it's going to take a while to sort it, so you are wondering what the most efficient way to go about sorting it is. This is an example of what is known as the sorting problem. Luckily, there are many different sorting algorithms to choose from.

The sorting problem has been around since the early days of computer science. Given a list of elements, rearrange them so that they are in a certain desired order. This may be ascending order, descending order, alphabetical order or some other desired arrangement. However, the new permutation must contain all of the original elements and no extras. There are countless ways this objective can be achieved but some are better than others in certain cases or for specific types of data. Since memory and processing speed are finite resources, often what makes one algorithm superior to another is efficiency.

In the computing industry, it is common to have large databases of information. These databases must allow users to search for specific terms and add or delete items. In order to perform these operations in an efficient manner, the data must be sorted. If we had to wait twenty

minutes or more for a database to sort again every time an element is added or removed, a lot of time would be wasted and users would get very frustrated. To remedy this, computer scientists have studied the sorting problem for many years and have come up with efficient solutions that allow millions of items to be sorted in under a second. We will be looking at six different sorting algorithms and trying to determine how they compare to each other.

Methodology

To analyze the sorting algorithms, we will be implementing them in Java using Eclipse Mars 4.5.2 and sorting arrays of integers. There will be five sizes of arrays: 100; 1,000; 10,000; 100,000 and 1,000,000. Each size will have three different arrays: one sorted in ascending order, one sorted in descending order and one pseudo-randomly distributed, all without duplication. The ascending arrays will include the integers 1 up to n, with n being the size of the array. Assume n is the size of the array for the remainder of the paper. The descending arrays will include the integers $n\ down\ to\ 1$. We will generate both of these types of arrays using *for* loops and writing them to files. For the pseudo-randomly distributed permutation we will generate arrays in ascending order using *for* loops with integers $1\ to\ n$. Then we will use Java's Collections.shuffle method on the array seven times to ensure they are as random as possible. Finally, we will write these arrays to files also for use in the experiment.

At the start of testing each different sort we will read the input from the files within the program and put it into arrays to be sorted. This is done so that the arrays are re-initialized to the same values for each sort. When we run the sort we will measure the wall clock time that each algorithm takes to run on each configuration and size of array. To do this we will make a call to System.nanoTime immediately before we begin the sort and then again immediately after so that the sort method call is the only thing that occurs between the timing. The difference between

these two times will be what we record as the running time. We will run three trials of the experiment and take the average of the results from each trial. This average will be the data we use to analyze the algorithms.

We will be executing all three trials of the experiment on my Dell XPS 15 9530 laptop running 64 bit Windows 10 while it is connected to a power supply, on high performance mode and while no other programs are running, with the exception of background system processes. My computer has an Intel Core i7-4712HQ processor at 2.30 GHz, 16.0 GB of RAM and a 512GB solid state drive. We will also edit Eclipse's configuration file, Eclipse.ini, to try to get the best performance possible from Eclipse and ensure that the JVM will not run out of memory when running the recursive sorts. We added the following lines to the file: -XX:+AggressiveOpts, -XX:PermSize=512m, -XX:MaxPermSize=512m, -XX:+UseParallelOld GC, -Xms2048m, -Xmx2048m, -Xmn512m, -Xss2m and -Xverify:none. We will convert the results from nanoseconds to seconds and record my data in a table using Microsoft Excel.

We will use Big-O notation to hypothesize bounds on the running time of our algorithms. The Big-O asymptotically bounds a function from above; in other words, it gives an upper bound on the running time of a function. The formal definition of Big-O is as follows:

> $O(g(n))$ = *{*$f(n)$*: there exist positive constants c and* n_0 *such that* $0 \leq f(n) \leq cg(n)$ *for all* $n \geq n_0$*}*

In this description $f(n)$ is the function that describes the behavior of the algorithm and $g(n)$ is a function that, when multiplied by a constant *c*, is always greater than $f(n)$. This is illustrated in Figure 1. Generally, $g(n)$ is a function similar to $f(n)$ but with only the highest ordered term. For example, if $f(n) = n^2 + 5n + 10$, then $g(n) = n^2$ and we could say $f(n)$ has a $O(n^2)$ because we abstract away from lower ordered terms. We can do this because as *n* gets large, the lower ordered terms have little impact on the overall value of the function. This $f(n)$ and $g(n)$

pair would satisfy the properties of Big-O because we could pick a constant c such that $g(n)$ would always be greater than or equal to $f(n)$. This is an oversimplification and we will see more complicated examples in the analysis of the sorting algorithms below.

Sort Descriptions

In our project, we studied six different sorting algorithms and their efficiency in sorting arrays of integers of sizes 100, 1,000, 10,000, 100,000 and 1,000,000 into ascending order when the input array is sorted in ascending order, descending order and when pseudo-randomly distributed. The sorting algorithms we examined were bubble sort, selection sort, insertion sort, merge sort, heap sort and quick sort. To help understand the different algorithms, we will provide pseudocode and describe each sort before hypothesizing the running time using Big-O notation. We will use the pseudocode to look at the number of operations performed in worst and best case scenarios and how the arrangement of the input data will affect the running time. Throughout this paper, we will assume that the indexing of arrays starts at one, not zero. Also assume that lg denotes the logarithm function with base two.

Bubble Sort

The first sorting algorithm we will look at is bubble sort. It is a simple but inefficient sorting algorithm that works by repeatedly passing through the array and comparing each pair of adjacent elements of an array and if they are out of order, swapping them. On the first pass, elements one and two are compared. If one if bigger than two, they are swapped. Then elements two and three are compared. If they are out of order, they are swapped. It continues until it compares the last two elements and after this last comparison is done, the nth element is in its sorted position. It repeats this process $n - 1$ times because on the last pass through, the first element will also end up in its sorted position, so another pass through would be redundant. In

the following code and for the remainder of the paper t_j denotes the number of times an *if* or *while* statement is executed. The pseudocode for bubble sort:

BUBBLESORT(A)				*Times*	*Cost*
1	**for** $i = 1$ **to** $A.length - 1$			n	c_1
2		**for** $j = A.length$ **downto** $i + 1$		$\sum_{j=0}^{n-2} n - j$	c_2
3			**if** $A[j] < A[j-1]$	$\sum_{j=1}^{n-2} j$	c_3
4			exchange $A[j]$ with $A[j-1]$	$\sum_{j=1}^{n-2} t_j$	c_4

The worst case for bubble sort will occur when it has to enter into the *if* statement on every iteration. This will happen when the array is sorted in descending order because the largest element remaining to be sorted will always start in position one and a swap will occur on every comparison. Since the *if* statement is executed every time, $t_j = j$. We let $T(n)$ denote the running time of the algorithm on an input of size *n*. Therefore, to sort *n* elements, the worst case running time can be calculated as follows:

$$T(n) = c_1(n) + c_2 \sum_{j=0}^{n-2} n - j + c_3 \sum_{j=1}^{n-2} j + c_4 \sum_{j=1}^{n-2} j$$

$$= c_1(n) + c_2\left(\frac{1}{2}n^2 + \frac{1}{2}n - 1\right) + c_3\left(\frac{1}{2}n^2 - \frac{3}{2}n + 1\right) + c_4\left(\frac{1}{2}n^2 - \frac{3}{2}n + 1\right)$$

$$= \left(\frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{2}\right)n^2 + \left(c_1 + \frac{c_2}{2} - \frac{3c_3}{2} - \frac{3c_4}{2}\right)n + (-c_2 + c_3 + c_4)$$

