An Empirical Study of Sorting Algorithms using METAL Data

Brenden Talasco
bm17tala@siena.edu

Sean O'Malley
s05omal@siena.edu

April 29, 2024

Abstract

In this study, we analyzed several different sorting algorithms with METAL Data as the input in order to demonstrate the efficiency of the sorting algorithms depending on the kinds of data inputted into them.

1 Introduction

This empirical study focuses on the BubbleSort, MergeSort, and QuickSort sorting algorithms. These sorting algorithms were implemented in the Python programming language. Each sorting algorithm was used with input data from 5 different .tmg files of varying sizes, with 3 trials of each file in order to determine the time and number of constant time operations completed by each algorithm.

2 Computing Environment

For this study, a Python program was developed to test the sorting algorithms. With each .tmg file, edges were ignored. Instead, for each trial of a .tmg file, the trial would first run the three sorting algorithms on the latitude of each vertex so they became sorted in southmost to northmost order, then they would be ran on the longitude in order to sort from westmost to eastmost order.

All runs were completed using the following Python version (this was printed by using "`print("Python", sys.version)`"):

```
Python 3.12.3 (tags/v3.12.3:f6650f9, Apr  9 2024, 14:05:25)
[MSC v.1938 64 bit (AMD64)]
```

The following tests were run on a Dell G7 15 laptop (while charging) under the Windows 10 22H2 x86_64 operating system, with an Intel Core i7-10750H CPU clocked at 2.60GHz, 6 Cores, 384KB of L1 Cache, 1.5MB of L2 Cache, and 12.0 MB of L3 Cache. The system has 16GB of RAM.

3 BubbleSort

The BubbleSort algorithm as shown in Figure 1 was implemented in Python, and was instructed to also count the number of array comparisons and modifications of array indices while sorting.

```
ALGORITHM BubbleSort(A[0..n − 1])
    //Sorts a given array by bubble sort
    //Input: An array A[0..n − 1] of orderable elements
    //Output: Array A[0..n − 1] sorted in nondecreasing order
    for i ← 0 to n − 2 do
        for j ← 0 to n − 2 − i do
            if A[j + 1] < A[j] swap A[j] and A[j + 1]
```

Figure 1: The BubbleSort sorting algorithm, as implemented for this study.

3.1 BubbleSort Expectations

The theoretical expectation is that BubbleSort will have an average and worst case time complexity of O(n^2), and a best case time complexity of O(n). This should then mean that .tmg files with vertices having latitudes/longitudes in a more sorted manner should complete closer to linear time, whereas less sorted latitudes/longitudes should complete in quadratic time.

4 MergeSort

The MergeSort algorithm as shown in Figure 2 was implemented in Python, and was instructed to also count the number of array comparisons and modifications of array indices while sorting.

```
**ALGORITHM** *Mergesort*(A[0..$n - 1$])
    //Sorts array A[0..$n - 1$] by recursive mergesort
    //Input: An array A[0..$n - 1$] of orderable elements
    //Output: Array A[0..$n - 1$] sorted in nondecreasing order
    if $n > 1$
        copy A[0..$n/2 - 1$] to B[0..$n/2 - 1$]
        copy A[$n/2$..$n - 1$] to C[0..$n/2 - 1$]
        Mergesort(B[0..$n/2 - 1$])
        Mergesort(C[0..$n/2 - 1$])
        Merge(B, C, A) //see below

ALGORITHM Merge(B[0..$p - 1$], C[0..$q - 1$], A[0..$p + q - 1$])
    //Merges two sorted arrays into one sorted array
    //Input: Arrays B[0..$p - 1$] and C[0..$q - 1$] both sorted
    //Output: Sorted array A[0..$p + q - 1$] of the elements of B and C
    $i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$
    while $i < p$ and $j < q$ do
        if B[$i$] $\leq$ C[$j$]
            A[$k$] $\leftarrow$ B[$i$]; $i \leftarrow i + 1$
        else A[$k$] $\leftarrow$ C[$j$]; $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    if $i = p$
        copy C[$j$..$q - 1$] to A[$k$..$p + q - 1$]
    else copy B[$i$..$p - 1$] to A[$k$..$p + q - 1$]
```

Figure 2: The MergeSort sorting algorithm, as implemented for this study.

