Assignment 3: Sorting - WRITEUP

Thadeus Ballmer

October 2021

1 Introduction

In assignment three, four distinct sorting algorithms were implemented: insertion sort, shell sort, heap sort, and quick sort. Each algorithm tackles the problem of sorting data in its own unique way. Although some algorithms are broadly more efficient than others, in specific cases each algorithm has its place.

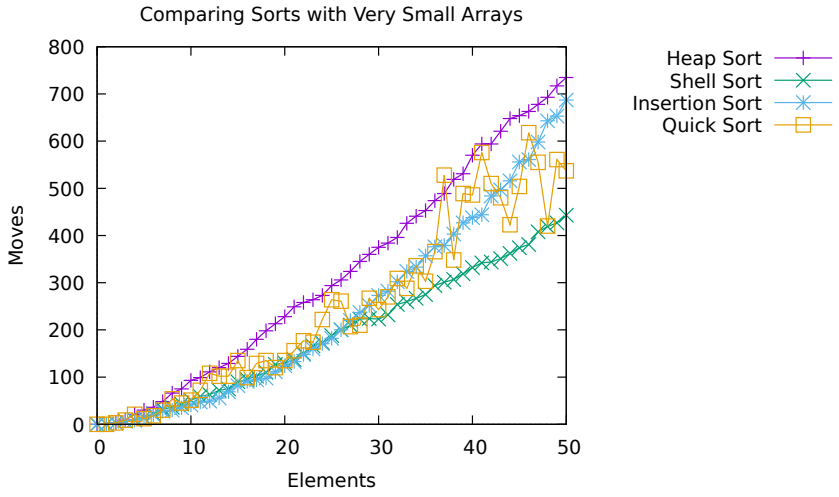
2 Insertion Sort

2.1 Code

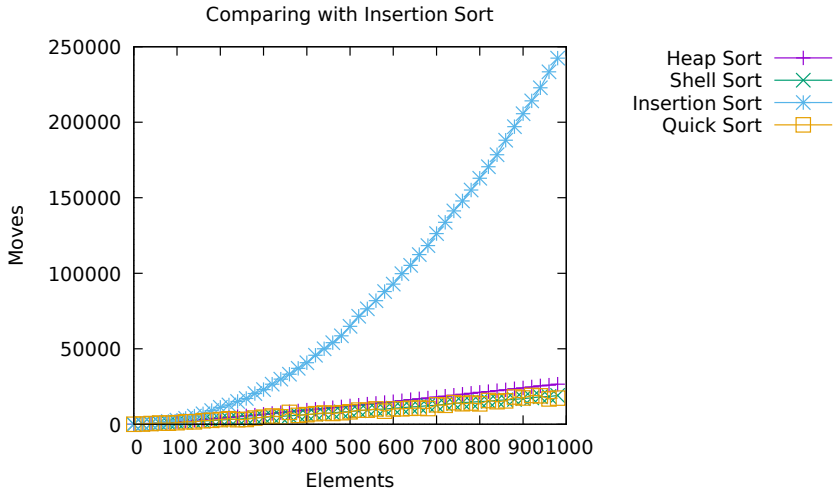
```
#include "insert.h"
#include "stats.h"
#include <inttypes.h>
#include <stdio.h>
#include <unistd.h>

// insertion_sort() is based on Professor Long's Python pseudocode
// found in the assignment 3 pdf.
void insertion_sort(Stats *stats, uint32_t *A, uint32_t n) {
    uint32_t j;
    uint32_t temp;
    for (uint32_t i = 1; i < n; i++) {
        j = i;
        temp = move(stats, A[i]);
        while (j > 0 && cmp(stats, A[j - 1], temp) == 1) {
            A[j] = move(stats, A[j - 1]);
            j--;
        }
        A[j] = move(stats, temp);
    }
}
```

2.2 Insertion Sort Analysis

Insertion sort's simplicity is to its benefit but also to its shortcoming. This algorithm moves along the entire length of the array, comparing each value with all previous descendants until the element reaches the correct placement. This functionality is implemented using a while loop nested in a for loop, classifying this algorithm with a time complexity of $O(n^2)$. Insertion sort requires few moves to sort an array at first—up to around 25 elements. When arrays are set at this length, insertion sort usually requires fewer moves than the other sorts tested[1].