If we let $a = \frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{2}$, $b = c_1 + \frac{c_2}{2} - \frac{3c_3}{2} - \frac{3c_4}{2}$ and $c = -c_2 + c_3 + c_4$, the worst case for bubble sort will have a running time of $T(n) = an^2 + bn + c$, which we say has a $O(n^2)$ because from the definition of Big-O, we abstract away from all but the highest ordered term in the function. The best case should occur when the input array is already sorted. This is because the *if* statement will never be entered and no swaps will occur. Therefore, $t_j = 0$ in this case and the running time can be calculated as follows:

$$T(n) = c_1(n) + c_2 \sum_{j=0}^{n-2} n - j + c_3 \sum_{j=1}^{n-2} j + c_4 \sum_{j=1}^{n-2} 0$$

$$= c_1(n) + c_2 \left(\frac{1}{2}n^2 + \frac{1}{2}n - 1\right) + c_3 \left(\frac{1}{2}n^2 - \frac{3}{2}n + 1\right) + c_4(0)$$

$$= \left(\frac{c_2}{2} + \frac{c_3}{2}\right) n^2 + \left(c_1 + \frac{c_2}{2} - \frac{3c_3}{2}\right) n + (-c_2 + c_3)$$

As we can see from this calculation, if we let $a = \frac{c_2}{2} + \frac{c_3}{2}$, $b = c_1 + \frac{c_2}{2} - \frac{3c_3}{2}$ and $c = -c_2 + c_3$ then the best case for bubble sort will have a $T(n) = an^2 + bn + c$, which is a $O(n^2)$. Therefore, we hypothesize that the best case will occur when the input array is sorted in ascending order and will also have a running time of $O(n^2)$. The worst case will occur when the input array is sorted in descending order and will have a running time of $O(n^2)$. The pseudo-random array should have a $O(n^2)$ because some swaps will still occur, but take less time than the descending order array because swaps are not made on every single comparison. Thus we hypothesize that all runs of bubble sort will have $O(n^2)$, but will differ by a constant factor.

Selection Sort

Selection sort is an in-place comparison sorting algorithm that looks through an array on each pass and finds the smallest value. It swaps that smallest value with the first element of the array and continues making passes until it has performed one swap for every element in the array. Thus the smallest not yet sorted element is placed into its sorted position on each pass. The algorithms makes $n - 1$ passes through the array because the final element is already in its sorted place after $n - 1$ pass. The pseudo code for selection sort:

SELECTIONSORT(A)		*Times*	*Cost*
1	**for** $i = 1$ **to** $A.length - 1$	n	c_1
2	$int\ j = i$	$n - 1$	c_2
3	**for** $k = i$ **to** $A.length$	$\sum_{k=2}^{n} k$	c_3
4	**if** $A[k] < A[j]$	$\sum_{k=1}^{n-1} k$	c_4
5	$j = k$	$\sum_{k=1}^{n-1} t_k$	c_5
6	exchange $A[j]$ with $A[i]$	$n - 1$	c_6

The worst case for selection sort will occur when the input array is sorted in descending order. This happens because the *if* statement inside the second *for* loop will be entered every time and the assignment on line 5 will occur on every pass through the loop. In this case, the running time can be calculated as follows:

$$T(n) = c_1(n) + c_2(n-1) + c_3\sum_{k=2}^{n} k + c_4\sum_{k=1}^{n-1} k + c_5\sum_{k=1}^{n-1} k + c_6(n-1)$$

$$= c_1(n) + c_2(n-1) + c_3\left(\frac{1}{2}n^2 + \frac{1}{2}n - 1\right) + c_4\left(\frac{1}{2}n^2 - \frac{1}{2}n\right) + c_5\left(\frac{1}{2}n^2 - \frac{1}{2}n\right) + c_6(n-1)$$

$$= \left(\frac{c_3}{2} + \frac{c_4}{2} + \frac{c_5}{2}\right)n^2 + \left(c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} - \frac{c_5}{2} + c_6\right)n + (-c_2 - c_3 - c_6)$$

If we let $a = \left(\frac{c_3}{2} + \frac{c_4}{2} + \frac{c_5}{2}\right)$, $b = \left(c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} - \frac{c_5}{2} + c_6\right)$ and $c = (-c_2 - c_3 - c_6)$, then $T(n) = an^2 + bn + c$ and we can abstract away from the lower ordered terms and hypothesize that the worst case running time of selection sort has a $O(n^2)$. The best case running time of selection sort occurs when the input array is already sorted in ascending order because the *if* statement on line 4 will never be entered, so we will never execute line 5. The running time for this case can be calculated as follows:

$$T(n) = c_1(n) + c_2(n-1) + c_3\sum_{k=2}^{n} k + c_4\sum_{k=1}^{n-1} k + c_5(0) + c_6(n-1)$$

$$= c_1(n) + c_2(n-1) + c_3\left(\frac{1}{2}n^2 + \frac{1}{2}n - 1\right) + c_4\left(\frac{1}{2}n^2 - \frac{1}{2}n\right) + c_5(0) + c_6(n-1)$$

$$= \left(\frac{c_3}{2} + \frac{c_4}{2}\right)n^2 + \left(c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} + c_6\right)n + (-c_2 - c_3 - c_6)$$

As we can see, the best case has a similar running time to the worst case, just one constant term is excluded. If we let $a = \left(\frac{c_3}{2} + \frac{c_4}{2}\right)$, $b = \left(c_1 + c_2 + \frac{c_3}{2} - \frac{c_4}{2} + c_6\right)$ and $c = (-c_2 - c_3 - c_6)$, then $T(n) = an^2 + bn + c$ and we can once again abstract away from the lowered ordered terms and hypothesize that the best case also has a $O(n^2)$. From this information we can hypothesize that the pseudo-randomly distributed input array will have a running time between the ascending and descending input arrays because the *if* statement on line 4 will be executed some of the time but not all the time.

Insertion Sort

Insertion sort is often compared to the way a hand of cards is sorted. If we begin with a pile of cards face down on the table and pick up one card at a time and insert it into its proper position in the hand, we are performing an insertion sort. It begins by assuming the first item in the list, position one, is already sorted. Then it takes the first item in the unsorted list and compares it to each item in the sorted list until it finds a smaller item or hits the end of the list to determine where it should be inserted. It repeated this process $n - 1$ times because the first item is assumed to already be sorted since a one item list is sorted. Insertion sort works well with small input sizes and with nearly sorted arrays. The pseudocode for insertion sort:

INSERTIONSORT(A)	*Times*	*Cost*
1 **for** $j = 2$ **to** $A.length$	n	c_1
2 key = $A[j]$	$n - 1$	c_2
3 $i = j - 1$	$n - 1$	c_3
4 **while** $i > 0$ and $A[i] > key$	$\sum_{j=2}^{n} t_j$	c_4
5 $A[i + 1] = A[i]$	$\sum_{j=2}^{n} t_j - 1$	c_5
6 $i = i - 1$	$\sum_{j=2}^{n} t_j - 1$	c_6
7 $A[i + 1] = key$	$n - 1$	c_7

The worst case for insertion sort occurs when the input array is sorted in descending order. This happens because it will have to enter the *while* loop on every iteration of the *for* loop and execute the two assignment statements. We can calculate the worst case running time as follows:

$$T(n) = c_1(n) + c_2(n-1) + c_3(n-1) + c_4\sum_{j=2}^{n} j + c_5\sum_{j=2}^{n} j - 1 + c_6\sum_{j=2}^{n} j - 1 + c_7(n-1)$$

$$= c_1(n) + c_2(n-1) + c_3(n-1) + c_4\left(\frac{1}{2}n^2 + \frac{1}{2}n - 1\right) + c_5\left(\frac{1}{2}n^2 - \frac{1}{2}n\right) + c_6\left(\frac{1}{2}n^2 - \frac{1}{2}n\right)$$
$$+ c_7(n-1)$$

$$= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n + (-c_2 - c_3 - c_4 - c_7)$$

