Jackson Campbell

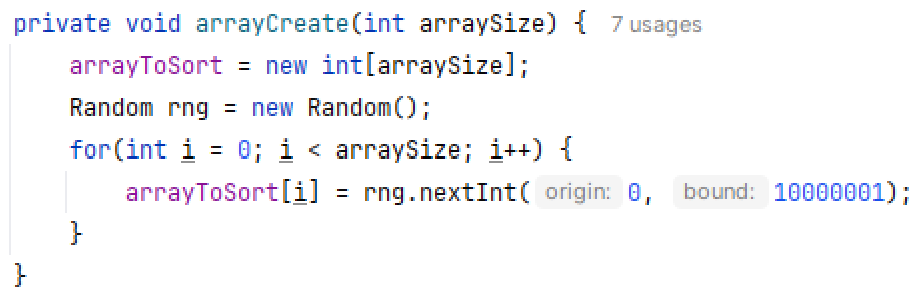
DSA 1 – Byron Hoy

Sorting Algorithms Results and Report

December 9th, 2024

Section 1: Array Creation

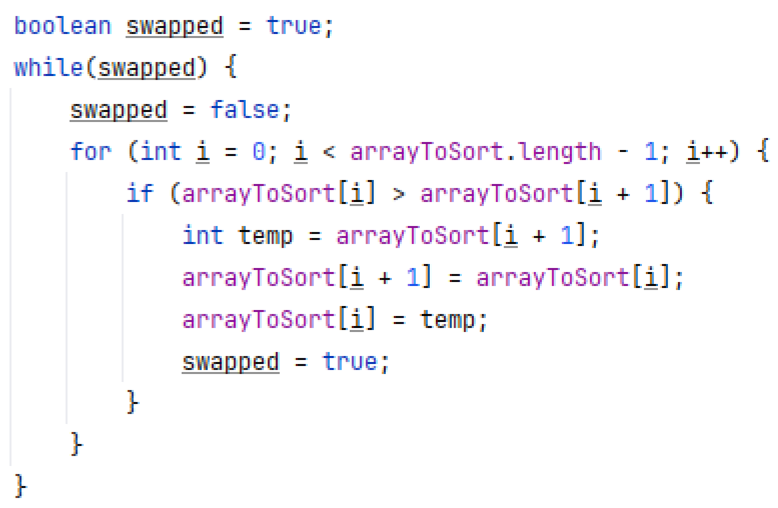
To start this project, I first needed to program a method to create an array based on a given size. While I do not need to necessarily create an array this way, I used a method to allow for complete automation when running the entire sorting simulation. The method is simple, as it only takes the already created “arrayToSort” integer array, and simply generates a random number from 0 to 10,000,000. These numbers generate to each position in the array, until the “arraySize” parameter has been reached in the for loop parameters. This helper method is used for every single sorting method, and is a key component to automating the entire process of data collection and sorting. Below is an attached photo of the method:



```
private void arrayCreate(int arraySize) {  7 usages
    arrayToSort = new int[arraySize];
    Random rng = new Random();
    for(int i = 0; i < arraySize; i++) {
        arrayToSort[i] = rng.nextInt( origin: 0,  bound: 10000001);
    }
}
```

Section 2: Bubble Sort

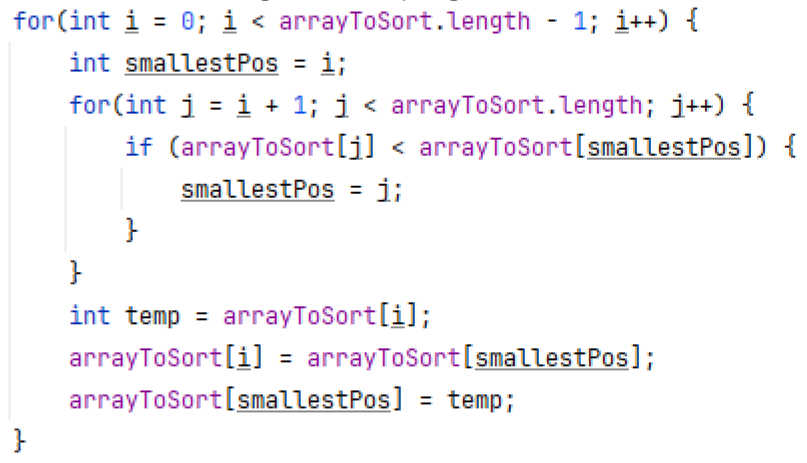
Bubble sort is the simplest of all the sorting algorithms, as it only looks forward to the next value in the loop, and if the forward value is less than the current value, they swap places. This algorithm keeps looping until it no longer detects a swap while it runs, meaning that the sorting is then completed. This, however, means that bubble sort is the slowest of all the sorting algorithms, as it must loop through every single value each time the loop reiterates. Once the data was collected, the total time to sort values from ten thousand to two hundred thousand was slightly over 6 minutes of time. Nothing else had come close to that amount of time, as listed in the excel spreadsheet captured below the report. Attached below is a photo of the sorting method itself (excluding timing mechanisms):



```
boolean swapped = true;
while(swapped) {
    swapped = false;
    for (int i = 0; i < arrayToSort.length - 1; i++) {
        if (arrayToSort[i] > arrayToSort[i + 1]) {
            int temp = arrayToSort[i + 1];
            arrayToSort[i + 1] = arrayToSort[i];
            arrayToSort[i] = temp;
            swapped = true;
        }
    }
}
```