However, after this successful stretch for insertion sort, its exhaustive nature reveals itself with a plummet in efficiency. Beginning with data sets a few hundred elements in length, insertion sort pulls away from the other algorithms in dramatic fashion.



[1]These results are specific to the default seed 13371453. The difference a change in seed can make to sorting outcomes is addressed in section 5.

Although insertion sort itself is only efficient in very small data sets, insertion sort is the base of another popular algorithm: shell sort.

3 Shell Sort

3.1 Code

```
#include "shell.h"
#include "stats.h"
#include <inttypes.h>
#include <math.h>
#include <stdlib.h>
#include <unistd.h>

// shell_sort() is based on Professor Long's Python pseudocode
// found in the assignment 3 pdf.
void shell_sort(Stats *stats, uint32_t *A, uint32_t n) {
    uint32_t range = log(2 * n + 3) / log(3);
    uint32_t gap;
    uint32_t j;
    uint32_t temp;

    do {
        gap = (pow(3, range) - 1) / 2;
        for (uint32_t i = gap; i < n; i++) {
            j = i;
            temp = move(stats, A[i]);
            while (j >= gap && cmp(stats, A[j - gap], temp) == 1) {
                A[j] = move(stats, A[j - gap]);
                j -= gap;
            }
            A[j] = move(stats, temp);
        }
    } while (--range > 0);
}
```

The pseudocode of shell sort provided in the assignment 3 pdf used a generator to compute *gap*. Since generators do not exist in C, I implemented a do while loop instead. My original idea was to compute an array of all the possible *gap* values first, but this solution proved inelegant.

3.2 Shell Sort Analysis

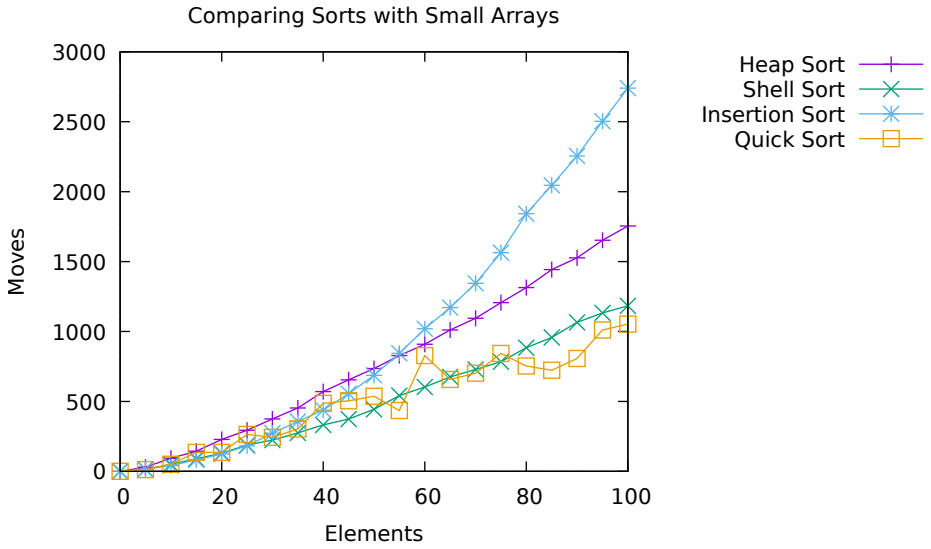
Shell sort functions in a similar way to insertion sort. However, shell sort modifies the distance between compared elements. Instead of comparing the values of indexes neighboring each other, values with a computed *gap* (or distance) between them are compared. The algorithm begins by comparing elements a large gap apart, but then decreases said gap with each sorting iteration. The gap lengths themselves are computed as