4.1 MergeSort Expectations

The theoretical expectation is that MergeSort will have a best, average and worst case time complexity of O(nlogn). This should then mean that regardless of how sorted the latitude and longitude of the inputted vertices are, the algorithm should complete in roughly the same amount of time with data sets of a similar size.

5 QuickSort

The QuickSort algorithm as shown in Figure 3 was implemented in Python, and was instructed to also count the number of array comparisons and modifications of array indices while sorting. For the partitions within the algorithm, the last index of each partition was used as a pivot value.

```
ALGORITHM Quicksort(A[l..r])
    //Sorts a subarray by quicksort
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //      indices l and r
    //Output: Subarray A[l..r] sorted in nondecreasing order
    if l < r
        s ←Partition(A[l..r]) //s is a split position
        Quicksort(A[l..s − 1])
        Quicksort(A[s + 1..r])

ALGORITHM Partition(A[l..r])
    //Partitions a subarray, using the last element
    //      as a pivot
    //Input: Subarray of array A[0..n − 1], defined by its left and right
    //      indices l and r (l < r)
    //Output: Partition of A[l..r], with the split position returned as
    //      this function's value
    p ← A[r]
    i ← l - 1

    for j ← l to r - 1 do
```

```
        if arr[j] <= p
            i = i + 1
            swap(A[i], A[j])
    swap(A[i+1], A[r])
    return i + 1
```

Figure 3: The QuickSort sorting algorithm, using the last index to pick ‘pivot’ values for partitions, as implemented for this study.

5.1 QuickSort Expectations

The theoretical expectation is that QuickSort will have an average and best case time complexity of O(nlogn), and a worst case time complexity of O(n^2). This should then mean that if the latitude and longitude of the vertices are less sorted, the algorithm should experience a time complexity closer to O(nlogn). If the latitude and longitude of the vertices are more sorted, the partitioning algorithm will have to deal with many more highly unbalanced partitions, increasing the time complexity to O(n^2).

6 Results

For each trial, the program creates a printout showing the sorting algorithm tested, the time in seconds that it took to complete, the comparisons along with modifications that the sorting algorithm completed, along with the vertex extremes that were found with the algorithm, as follows (the output from Trial 1 of NY-region.tmg):

```
---latitudes---

Bubble Sort - time: 15.798707 seconds, comparisons: 22879230,
modifications: 31663804

     Northmost vertex: US11/QC223@USA/CAN : 45.010448
     Southmost vertex: NJ440/NY440@NJ/NY : 40.524801

Merge Sort - time: 0.184506 seconds, comparisons: 66708
modifications: 86518
```

```
     Northmost vertex: US11/QC223@USA/CAN : 45.010448
     Southmost vertex: NJ440/NY440@NJ/NY : 40.524801

Quick Sort - time: 0.105717 seconds, comparisons: 125250
modifications: 125328

     Northmost vertex: US11/QC223@USA/CAN : 45.010448
     Southmost vertex: NJ440/NY440@NJ/NY : 40.524801

---longitudes---

Bubble Sort - time: 14.522126 seconds, comparisons: 22879230,
modifications: 24528024

     Eastmost vertex: NY27@End : -71.858611
     Westmost vertex: I-86/NY17@PA/NY : -79.761944

Merge Sort - time: 0.177037 seconds, comparisons: 69908
modifications: 86518

     Eastmost vertex: NY27@End : -71.858611
     Westmost vertex: I-86/NY17@PA/NY : -79.761944

Quick Sort - time: 0.111741 seconds, comparisons: 140892
modifications: 145851

     Eastmost vertex: NY27@End : -71.858611
     Westmost vertex: I-86/NY17@PA/NY : -79.761944
```

Figure 4: The number of seconds that each sorting algorithm took to complete with each .tmg file tested within a graph.