If we let $a = \frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}$, $b = c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7$ and $c = -c_2 - c_3 - c_4 - c_7$, we can write $T(n) = an^2 + bn + c$ and hypothesize that the worst case for insertion sort will have a $O(n^2)$. The best case for insertion sort will occur when the input array is already sorted in ascending order. This happens because the *while* statement never has to be entered and thus we will have no nested loops. We will still have to check the conditions of the *while* loop so in this case $t_j = 1$. The running time for the best case can be calculated as follows:

$$T(n) = c_1(n) + c_2(n-1) + c_3(n-1) + c_4\sum_{j=2}^{n} 1 + c_5(0) + c_6(0) + c_7(n-1)$$

$$= c_1(n) + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

$$= (c_1 + c_2 + c_3 + c_4 + c_7)n + (-c_2 - c_3 - c_4 - c_7)$$

In this case if we let $a = c_1 + c_2 + c_3 + c_4 + c_7$ and $b = -c_2 - c_3 - c_4 - c_7$, then we have $T(n) = an + b$. Instead of a quadratic equation, we have a linear one this time. Thus we hypothesize that the best case for insertion sort has $O(n)$. The pseudo-randomly distributed input

array will have to enter the *while* loop some of the time so this will lead to a quadratic equation and thus we hypothesize a $O(n^2)$. However, it should take less time than the descending order input array because it will only have to enter the *while* loop some of the time instead of on every iteration.

Merge Sort

Our next sorting algorithm, merge sort is a divide and conquer recursive sort algorithm. Divide and conquer entails dividing the original problem, in this case the array to be sorted, into sub problems that are smaller instances of the original problem. Then we conquer the sub problems using recursion. When we are done with this, we combine the solutions to the sub problems to get the solution to the original problem. Merge sort works by picking an element in the middle of the array and then calling itself on each half. It continues with recursive calls until each subarray is of length 1, which is the base case. It then calls the merge function to combine each of the subarrays but in sorted order using comparisons. The pseudocode for merge and merge sort:

MERGESORT(*A,p,r*)
1 **if** $p < r$
2 $q = \lfloor (p + r)/2 \rfloor$
3 MERGESORT(*A,p,q*)
4 MERGESORT(*A,q+1,r*)
5 MERGE(*A,p,q,r*)

MERGE(*A,p,q,r*)
1 $n_1 = q - p + 1$
2 $n_2 = r - q$
3 let $L[1 \dots n_1 + 1]$ and $R[1 \dots n_2 + 1]$ be new arrays
4 **for** $i = 1$ **to** n_1
5 $L[i] = A[p + i - 1]$
6 **for** $j = 1$ **to** n_2
7 $R[j] = A[q + j]$
8 $L[n_1 + 1] = \infty$
9 $R[n_2 + 1] = \infty$
10 $i = 1$

```
11      j = 1
12      for k = p to r
13          if L[i] ≤ R[j]
14              A[k] = L[i]
15              i = i + 1
16          else A[k] = R[j]
17              j = j + 1
```

In the merge sort call, p is the first element of the input array and r is the last element. The divide portion of this divide and conquer algorithm occurs on line 2 of the pseudocode. The input array is divided as close to in half as possible. It then uses recursion to solve the sub problems by repeatedly dividing them in half until it reaches the base case where $p \geq r$. When this is done, the merge function is called to combine the individual solutions into the answer to the original problem. The outcome is an array sorted in ascending order. Since merge sort always divides the input array in half regardless of the arrangement of the elements, we predict that its performance will not be affected by the different permutations of the input arrays. From discussion in Introduction to Algorithms, we know that the recurrence equation the describes the best and worst case of merge sort is $T(n) = \begin{cases} c & if\ n = 1 \\ 2T(n/2) + cn & if\ n > 1 \end{cases}$ (Cormen 35-37). Solving this equation leads us to hypothesize a $O(nlgn)$ for all cases of merge sort that differ by only constant factors.

Heap Sort

Our next sorting algorithm, heap sort, is different than the sorts we have looked at so far. It uses a heap data structure which is similar to a nearly complete binary tree except where the parent of every node is greater than both its children. It begins by first putting the input array into a max-heap using the build-max-heap function. It then uses the max heap to sort the array by swapping the first and last element because the first element on the tree is the smallest since it satisfies the max-heap property of every parent being greater than both its children. Then it puts

the array back into a max heap using the max-heapify function and repeats this process for one less than the number of elements in the array. The pseudocode for these three functions is as follows:

HEAPSORT(A)	*Times*	*Cost*
1 BUILD-MAX-HEAP(A)	1	$O(n)$
2 **for** $i = A.length$ **downto** 2	n	c_2
3 exchange $A[1]$ with $A[i]$	$n-1$	c_3
4 $A.heapsize = A.heapsize - 1$	$n-1$	c_4
5 MAX-HEAPIFY(A,1)	$n-1$	$O(lgn)$

BUILD-MAX-HEAP(A)
1 $A.heapsize = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3 MAX-HEAPIFY(A,*i*)

MAX-HEAPIFY(A,*i*)
1 *l* = LEFT(*i*)
2 *r* = RIGHT(*i*)
3 **if** $l \leq A.heapsize$ and $A[l] > A[i]$
4 $largest = l$
5 **else** $largest = i$
6 **if** $r \leq A.heapsize$ and $A[r] > A[largest]$
7 $largest = r$
8 **if** $largest \neq i$
9 exchange $A[i]$ with $A[largest]$
10 MAX-HEAPIFY(A, *largest*)

The costs of build-max-heap and max-heapify are discussed in Intro. to Algorithms (Cormen 155-159). From there I learned that build-max-heap has $O(n)$ and max-heapify has $O(lgn)$. The actual heap sort algorithm is not affected by the order of the input data, but max-heapify is. Build-max-heap is also affected, but only because it calls max-heapify. The worst case for max-heapify occurs when the bottom level of the tree is exactly half full. However, both the best and worst case of max-heapify have $O(n)$, so we can calculate the running time of heap sort as follows:

$$T(n) = O(n)(1) + c_2(n) + c_3(n-1) + c_4(n-1) + O(lgn)(n-1)$$

$$= O(n) + O(nlgn - lgn) + (c_2 + c_3 + c_4)n + (-c_3 - c_4)$$

We can abstract away from lowered ordered terms and conclude that the best and worst case running times of heap sort are $O(nlgn)$. Thus, regardless of the distribution of the input array, we hypothesize that the running times for all three permutations will be similar, differing by only a constant factor.

Quick Sort

Quick sort is a recursive sorting algorithms, meaning that it calls itself inside the method. It begins by dividing the original array into two smaller arrays based on a pivot value with a partition method that is separate from the actual sorting algorithm. All elements less than the pivot are placed in one subarray and all elements greater than or equal to are placed in another. It continues partitioning the array through recursive calls with new pivot values until it reaches the base case of an array of size zero or one, which do not need to be sorted. They do not need to be sorted because an array of size zero or one is already sorted. After each recursive call, the element used as the pivot is in its sorted position. Therefore, at the end of the recursion, the array is sorted. The pseudocode for quick sort and partition:

```
QUICKSORT(A,p,r)
1    if p < r
2        q = PARTITION(A,p,r)
3        QUICKSORT(A,p,q-1)
4        QUICKSORT(A,q+1,r)

PARTITION(A,p,r)
1    x = A[r]
2    i = p - 1
3    for j = p to r - 1
4        if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7    exchange A[i + 1] with A[r]
```

8 **return** $i + 1$

In the original call to quick sort *p* is the first element of the array and *r* is the last element. The selection of the pivot value is the determining factor for the performance of the algorithm. If the value selected as the pivot is the smallest or largest value in the array, the partition will divide into arrays of size 0 and size $n - 1$ instead of two arrays of size $n/2$, which is ideal. This worst case will occur when the array is already sorted in either ascending or descending order. From discussion in Intro. to Algorithms, we know that $T(n) = T(n - 1) + O(n)$ for the worst case and $T(n) = 2T(n/2) + O(n)$ for the best case (Cormen 170-178). The book uses the master method to solve this recurrence and determine the big-O. Based on these calculations, we hypothesize that for the pseudo-randomly distributed array, quick sort will have a running time of $O(nlgn)$. In the worst case, when the array is already sorted in either ascending or descending order, we hypothesize it will have a running time of $O(n^2)$.