$$\left\lfloor \frac{3^k - 1}{2} \right\rfloor$$

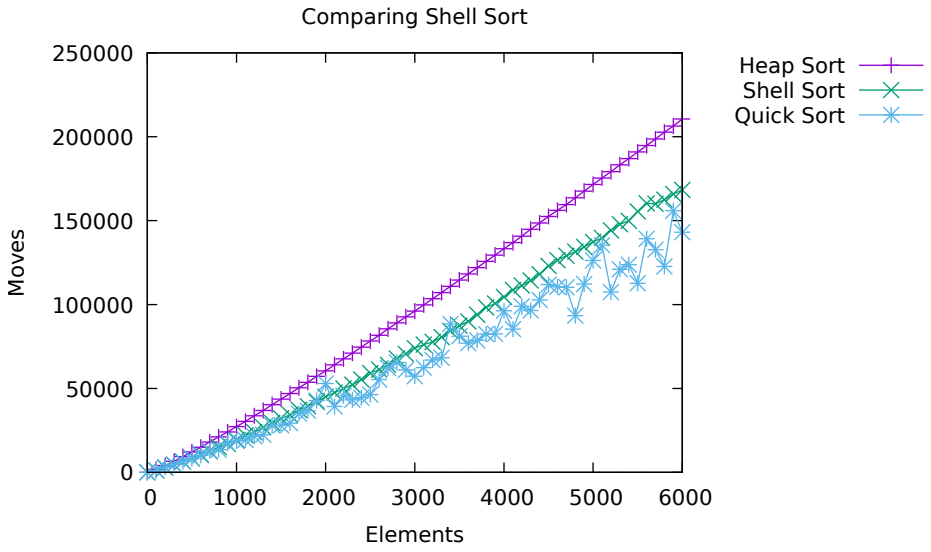
with the largest $k =$

$$\left\lfloor \frac{\log(2n+3)}{\log(3)} \right\rfloor$$

where n is the length of the array. There are many ways to compute this series of gaps, but with Donald Knuth's calculation the time complexity of shell sort is only $O(n^{\frac{3}{2}})$. Just like its cousin insertion sort, shell sort thrives with small data sets. However, shell sort stays efficient much longer than insertion sort. Shell sort is even faster than heap sort by a large degree for the entirety of small data sets.



Although quick sort appears faster than shell sort, quick sort's efficiency is more random than shell sort. Quick sort is also burdened with a worse possible time complexity of $O(n^2)$ (more on this later). In fact, shell sort keeps up with quick sort for a few thousand elements



With these larger data sets, although shell sort lacks slightly behind quick sort, shell sort remains dominant over heap sort. By staying consistent through small to large data sets, shell sort proves itself as a solid choice for an all around sorting algorithm.

4 Heap Sort

4.1 Code

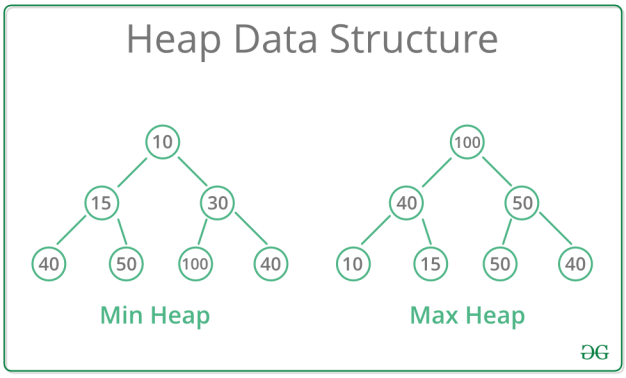
```
#include "heap.h"
#include "stats.h"
#include <inttypes.h>
#include <stdbool.h>
#include <stdio.h>
#include <unistd.h>

// All functions in heap.c are based on Professor Long's pseudocode
// found in assignment 3 pdf.
// max_child() returns the location of the largest child
uint32_t max_child(Stats *stats, uint32_t *A, uint32_t first, uint32_t last) {
    uint32_t left = 2 * first;
    uint32_t right = left + 1;
    if (right <= last && cmp(stats, A[right - 1], A[left - 1]) == 1) {
        return right;
    }
    return left;
}
```

```
// fix_heap() helps place children in the correct position to be sorted
void fix_heap(Stats *stats, uint32_t *A, uint32_t first, uint32_t last) {
    bool found = false;
    uint32_t mother = first;
    uint32_t great = max_child(stats, A, mother, last);

    while (mother <= last / 2 && !found) {
        if (cmp(stats, A[great - 1], A[mother - 1]) == 1) {
            swap(stats, &A[mother - 1], &A[great - 1]);
            mother = great;
            great = max_child(stats, A, mother, last);
        } else {
            found = true;
        }
    }
}

// build_heap constructs the heap to be sorted
void build_heap(Stats *stats, uint32_t *A, uint32_t first, uint32_t last) {
    for (uint32_t father = last / 2; father > first - 1; father--) {
        fix_heap(stats, A, father, last);
    }
}

// heap_sort sorts the heap
void heap_sort(Stats *stats, uint32_t *A, uint32_t n) {
    uint32_t first = 1;
    uint32_t last = n;
    build_heap(stats, A, first, last);
    for (uint32_t leaf = last; leaf > first; leaf--) {
        swap(stats, &A[first - 1], &A[leaf - 1]);
        fix_heap(stats, A, first, leaf - 1);
    }
}
```