Section 3: Selection Sort

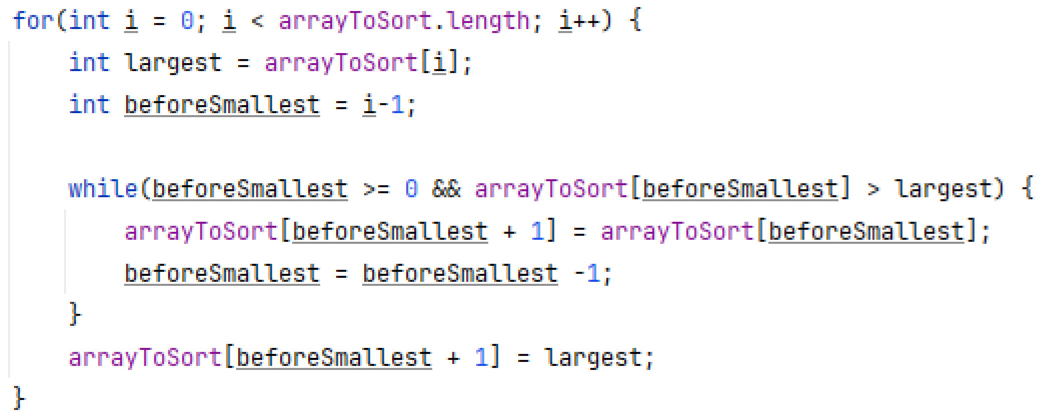
Selection sort is the second sorting algorithm in our testing required, and it's the overall 3rd slowest in my findings. Compared to insertion sort, however, selection is slower at smaller value totals, but works slightly better than insertion at larger value totals to sort. Selection sort works by searching through the rest of the array aside from your current position to find the next smallest value compared to your current value. Once found, the smallest value position is set and then a temporary value is created to swap the current value with the smallest value position's value, ensuring a proper sort to the value. Below is an attached photo of how the algorithm is programmed:



```
for(int i = 0; i < arrayToSort.length - 1; i++) {
    int smallestPos = i;
    for(int j = i + 1; j < arrayToSort.length; j++) {
        if (arrayToSort[j] < arrayToSort[smallestPos]) {
            smallestPos = j;
        }
    }
    int temp = arrayToSort[i];
    arrayToSort[i] = arrayToSort[smallestPos];
    arrayToSort[smallestPos] = temp;
}
```

Section 4: Insertion Sort

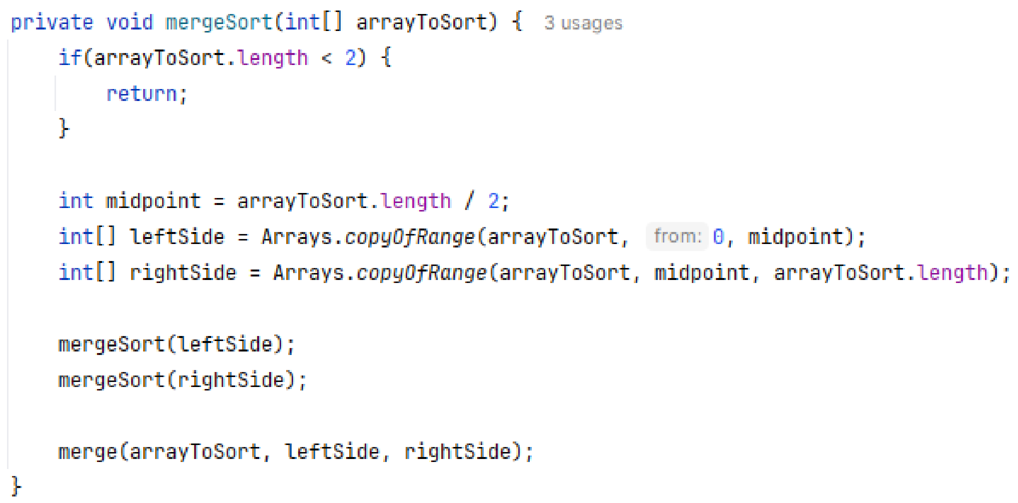
Insertion sort is the third sorting algorithm in our testing required, and is the 2nd overall slowest sorting algorithm in results. It works by iteratively going through the array, and then inserting the element required at the current position to sort the array into the sorted portion of the list. This method works well for partially sorted arrays, and for smaller value sets as compared to selection sort, but works slower at larger value sets than selection does. This sorting method is the last of the seven to work properly with the timing mechanism, as the other methods ran abnormally fast within my program. Below is an attached photo of the programming for insertion sort:



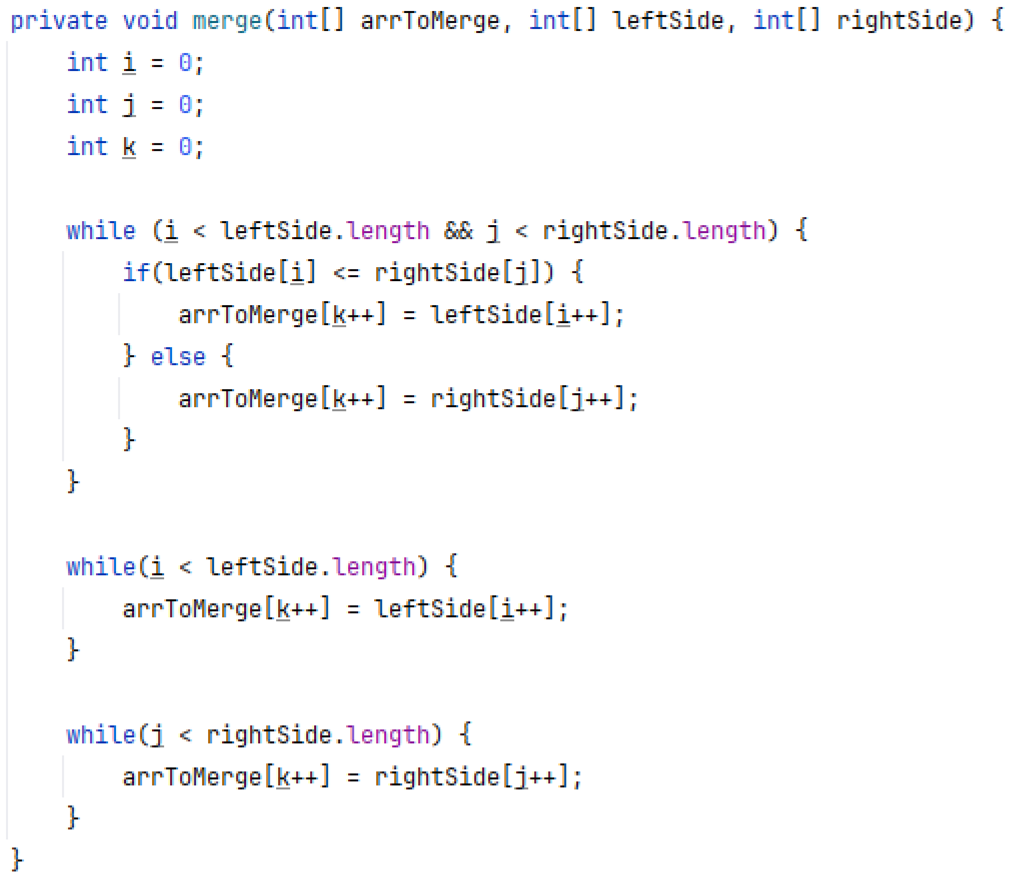
```
for(int i = 0; i < arrayToSort.length; i++) {
    int largest = arrayToSort[i];
    int beforeSmallest = i-1;

    while(beforeSmallest >= 0 && arrayToSort[beforeSmallest] > largest) {
        arrayToSort[beforeSmallest + 1] = arrayToSort[beforeSmallest];
        beforeSmallest = beforeSmallest -1;
    }
    arrayToSort[beforeSmallest + 1] = largest;
}
```

Section 5: Merge Sort

Merge sort is the fourth required sorting algorithm in this project, and is one of the faster running algorithms compared to others (when working properly). However, in my findings for merge, shell, quick and heap sort, my timing method seemed to appear broken as the photos at the end of the report will show. Merge sort works by implementing a divide-and-conquer style of sorting, as it splits the array into smaller sections, and merges the smaller sections back to sorted larger sections. This method, along with others, works by using recursion, and recursively iterates to separate the array into two halves over and over, and then sorting and merging the sub-arrays back into one. Below is an attached photo of the merge sort, along with the “merge” helper method:



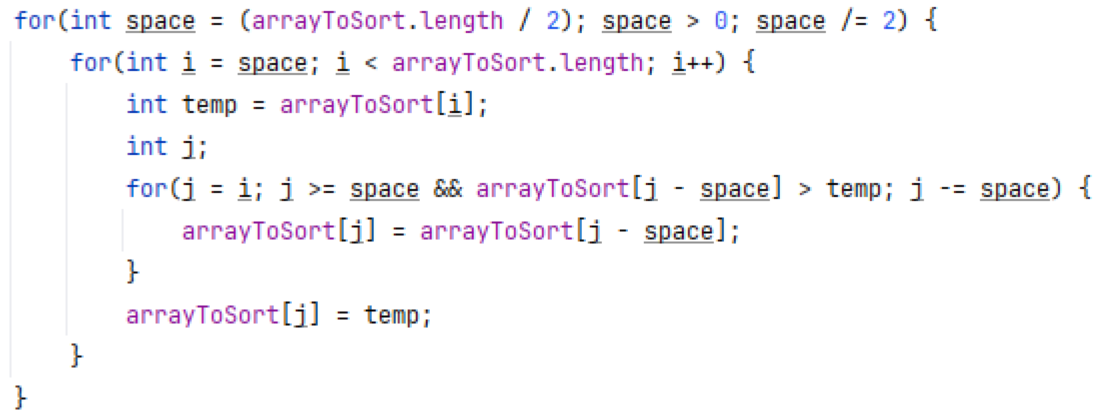
```
private void mergeSort(int[] arrayToSort) {  3 usages
    if(arrayToSort.length < 2) {
        return;
    }

    int midpoint = arrayToSort.length / 2;
    int[] leftSide = Arrays.copyOfRange(arrayToSort,  from: 0, midpoint);
    int[] rightSide = Arrays.copyOfRange(arrayToSort, midpoint, arrayToSort.length);

    mergeSort(leftSide);
    mergeSort(rightSide);

    merge(arrayToSort, leftSide, rightSide);
}
```



```
private void merge(int[] arrToMerge, int[] leftSide, int[] rightSide) {
    int i = 0;
    int j = 0;
    int k = 0;

    while (i < leftSide.length && j < rightSide.length) {
        if(leftSide[i] <= rightSide[j]) {
            arrToMerge[k++] = leftSide[i++];
        } else {
            arrToMerge[k++] = rightSide[j++];
        }
    }

    while(i < leftSide.length) {
        arrToMerge[k++] = leftSide[i++];
    }

    while(j < rightSide.length) {
        arrToMerge[k++] = rightSide[j++];
    }
}
```