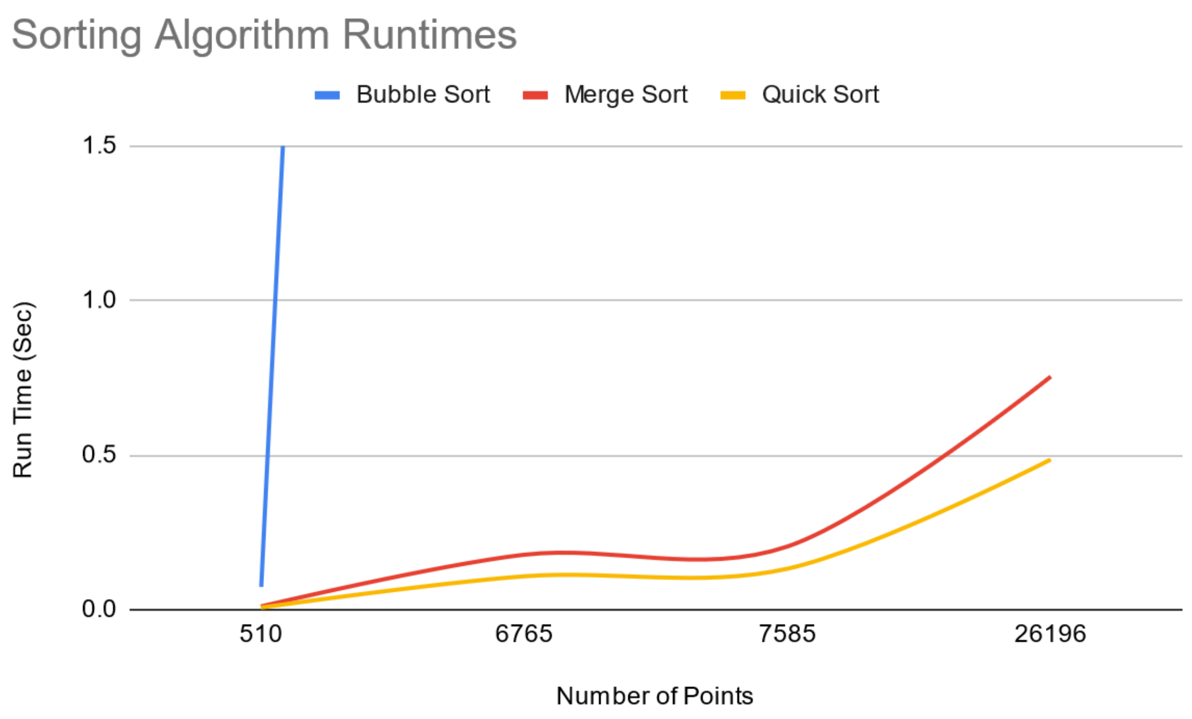


Figure 5: Times in seconds for each algorithm to sort each set of points (Average times for latitudes and longitudes are used).