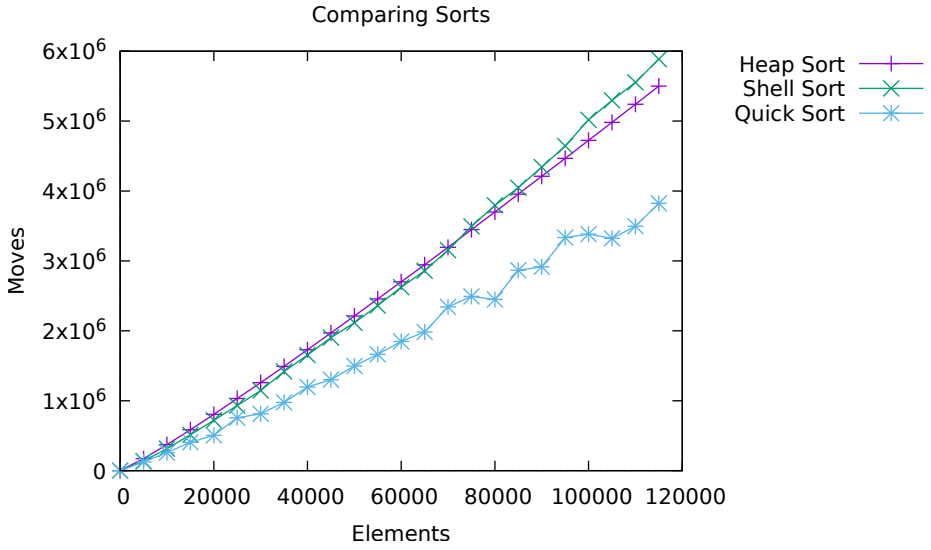
4.2 Heap Sort Analysis

Heap sort is unlike the other algorithms tested in assignment 3. Heap sort does not sort arrays as given, but instead constructs the array data into the heap data structure. The heap data structure can be visualized as a binary tree, where a parent's index k has a left child with index

$2k$ and a right child with index $2k+1$.[2]



There are two main types of heaps: min heaps and max heaps. In min heap, a parent's value is less than or equal to its children. In max heap, a parent's value is greater than or equal to its children. Heap sort constructs data into max heaps before sorting. Heap sort functions in several steps: constructing a heap, fixing said heap, finding a correct position, removing that index from sorting, then fixing said heap, etc. Even with several steps, heap sort's time complexity only reaches $O(nlog(n))$. Although this algorithm requires many steps and much code, the heap data structure manages large data sets exceptionally.



The heap data structure is slower than shell sort for over 70,000 elements, but manages to surpass shell sort in efficiency at around 80,000 elements. Although by this many elements, quick sort remains the most efficient. These results indicate that heap sort is useful when large data set solutions require the heap data structure.