Hypothesis Summary

Having now looked at all the sorts we will be testing in detail, we can formulate more general hypothesis as to how we expect their running times to compare. In the following chart we see the big-O for all of the algorithms in their best, worst and average case.

Sort:	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Heap Sort	Quick Sort
Best:	$O(n^2)$	$O(n^2)$	$O(n)$	$O(nlgn)$	$O(nlgn)$	$O(nlgn)$
Worst:	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(nlgn)$	$O(nlgn)$	$O(n^2)$
Average:	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(nlgn)$	$O(nlgn)$	$O(nlgn)$

We expect bubble sort to be the slowest overall. It is the most basic sort and has two embedded *for* loops, so we expect the $O(n^2)$ running time to be the slowest in all cases. We expect selection sort to be slightly faster than bubble sort but since it still has $O(n^2)$, we predict it will still be quite slow as *n* grows large. The worst case of insertion sort may take longer than

selection sort, but since they are both $O(n^2)$ with different constant costs, it is difficult to say for sure. The best case of insertion sort, when the array is already sorted, is what we expect to be the fastest overall sort, since it is our only algorithm that sorts in linear time. We expect heap sort, quick sort and merge sort to be comparable, with the exception of quick sort's worst case, when the array is sorted in ascending or descending order, since it is $O(n^2)$. We expect this case to be similar to selection and bubble sort. For heap sort, merge sort and the best and average cases of quick sort, we expect all of the running times to be comparable, differing by only a constant factor. We predict merge sort and heap sort to be the fastest in general, since they are not dependent on the arrangement of the input data.

Results

After running three trials of the experiment and calculating the average running time of each of the sorts for all of the different array sizes and arrangements of input arrays, we gathered a lot of data. Using Microsoft Excel to help organize all of the data, a complete set of the results is displayed in Table 1. This table shows the running times of each sort for each array in seconds. For the most part, the sorts performed as we expected them to with few exceptions. With each jump in size of array, we multiplied the size by ten. For example, $100 * 10 = 1{,}000$; $1{,}000 * 10 = 10{,}000$; et cetera. Thus, for a sort that runs in linear time, meaning it has $O(n)$, we would expect the running time to be multiplied by at most a factor of ten with each increase in array size, since the big-O is an upper bound. For a sort with $O(n^2)$, we would expect the running time to increase by a factor of at most one hundred.

Bubble Sort

For bubble sort, we expected the array sorted in ascending order to be the best case, the array sorted in descending order to be the worst case and pseudo-random to be in between. All

cases were $O(n^2)$, so we expected them all to have similar running times, differing by a constant factor. We can see the exact results of running bubble sort in Table 1 and a more general graphical representation Figure 2. For the ascending arrays of size 100, 1,000 and 10,000, the running time only increased by a factor of ten on each run, which was within the big-O bound we expected. Recall that according to our hypothesis it should have increased by less than a factor of one hundred. Going up to arrays to size 100,000 and 1,000,000, we can see that the running time did increase as expected, by approximately a factor of one hundred.

For the descending arrays we saw similar behavior. The running time for arrays of size 100 and 1,000 increased by a factor of ten. Between 1,000 and 10,000, the increase was more than a factor of ten but less than one hundred, by approximately a factor of forty. Moving up to arrays of size 100,000 and 1,000,000, the increase in running time was almost as predicted: a factor of one hundred. The pseudo-random array displayed the closest to predicted behavior. From 100 to 1,000 elements, the running time increased by approximately a factor of forty. However, moving up to 10,000 elements the running time only increased by approximately a factor of twenty. I believe this discrepancy is caused by the different arrangement of random numbers in the different sizes of arrays. Moving up to arrays of size 100,000 and 1,000,000, the behavior was within the expected bound, an increase by a factor of approximately one hundred.

The most surprising result of the bubble sort run was the fact that the pseudo-randomly sorted array took longer on all array sizes, except 100, than the array sorted in descending order, which we hypothesized would be the worst case. I believe this unexpected result was caused by an attribute of the computer's processor called branch prediction in which the processor attempts to predict whether it will execute the *if* statement within the code or not. When it is sorting the array of descending order, it can accurately make this prediction because it has to enter the *if*

statement every time. However, on the pseudo-randomly sorted array, sometimes it has to execute the *if* statement and sometimes it does not, so this slows down the processing speed. As a result, the pseudo-randomly sorted array took much longer to sort than the descending order array.

Selection Sort

For selection sort, we expected similar results to what we expected from bubble sort. We expected a running time of $O(n^2)$ and the ascending array to be the best case and descending array to be the worst case, with the pseudo-random array somewhere in between. In Table 1 we can see the exact results of the running times of selection sort in seconds and a more graphical representation in Figure 3. For the ascending array and sizes of 100 and 1,000, we saw an increase by a factor of approximately twenty. Strangely, going from size 1,000 to 10,000, the increase was only approximately a factor of five. Then going from 10,000 to 100,000, there was a dramatic increase in running time by a factor of approximately two hundred and fifty. But then going up to 1,000,000, the increase was by a factor of one hundred as we would expect for a $O(n^2)$ running time algorithm. The behavior of this sorting algorithm for small values was not exactly as we expected, but as n grew large, it was exactly as expected.

For the descending arrays we saw similar behavior with large values behaving the way we expected and small values displaying some strange behavior. For arrays of size 100 to 1,000, the increase in running time was approximately a factor of fifteen. Then moving up to 10,000, the increase was a factor of approximately twenty. For the larger values of 100,000 and 1,000,000, the running time increased by a factor of one hundred each time. The pseudo-random array's behavior was almost identical to the ascending array. The difference in running times between the different arrangement of arrays was as we predicted. The ascending array was the

best case and the descending array was the worst case. The pseudo-random array's running time was between the other two arrangements for all test sizes, as we predicted.

Insertion Sort

Insertion sort was the sort we expected to be the fastest for arrays sorted in ascending order and our tests proved this to be true. The exact results for the running times can be seen in Table 1 and this data is illustrated graphically in Figure 4. The best case of insertion sort was the only sort we expected to run in linear time, $O(n)$. This corresponds to an increase by a factor of at most ten when increasing the input array size by ten. From 100 to 1,000 element arrays, the increase in running time was exactly as we predicted, a factor of ten. Interestingly, the increase from 1,000 to 10,000 was only approximately a factor of two. Then going up to 100,000, the increase was approximately a factor of three. However, at 1,000,000, the increase was once again almost exactly a factor of ten. We found it interesting that the middle sizes of the arrays had smaller increases in running time but the first and last had an increase by a factor of ten. We were unable to determine the exact cause of this behavior but we know that a big-O is an upper bound, so it makes sense that some increases were less than ten but the biggest increase was at most ten. This leads us to believe that insertion sort is the best sort to use when we know that there is a high probability that the input data is already sorted.