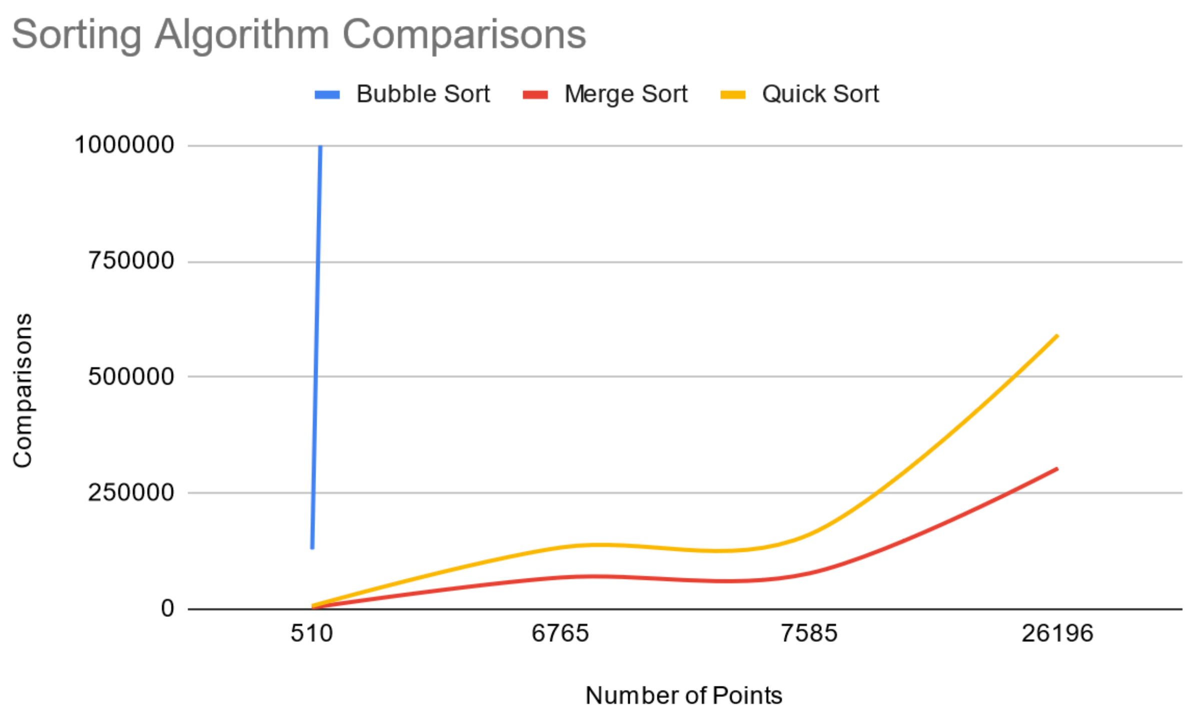


Figure 6: Number of comparisons made for each algorithm to sort each set of points (Average number of comparisons for latitude and longitude are used).