[2]This is constructed using 1-based indexing.

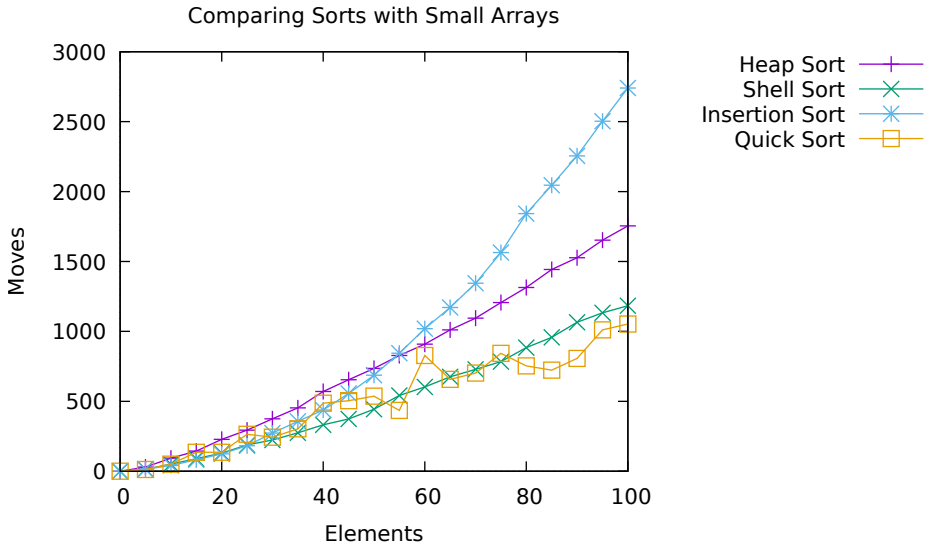
5 Quick Sort

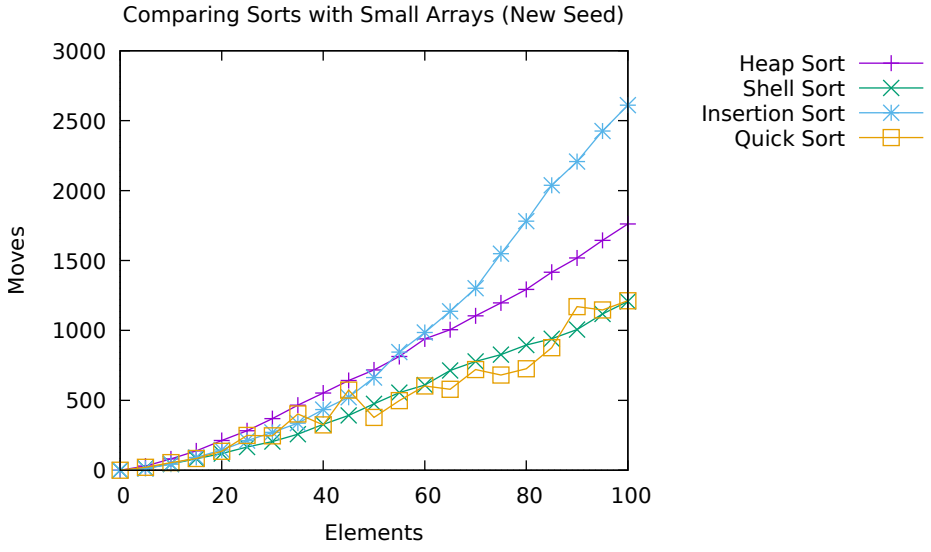
5.1 Code

```c
#include "quick.h"
#include "stats.h"
#include <inttypes.h>
#include <stdio.h>
#include <unistd.h>

// All functions in quick.c are based on Professor Long's python pseudocode
// found in the assignment 3 pdf.

// partition() compares and swaps two values when needed. It returns
// the new partition to be used.
uint32_t partition(Stats *stats, uint32_t *A, uint32_t lo, uint32_t hi) {
    uint32_t i = lo - 1;
    for (uint32_t j = lo; j < hi; j++) {
        if (cmp(stats, A[hi - 1], A[j - 1]) == 1) {
            i++;
            swap(stats, &A[i - 1], &A[j - 1]);
        }
    }
    swap(stats, &A[i], &A[hi - 1]);
    return i + 1;
}

// quick_sorter() implements the recursion functionality of quick_sort().
void quick_sorter(Stats *stats, uint32_t *A, uint32_t lo, uint32_t hi) {
    if (lo < hi) {
        uint32_t p = partition(stats, A, lo, hi);
        quick_sorter(stats, A, lo, p - 1);
        quick_sorter(stats, A, p + 1, hi);
    }
}

// quick_sort() itself is used to connect the helper functions.
void quick_sort(Stats *stats, uint32_t *A, uint32_t n) {
    quick_sorter(stats, A, 1, n);
}
```