Section 6: Shell Sort

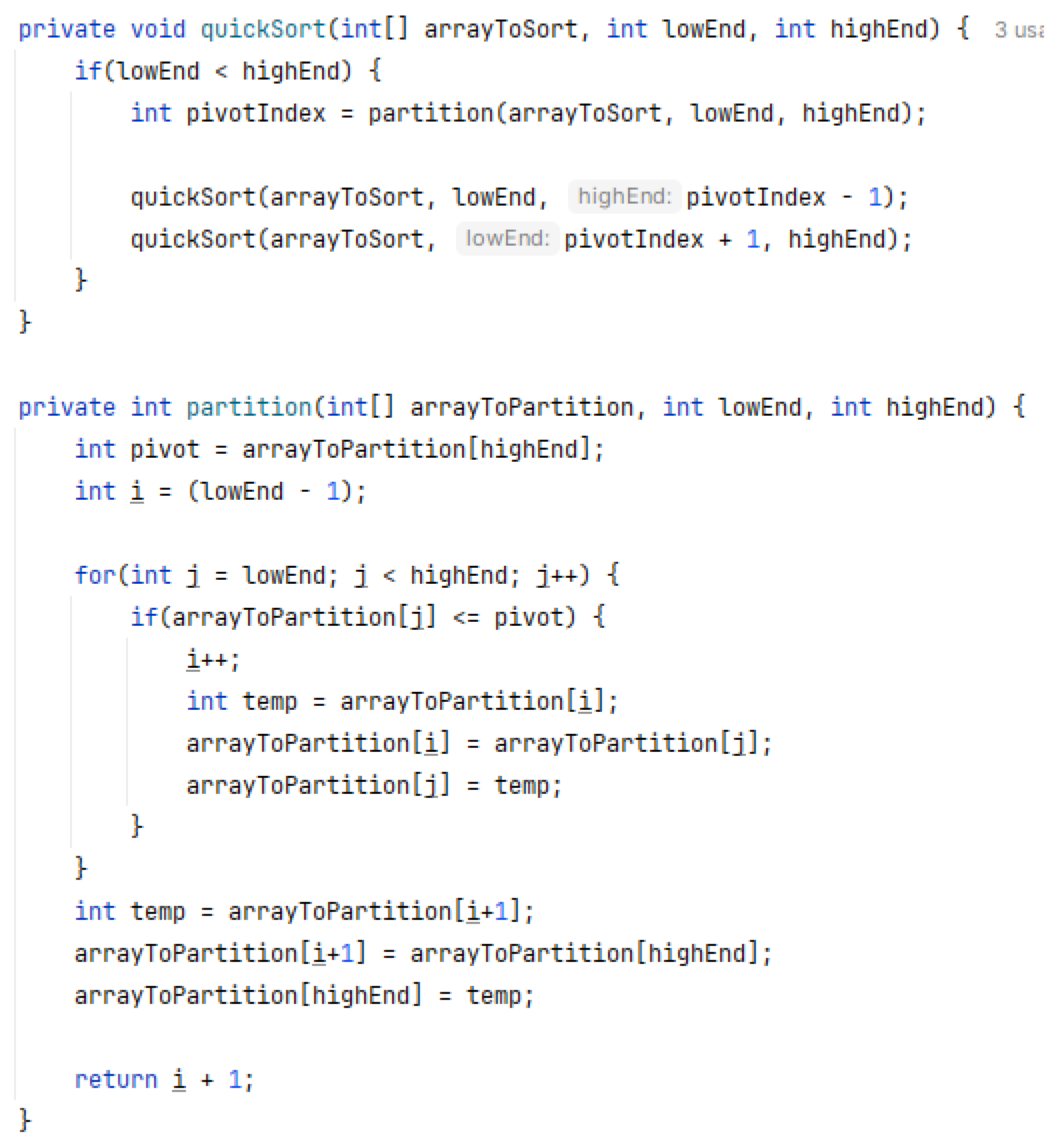
Shell sort is our 5th sorting algorithm required in this project, and works as a variation on insertion sort. Shell works by exchanging further items than insertion does, without making as many movements as insertion needs to make. This sorting algorithm was one I tested far more than others, and even forced me to make a separate method for validating a sorted array. I had thought that my timing was off with shell sort, and as such had switched to an even smaller increment of timing (nanoTime) instead of the typical currentTimeMillis that java can give me. This is then in turn why this data set is the only one in decimals, as the others are already in milliseconds. The algorithm does actually sort, but as to why timing works, I am entirely unsure as this occurs with other methods too, not just this one. Below is an attached photo of the algorithm's programming:



```java
for(int space = (arrayToSort.length / 2); space > 0; space /= 2) {
    for(int i = space; i < arrayToSort.length; i++) {
        int temp = arrayToSort[i];
        int j;
        for(j = i; j >= space && arrayToSort[j - space] > temp; j -= space) {
            arrayToSort[j] = arrayToSort[j - space];
        }
        arrayToSort[j] = temp;
    }
}
```