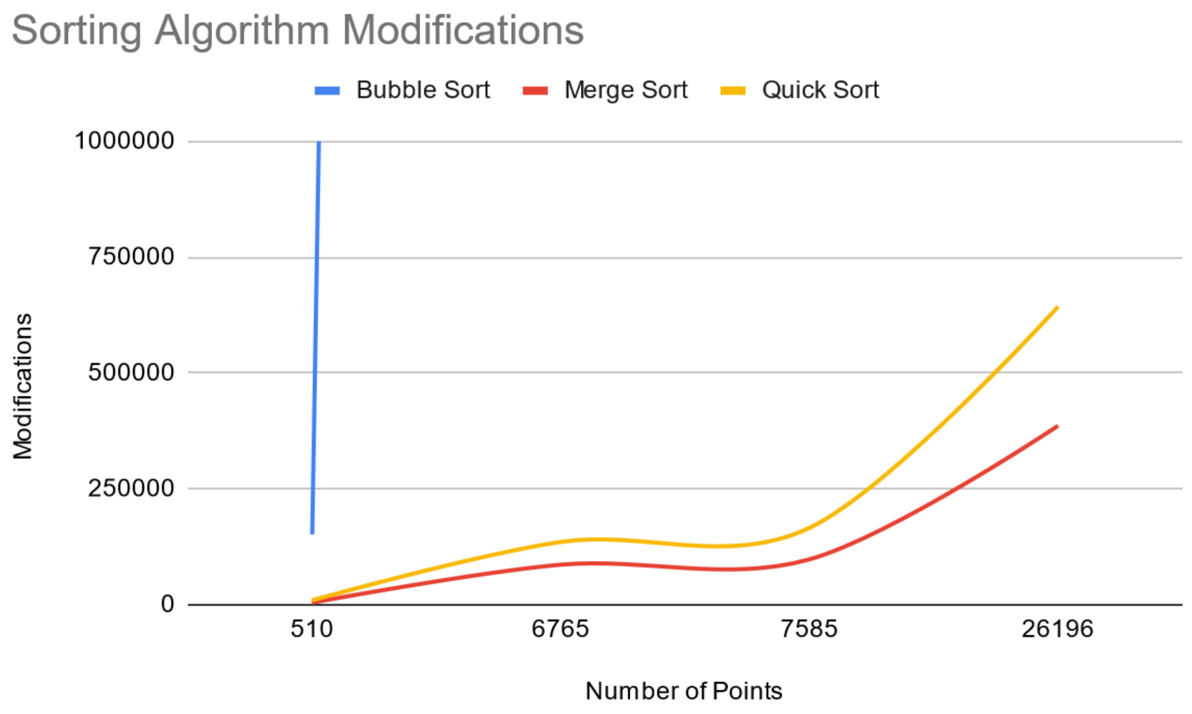
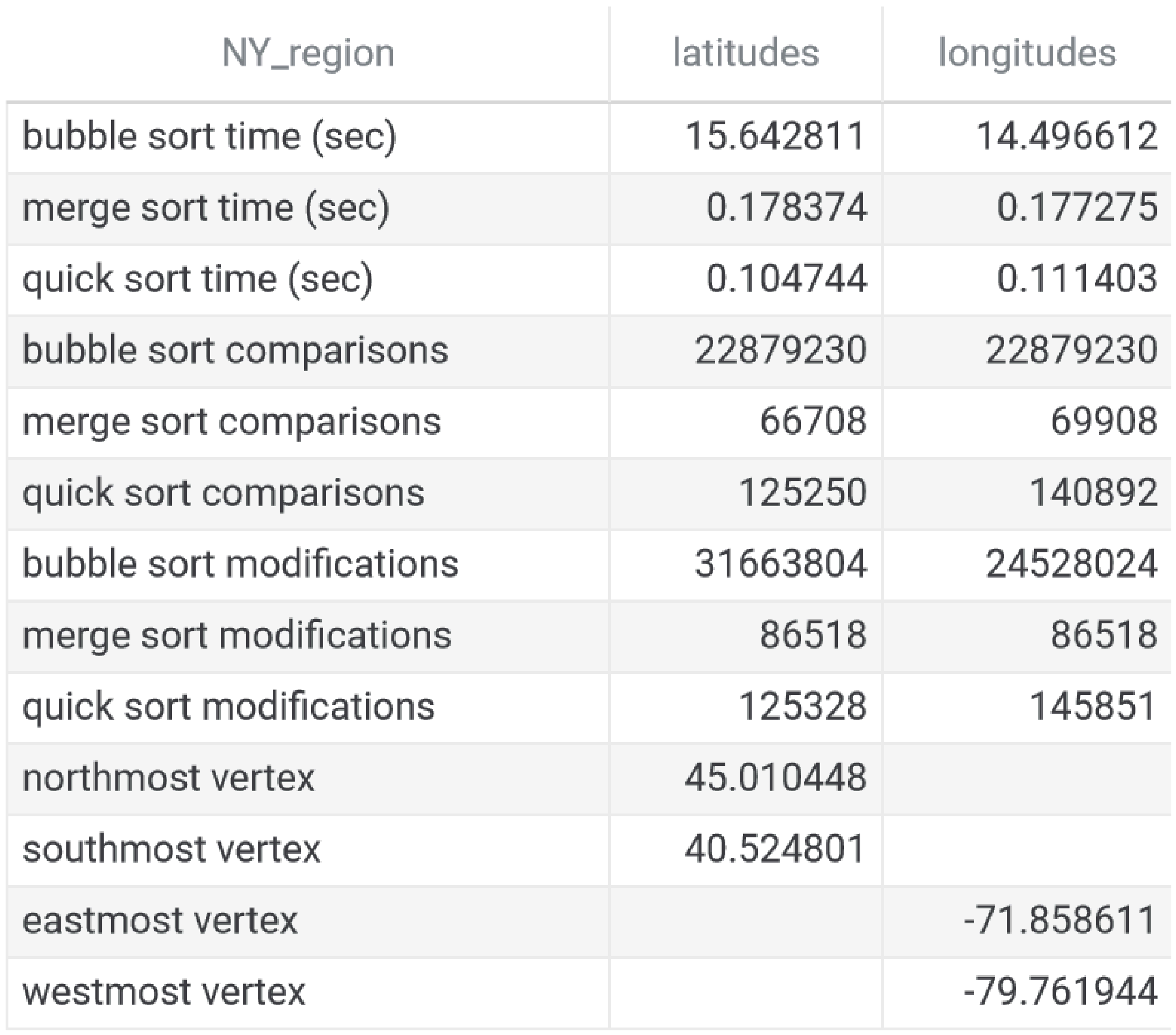