We predicted the running times of insertion's worst and average case to be similar to that of section and bubble sort, since we predicted these cases would also have $O(n^2)$. The behavior was also exactly as predicted. For all input sizes the descending array took the longest and the pseudo-random array's running time was between that of ascending and descending. For descending going from size 100 to 1,000, the running time increased by a factor of approximately twenty-five. For 10,000, the increase was approximately a factor of twenty. Going

up to 100,00, the running time increased by approximately a factor of thirty-five. Then finally going up to 1,000,000, the increase was a factor of one hundred like we predicted.

For the pseudo-random array, the results were similar. The first jump from 100 to 1,000 was approximately a factor of ten. The next jump up to 10,000 was approximately a factor of twenty. The subsequent two jumps to 100,000 and 1,000,000 were by a factor of one hundred, as we predicted. Like selection sort and bubble sort, the increase in running time for smaller input sizes was not as large as we predicted it to be.

Merge Sort

Merge sort was one of the best sorts overall, especially for large input sizes. It has a running time of $O(nlgn)$ for best, worst and average case. When the input size increases by a factor of ten, $10 * \log_2 10 \approx 33.219$ is the factor that we expect to see the running time increase by. We hypothesized that the running time of merge sort is not dependent on the distribution of the data. Our results reflected this for the most part. The exact results can be seen in Table 1 and they are illustrated graphically in Figure 5. For four out of five input sizes, pseudo-random took slightly longer than the other two arrangements. However, for size 1,000, the ascending array took the longest. Since the big-O is an upper bound, the running time never actually increased by a factor of 33.219, as we see reflected in our results.

For the input arrays of size 100 and 1,000, the ascending increased by approximately a factor of five while the descending and pseudo-random arrays increased by only about a factor of two. For the arrays of size 10,000, the ascending increased by approximately a factor of three, the descending increased by a factor of about seven and the pseudo-random increased by about ten, all less than 33.219. For both 100,000 and 1,000,000, all of the increases were approximately a factor of ten. I speculate that as *n* grows larger and larger, the running time increase would get

closer to the hypothesized $O(nlgn)$. One thing to note is that merge sort was the fastest overall sort in general for the arrays of size 1,000,000, with insertion sort beating it for the ascending array and quick sort beating it for the pseudo-random array. From this data we conclude that merge sort is the most versatile of the sorts that we tested and would be a good choice if we have no prior knowledge of how the data to be sorted will be distributed.

Heap Sort

Heap sort's performance was very similar to that of merge sort. We also hypothesized that heap sort would have a $O(nlgn)$ running time for all of the best, worst and average case scenarios. As with merge sort, this would also translate to an upper bound of approximately 33.219 in the increase in running time between array sizes. Also, as with merge sort, we hypothesized that heap sort's running time would not be affected by the distribution of the data. Whether the array is in ascending, descending or pseudo-random order, we predicted the running times should be very similar. The exact results are shown in Table 1 and illustrated graphically in Figure 6.

For the smaller size arrays of 100, 1,000 and 10,000, the running times increased by a factor of two or three for all the different distributions of the data. For 100,000 and 1,000,000, there was an increase by a factor of ten. This performance was very comparable to that of merge sort. For array sizes up to 100,000, heap sort actually beat merge sort. This leads up to conclude that heap sort is the best sort to use for smaller input sizes when we do not know how the data will be distributed.

Quick Sort

Quick sort is the final sort we tested. We hypothesized that the best and average case of quick sort would be $O(nlgn)$, while the worst case would be $O(n^2)$. We believed the worst case

would occur when the array was sorted in either ascending or descending order while the best case would occur when the array was as randomly distributed as possible. The average case occurs for semi-random input distributions. Therefore, our ascending and descending arrays should produce the worst case and the pseudo-random array should produce an average or best case. Our test results can be seen in Table 1 and illustrated graphically in Figure 7.

The ascending and descending arrays displayed very similar results in our tests, as we predicted since they both have $O(n^2)$. There was a factor of ten increase for both when going from 100 to 1,000 and 1,000 to 10,000. Then going to 100,000 and 1,000,000, the increase was by approximately a factor of one hundred. Quick sort's best case performance is much better for our pseudo-randomly sorted array. For the increases in array size, there was an increase in running time by a factor of ten between all of them. For the random array, it performed better than heap sort and merge sort on all sizes. This leads us to conclude that quick sort would be the best sort to choose if we know that our input data will be randomly distributed.

Conclusion

So what would be the best way to sort our large pile of money? From our original description of the problem, we know the money is in no particular order. From our analysis, we learned that quick sort is the fastest for sorting data in random order. In this case, we knew something about the arrangement of the input data, so this gave us a big advantage in choosing how to sort the input. However, we do not always have this information. If we do not know anything about the data to be sorted, we learned that we are better off using heap sort or merge sort.

Our results, summarized in Table 1, can be seen graphically in Figures 8 – 12. These graphs compare the running times of each of the six sorting algorithms for all of the different

input array sizes. These graphs make it very easy to see how the different sorts compare to each other. Almost all of the algorithms displayed the behavior we hypothesized. The one exception was bubble sort taking longer to sort the pseudo-random array than the descending array. We believe this was due to the branch prediction property of the computer's processor. All the other sorts performed as we expected them to. This leads us to believe that the big-O is a very good tool for predicting efficiency of the running time of algorithms.

This project forced us to look in depth at different ways to solve a problem and seek out the most efficient approach. Often, a brute force method is the easiest to conceptualize for a program, but these approaches are generally not scalable to larger problems. This is where the power of algorithms comes into play, which we saw firsthand as our input array sizes grew large. When working with larger problems, we are forced to look at aspects of the computer we usually do not worry about for small programs, such as memory and running time efficiency. In conclusion, this project was all sorts of fun.

Figures

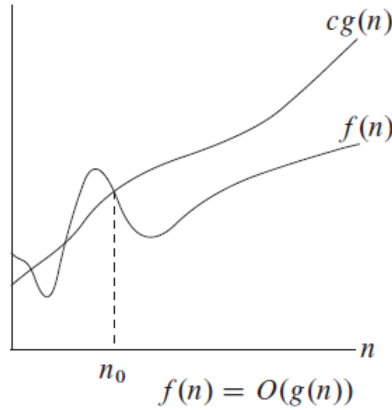

Figure 1 – Big O Graph (Cormen 45)

	Bubble	Selection	Insertion	Merge	Heap	Quick
100 Ascending	0.000122123	0.000122569	0.000005801	0.000116917	0.000125990	0.000230709
100 Descending	0.000294225	0.000130155	0.000149790	0.000121230	0.000106802	0.000181176
100 Random	0.000238742	0.000123759	0.000076308	0.000127329	0.000093860	0.000016511
1,000 Ascending	0.003151394	0.003138452	0.000050574	0.000596334	0.000307166	0.002404079
1,000 Descending	0.002283444	0.001956494	0.004026483	0.000270723	0.000276970	0.002256818
1,000 Random	0.008251695	0.001922579	0.000972966	0.000279648	0.000311629	0.000147261
10,000 Ascending	0.033006334	0.015392235	0.000091183	0.001657955	0.003157343	0.074512030
10,000 Descending	0.087250139	0.045703836	0.088969825	0.001565433	0.001760294	0.060224700
10,000 Random	0.159235372	0.039780952	0.015828516	0.002329854	0.002202971	0.000960620
100,000 Ascending	3.338848581	3.843916613	0.000314604	0.012982802	0.014741311	7.788069323
100,000 Descending	8.753156601	4.496642037	3.070005967	0.011288106	0.011538004	6.003555557
100,000 Random	19.248969309	3.941484394	1.523053099	0.019774824	0.017848229	0.011306551
1,000,000 Ascending	334.686938602	388.069812272	0.003195721	0.117029813	0.122328545	778.336497701
1,000,000 Descending	875.033545626	457.284102098	307.787082603	0.115410681	0.126052324	604.870652175
1,000,000 Random	1915.276634493	389.161435611	153.113520948	0.217893448	0.234958365	0.135017270