5.2 Quick Sort Analysis

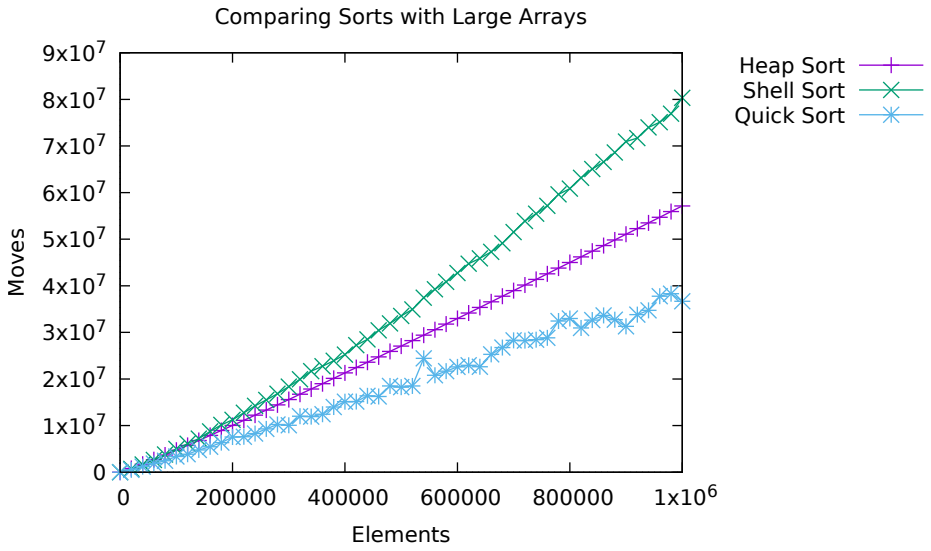
There is a reason quick sort has quick in its name. Quick sort remains one of the fastest and most popular sorts today. Quick sort is a recursive *divide-and-conquer* algorithm. It creates a partition, then swaps values on either side of the partition to align with the partition. Then it repeats this process recursively on both sides of the partition. This simple but powerful sorting algorithm dominates in efficiency from small to large data sets—with an average time complexity of only $O(nlog(n))$. Although quick sort is fast, in all the graphs quick sort appears inconsistent; just compare how much quick sort's results differ with only a change in seed.





Quick sort is so inconsistent that it has a worse possible time complexity of $O(n^2)$. Quick sort's inconsistency stems from the great difference partitions can make. Quick sort's swaps change

in behavior so much due to the differences in partitions—especially the first partition. In other sorts, their swap processes are relatively predictable. Although this partitioning behaviour is unstable, it also gives quick sort great flexibility. Quick sort stays efficient no matter the length of the array.



Despite its inconsistencies, from 0 to 1,000,000 and beyond, this algorithm sorts faster than any other sort in assignment 3.

6 Conclusion

This assignment taught me how the most popular sorting algorithms work. I learned their strengths and weaknesses through comparison and experimentation. This gives me the confidence to make informed decisions on which sort will best optimize specific scenarios. Beyond the algorithms themselves, I also learned about time complexity and Big-O notation. Being able to assign a value to the efficiency of any algorithm is a crucial tool for any computer scientist.