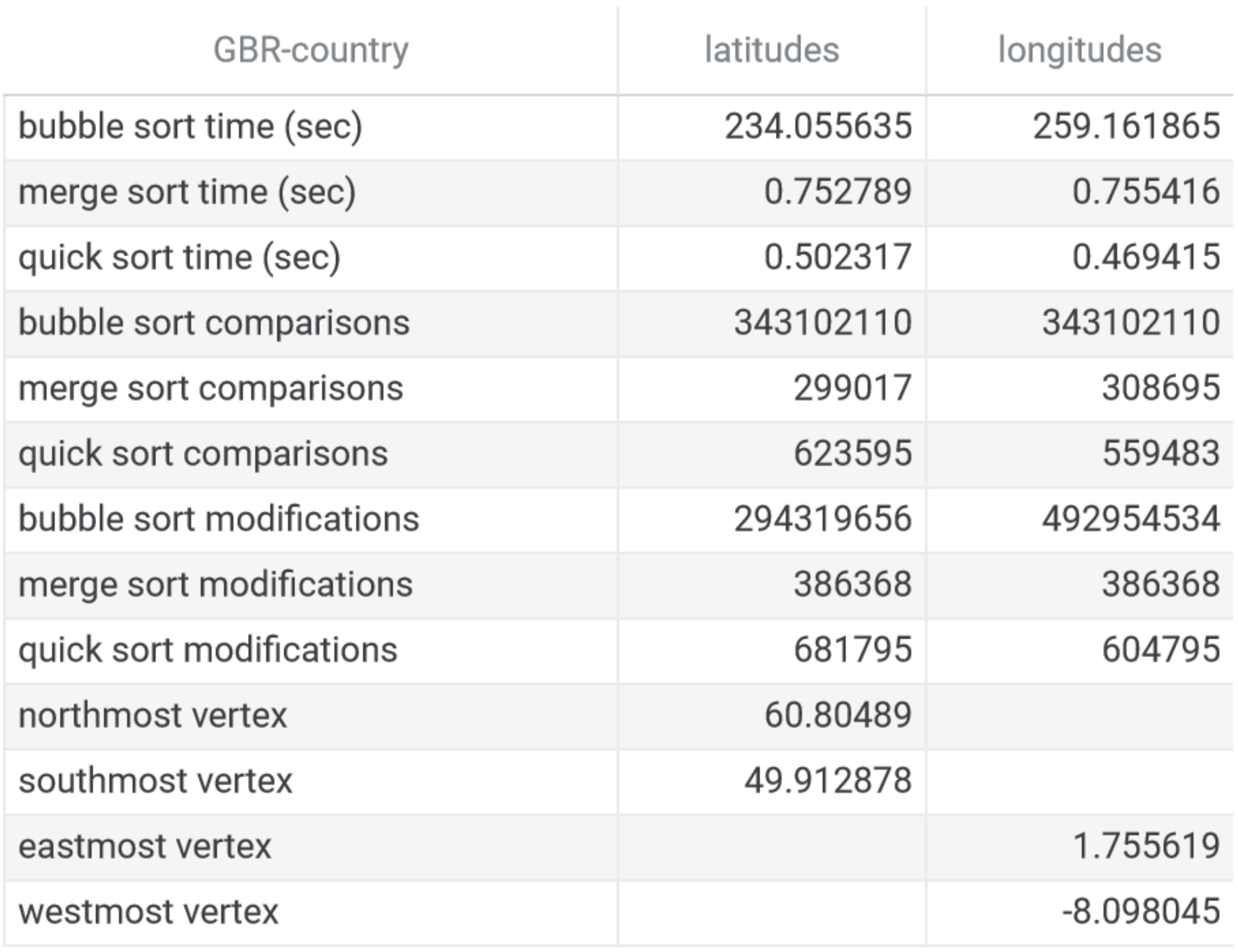