Table 1 – Results of Experiment, running time of sorts in seconds

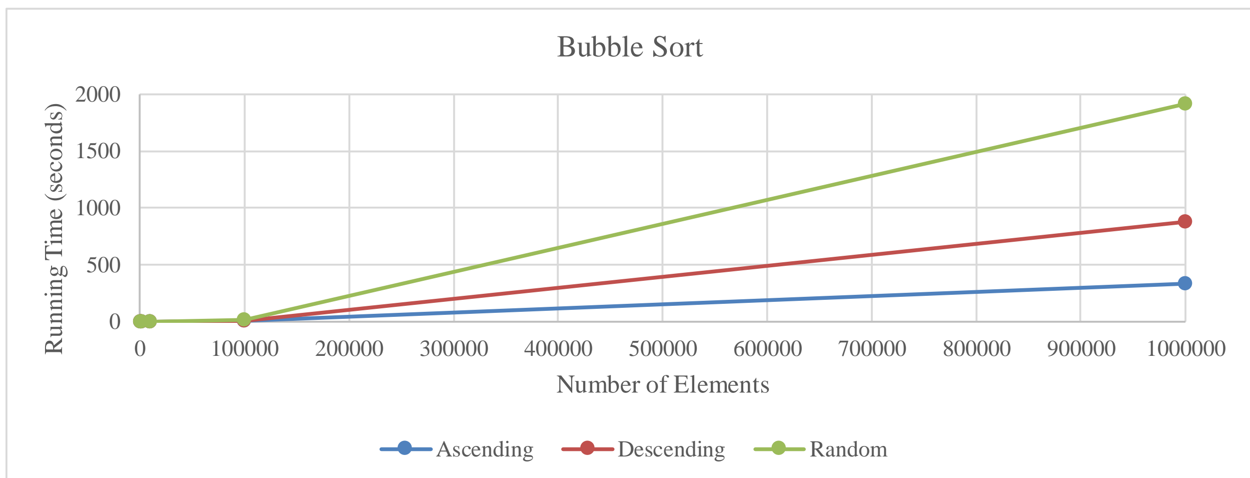

Figure 2 – Running times of Bubble Sort

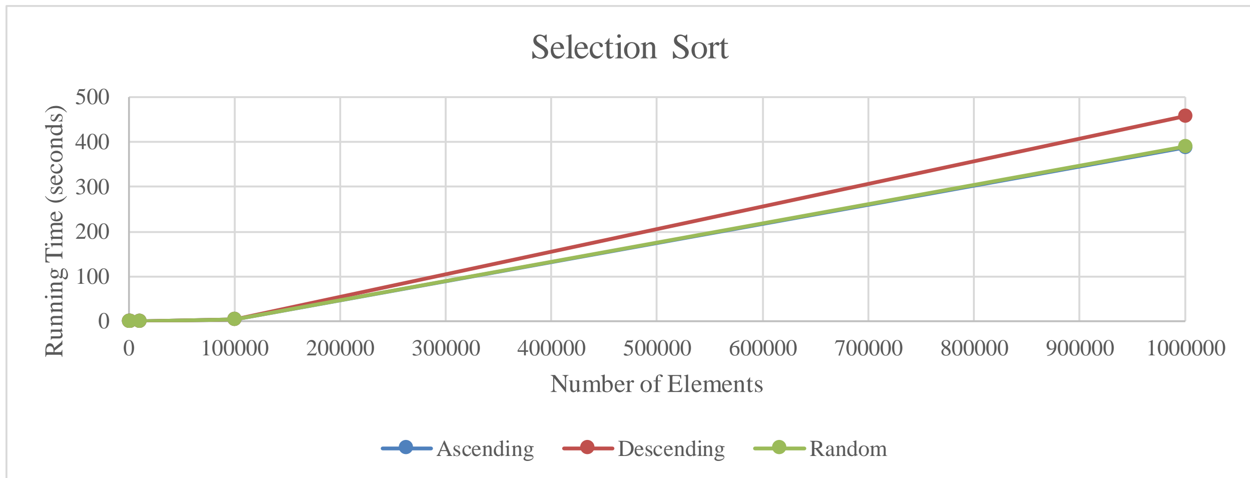

Figure 3 – Running times of Selection Sort

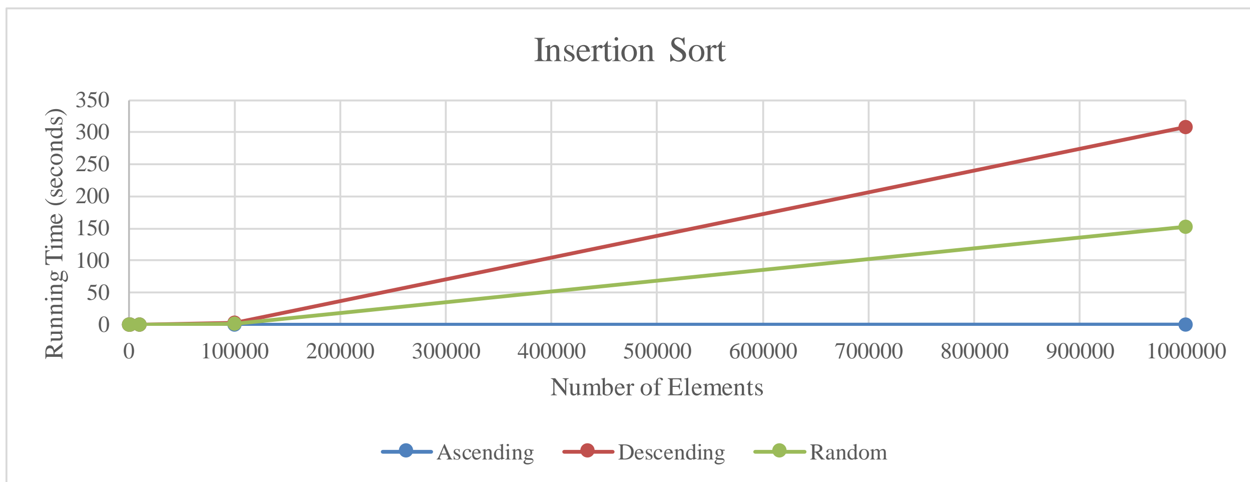

Figure 4 – Running times of Insertion Sort

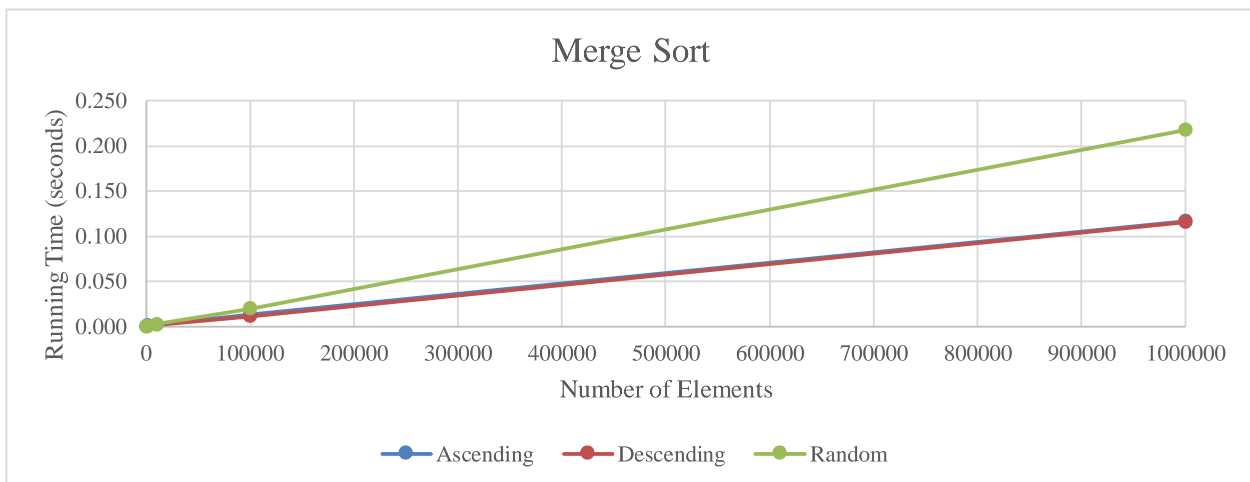

Figure 5 – Running times of Merge Sort