Section 7: Quick Sort

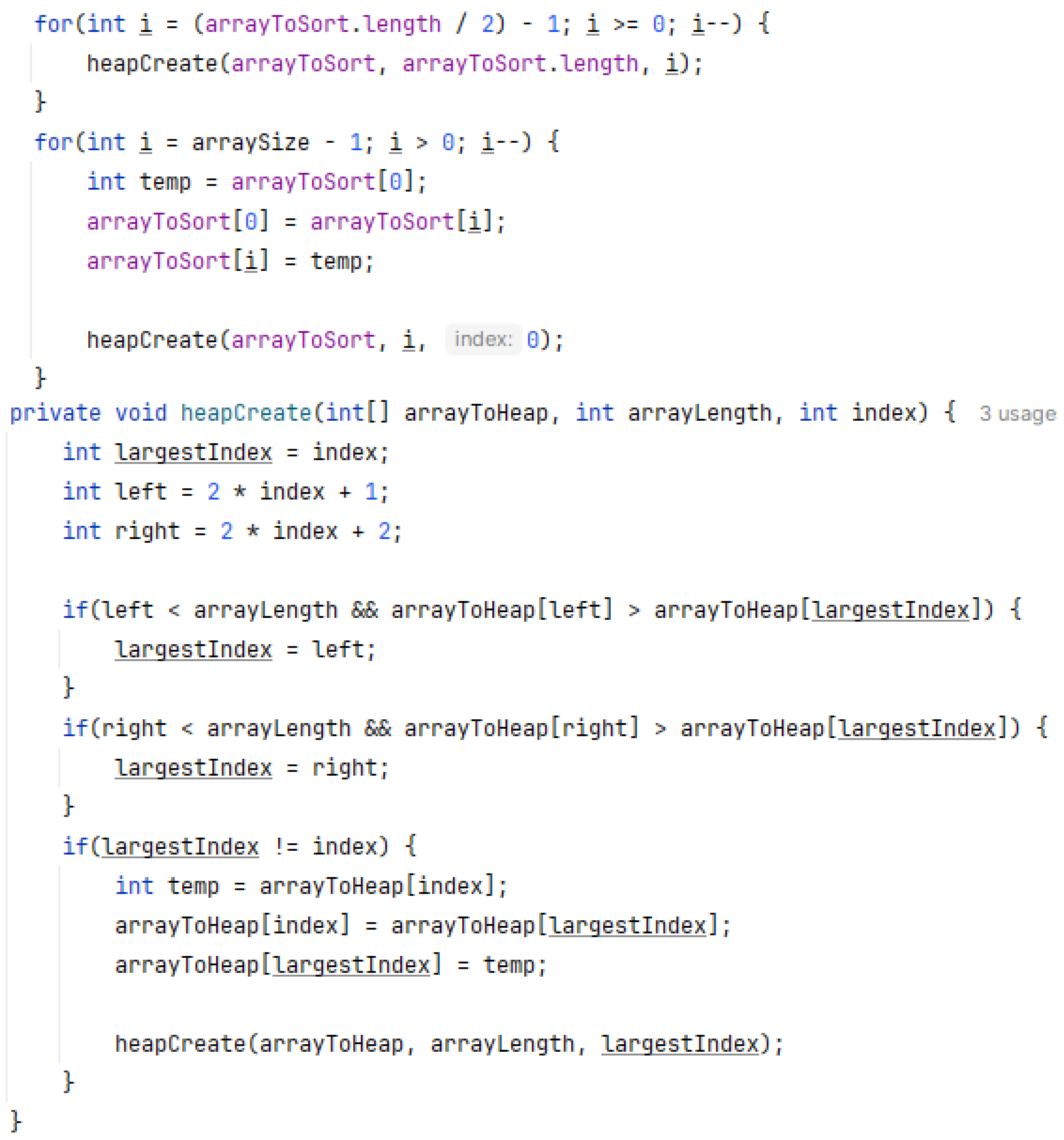
Quick sort, our second to last sorting method within the algorithms, works by sorting on a pivot. This algorithm is one of the fastest sorting algorithms in general, but works abnormally fast within my own programming. More specifically as to how quick sort works on a pivot, it creates partitions from the array that work around the pivot to place the pivot itself into the correct position in the array, ensuring that it's sorted. This method was the fastest out of all of my methods, but fast also meant abnormally fast, sorting from 10-200 thousand in a mere tenth of a second total. Below is an attachment of both the partitioner and the quick sort itself:



```
private void quickSort(int[] arrayToSort, int lowEnd, int highEnd) {  3 usa
    if(lowEnd < highEnd) {
        int pivotIndex = partition(arrayToSort, lowEnd, highEnd);

        quickSort(arrayToSort, lowEnd,  highEnd: pivotIndex - 1);
        quickSort(arrayToSort,  lowEnd: pivotIndex + 1, highEnd);
    }
}

private int partition(int[] arrayToPartition, int lowEnd, int highEnd) {
    int pivot = arrayToPartition[highEnd];
    int i = (lowEnd - 1);

    for(int j = lowEnd; j < highEnd; j++) {
        if(arrayToPartition[j] <= pivot) {
            i++;
            int temp = arrayToPartition[i];
            arrayToPartition[i] = arrayToPartition[j];
            arrayToPartition[j] = temp;
        }
    }
    int temp = arrayToPartition[i+1];
    arrayToPartition[i+1] = arrayToPartition[highEnd];
    arrayToPartition[highEnd] = temp;

    return i + 1;
}
```

Section 8: Heap Sort

Heap sort is the final sorting algorithm needed to complete the entire automation of testing each sorting method in this project. Heap sort works based on a binary heap structure, and is optimized over selection sort similar to how shell is optimized over insertion. Binary heaps are used to quickly find the max element, and use it to sort the array properly. This method also worked abnormally fast, just like the past three algorithms too. Below is an attached photo for the method to create a heap, and the heap sort algorithm itself:



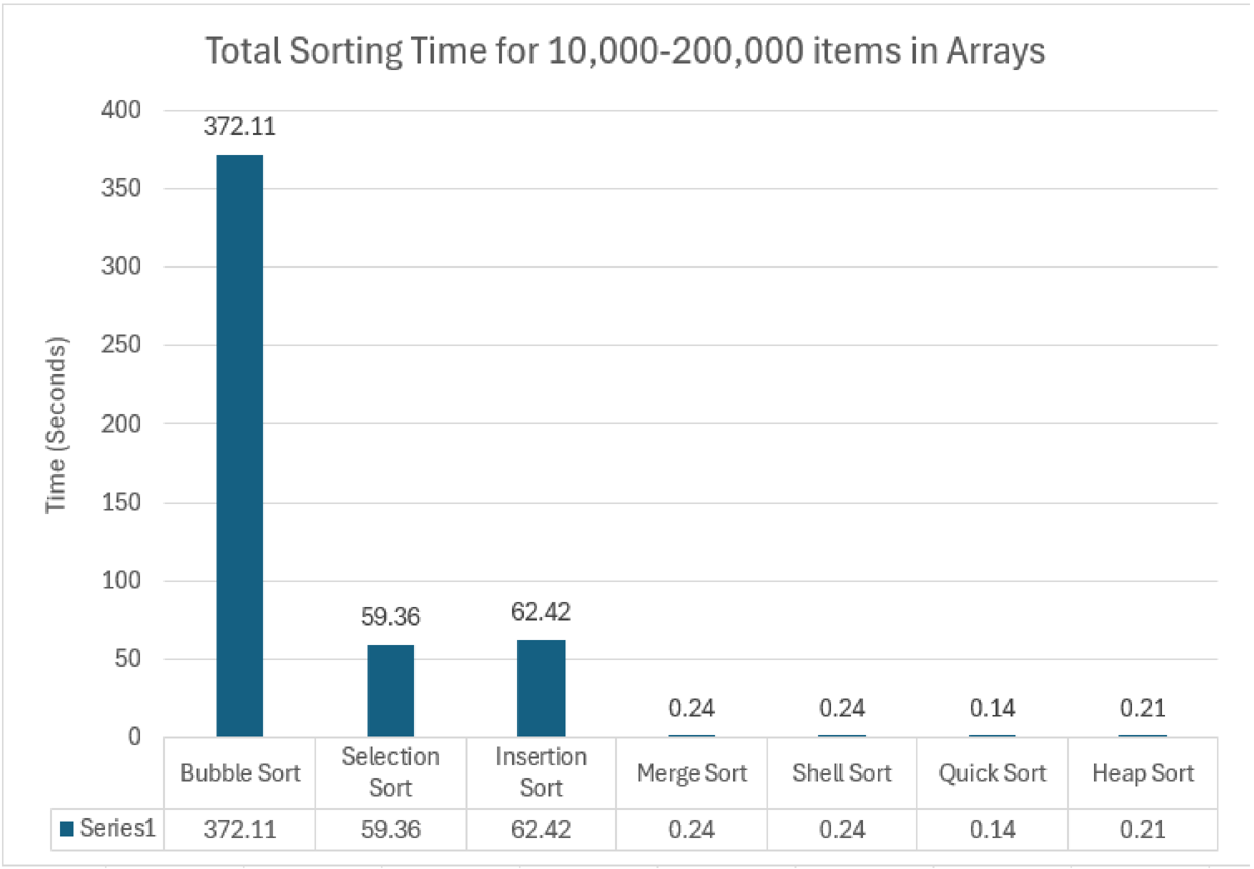
```
    for(int i = (arrayToSort.length / 2) - 1; i >= 0; i--) {
        heapCreate(arrayToSort, arrayToSort.length, i);
    }
    for(int i = arraySize - 1; i > 0; i--) {
        int temp = arrayToSort[0];
        arrayToSort[0] = arrayToSort[i];
        arrayToSort[i] = temp;

        heapCreate(arrayToSort, i,  index: 0);
    }
private void heapCreate(int[] arrayToHeap, int arrayLength, int index) {  3 usage
    int largestIndex = index;
    int left = 2 * index + 1;
    int right = 2 * index + 2;

    if(left < arrayLength && arrayToHeap[left] > arrayToHeap[largestIndex]) {
        largestIndex = left;
    }
    if(right < arrayLength && arrayToHeap[right] > arrayToHeap[largestIndex]) {
        largestIndex = right;
    }
    if(largestIndex != index) {
        int temp = arrayToHeap[index];
        arrayToHeap[index] = arrayToHeap[largestIndex];
        arrayToHeap[largestIndex] = temp;

        heapCreate(arrayToHeap, arrayLength, largestIndex);
    }
}
```