Figure 7: Number of modifications made for each algorithm to sort each set of points (Average number of modifications for latitude and longitude are used).



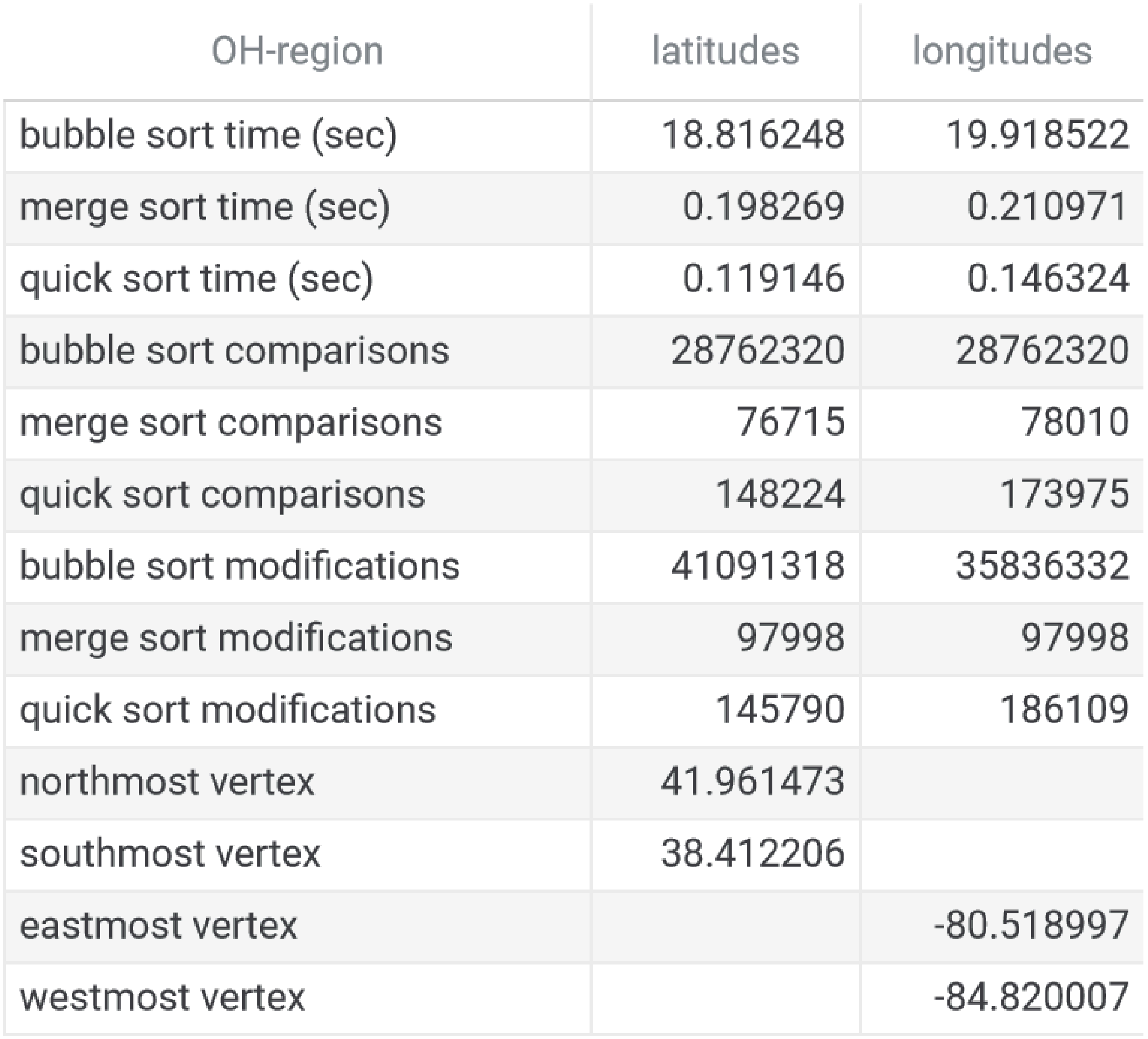
NY_region	latitudes	longitudes
bubble sort time (sec)	15.642811	14.496612
merge sort time (sec)	0.178374	0.177275
quick sort time (sec)	0.104744	0.111403
bubble sort comparisons	22879230	22879230
merge sort comparisons	66708	69908
quick sort comparisons	125250	140892
bubble sort modifications	31663804	24528024
merge sort modifications	86518	86518
quick sort modifications	125328	145851
northmost vertex	45.010448	
southmost vertex	40.524801	
eastmost vertex		-71.858611
westmost vertex		-79.761944