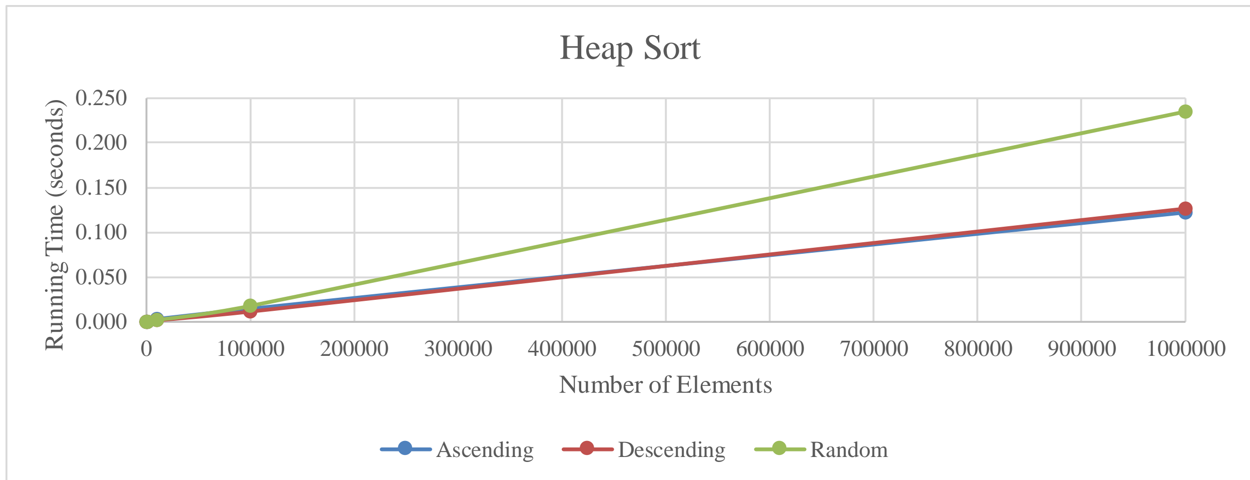

Figure 6 – Running times of Heap Sort

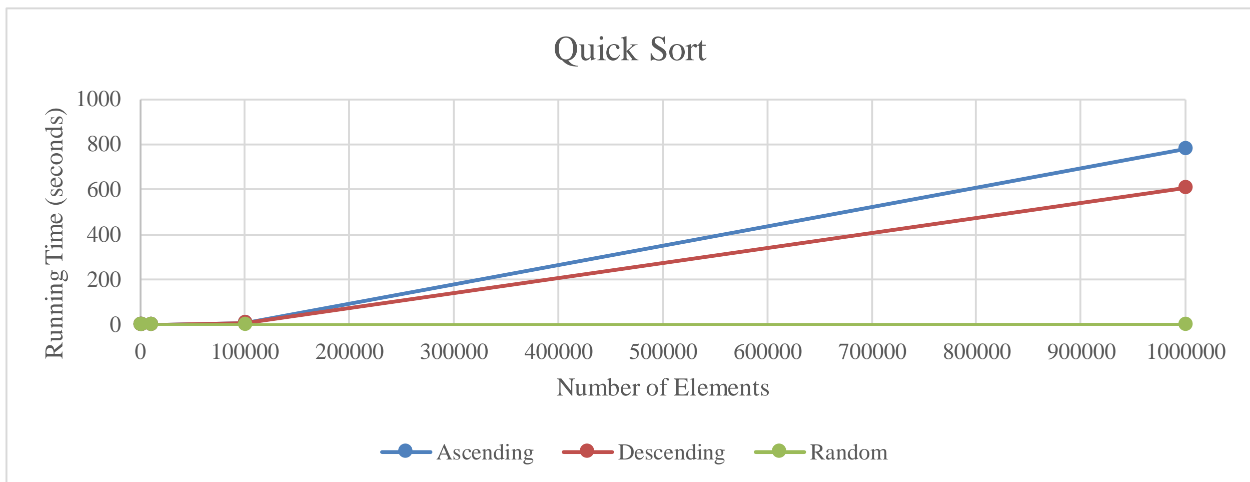

Figure 7 – Running times of Quick Sort

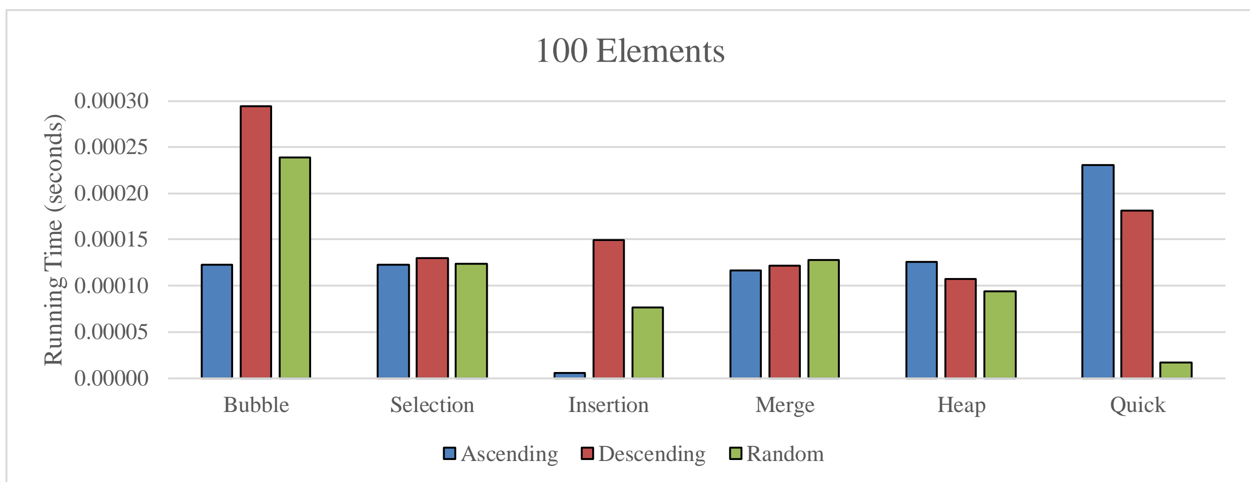

Figure 8 – Running times of all sorts on 100 elements

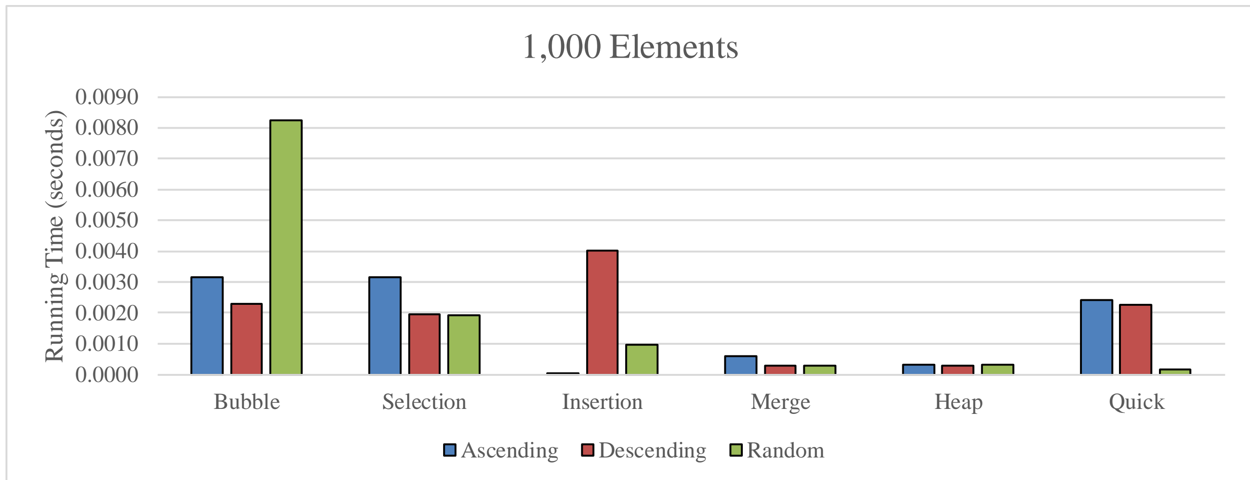

Figure 9 – Running times of all sorts on 1,000 elements

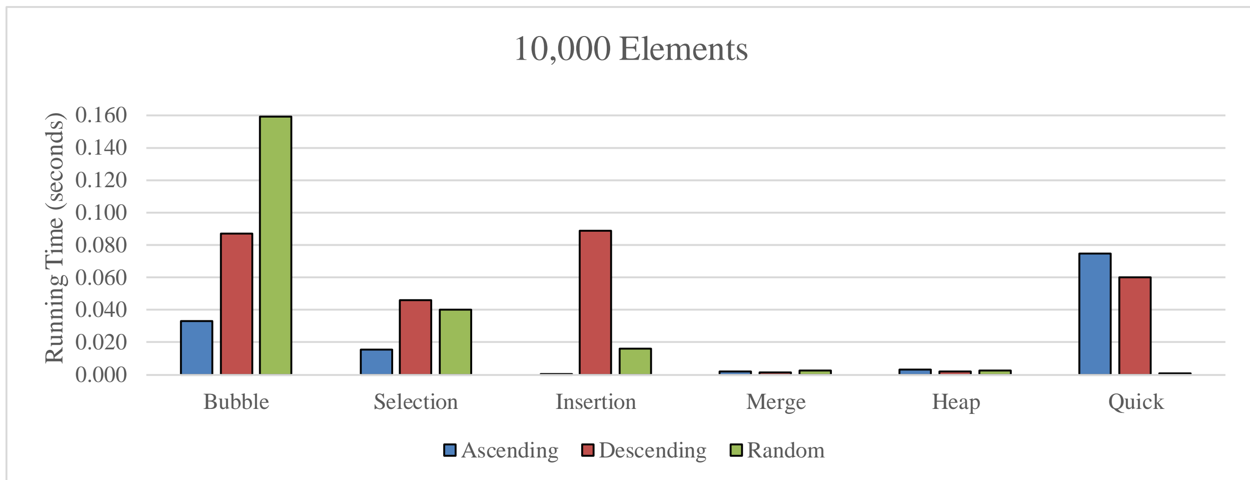