Table 1: Actual times, comparison counts, modification counts, and extreme vertex coordinates for NY_region



GBR-country	latitudes	longitudes
bubble sort time (sec)	234.055635	259.161865
merge sort time (sec)	0.752789	0.755416
quick sort time (sec)	0.502317	0.469415
bubble sort comparisons	343102110	343102110
merge sort comparisons	299017	308695
quick sort comparisons	623595	559483
bubble sort modifications	294319656	492954534
merge sort modifications	386368	386368
quick sort modifications	681795	604795
northmost vertex	60.80489	
southmost vertex	49.912878	
eastmost vertex		1.755619
westmost vertex		-8.098045

Table 2: Actual times, comparison counts, modification counts, and extreme vertex coordinates for GBR-country



OH-region	latitudes	longitudes
bubble sort time (sec)	18.816248	19.918522
merge sort time (sec)	0.198269	0.210971
quick sort time (sec)	0.119146	0.146324
bubble sort comparisons	28762320	28762320
merge sort comparisons	76715	78010
quick sort comparisons	148224	173975
bubble sort modifications	41091318	35836332
merge sort modifications	97998	97998
quick sort modifications	145790	186109
northmost vertex	41.961473	
southmost vertex	38.412206	
eastmost vertex		-80.518997
westmost vertex		-84.820007

Table 3: Actual times, comparison counts, modification counts, and extreme vertex coordinates for OH-region

siena25	latitudes	longitudes
bubble sort time (sec)	0.084131	0.066523
merge sort time (sec)	0.010638	0.010971
quick sort time (sec)	0.00698	0.006982
bubble sort comparisons	129795	129795
merge sort comparisons	3417	3602
quick sort comparisons	7861	6465
bubble sort modifications	217600	87920
merge sort modifications	4588	4588
quick sort modifications	9284	9995
northmost vertex	43.079578	
southmost vertex	42.363079	
eastmost vertex		-73.273154
westmost vertex		-74.235156

Table 4: Actual times, comparison counts, modification counts, and extreme vertex coordinates for siena25

Figures 5, 6, and 7 show the runtimes, comparison counts, and modification counts respectively for each algorithm when run on each set of points (rearranged in ascending order).

6.1 Discussion

The timings and latitude/longitude comparisons match well with the expected behaviors for all 3 algorithms (O(n^2) for bubble sort, and O(nlogn) for merge and quick sort). The graphs in Figures 5-7 show the expected parabolic shape for bubble sort, and the expected linearithmic shape for merge sort and quick sort. The actual numbers in tables 1-4 align well. The number of comparisons and modifications are exactly as predicted. Any variation there would have indicated an error in the algorithm or in the instrumentation that counted the operations. For the timings, small problem sizes are subject to errors due to the accuracy of the nanosecond timer over short timespans. However, when we look at the larger problem sizes, we see exactly what we would expect, when we increase the problem size, the time taken to compute the solution increases exponentially for bubble sort, and linearithmically for merge and quick sort.

7 Conclusions

In this simple study, we found that all of our results matched the expected behavior from the theory. The small variations in run time can perhaps be explained by how soon the algorithm sorts the vertices. The faster the algorithm sorts the vertices, the less often the body of the if/while statements in any of the algorithms need to execute. There is also the issue that the computer used for this study had several other processes such as terminal windows open. This is partially accounted for by taking the average times across 3 different runs of each set of data.