Figure 10 – Running times of all sorts on 10,000 elements

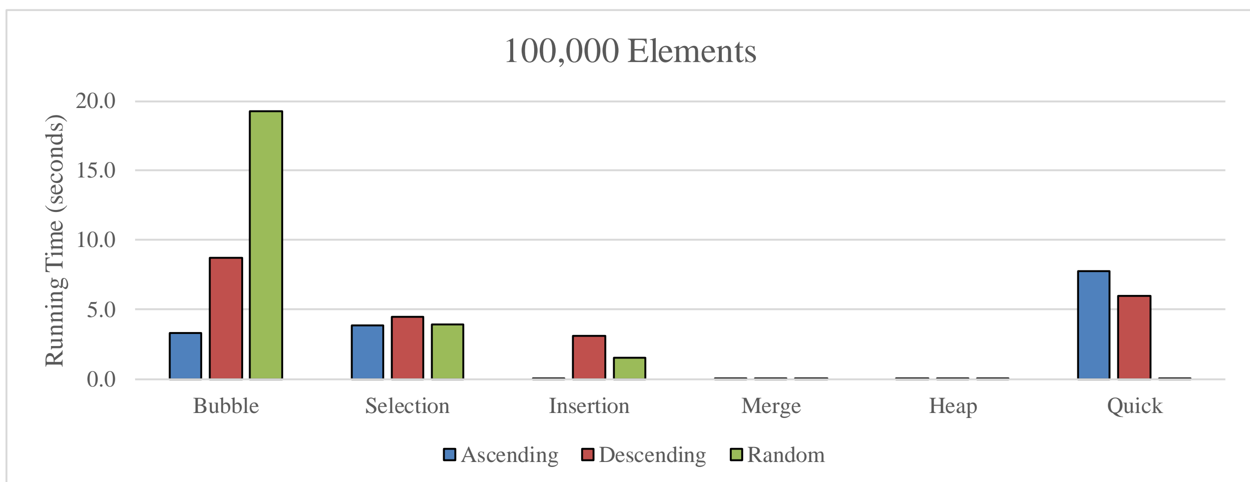

Figure 11 – Running times of all sorts on 100,000 elements

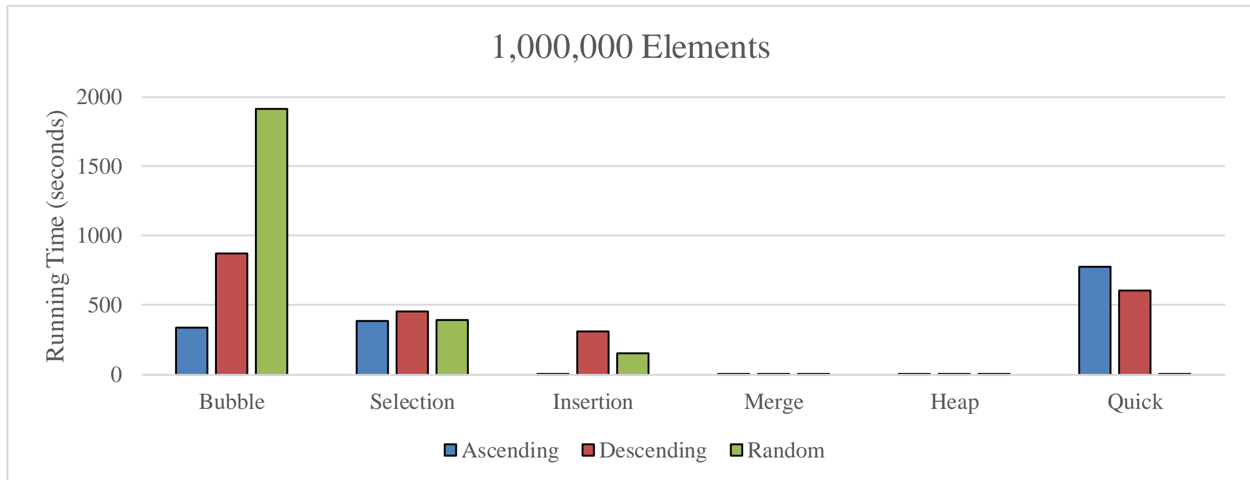

Figure 12 – Running times of all sorts on 1,000,000 elements

Bibliography

1. Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Cambridge (Inglaterra): Mit, 2009. PDF.