Section 9: Excel Data

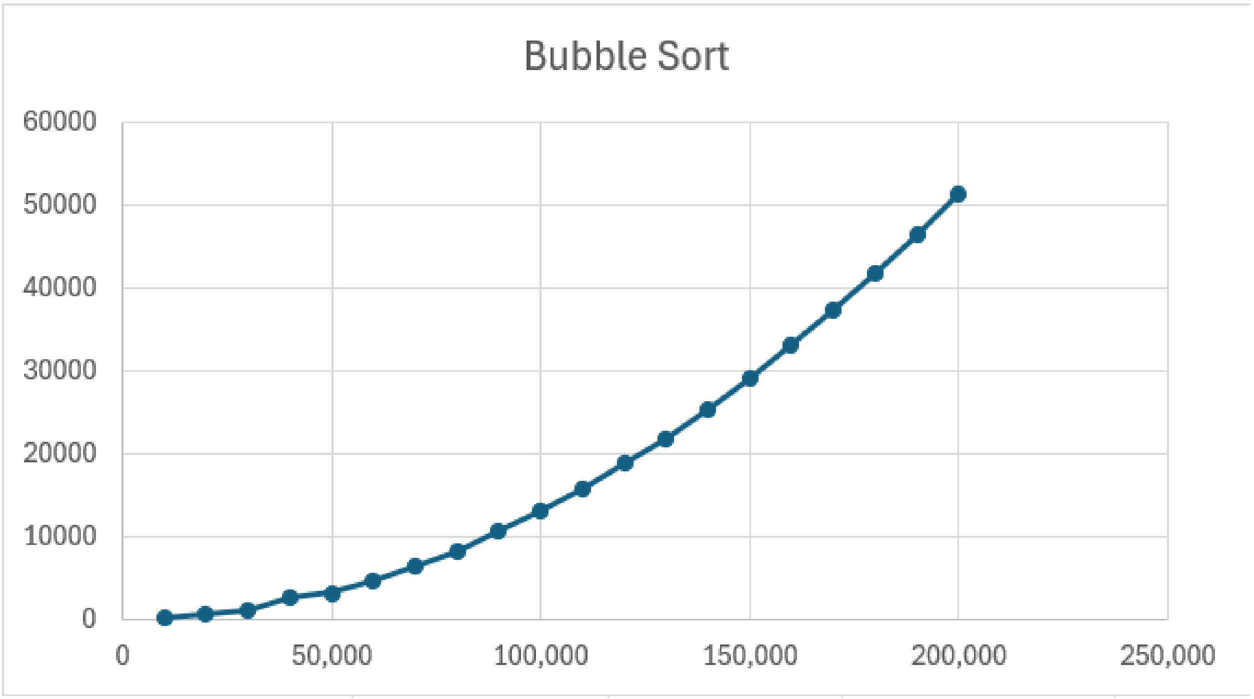
Below is the data graphed in Excel for each sorting method and their accompanying graphs. Alongside these, is the total amount of time for each sorting method passing through 10-200 thousand values. As stated above, merge, shell, quick and heap sort worked abnormally fast, and it could very well be due to an incorrect timing method implementation. Here are the attached photos for the collected data:

Amount to Sort	Bubble Sort	Selection Sort	Insertion Sort	Merge Sort	Shell Sort	Quick Sort	Heap Sort
10,000	178	40	31	3	2.97	1	2
20,000	711	151	95	2	2.45	1	3
30,000	1176	328	183	3	2.98	2	4
40,000	2649	334	329	5	4.1	2	4
50,000	3238	520	548	7	5.11	4	5
60,000	4691	749	794	6	6.23	3	5
70,000	6406	1022	1075	7	7.53	5	7
80,000	8322	1339	1413	9	9.1	5	8
90,000	10619	1679	1779	11	10.48	6	8
100,000	13096	2080	2195	10	11.33	7	9
110,000	15729	2476	2625	10	12.37	8	10
120,000	18805	3071	3150	12	14.1	7	12
130,000	21846	3465	3670	15	15.4	9	13
140,000	25314	4032	4261	17	16.41	9	13
150,000	29035	4617	4880	19	17.77	10	15
160,000	33207	5266	5573	19	19.88	10	16
170,000	37379	5957	6282	21	21.26	11	17
180,000	41883	6660	7057	23	22.24	12	18
190,000	46417	7384	7818	20	23.25	13	19
200,000	51409	8191	8663	21	24.45	13	20
TOTAL IN SECONDS	372.11	59.36	62.42	0.24	0.24	0.14	0.21

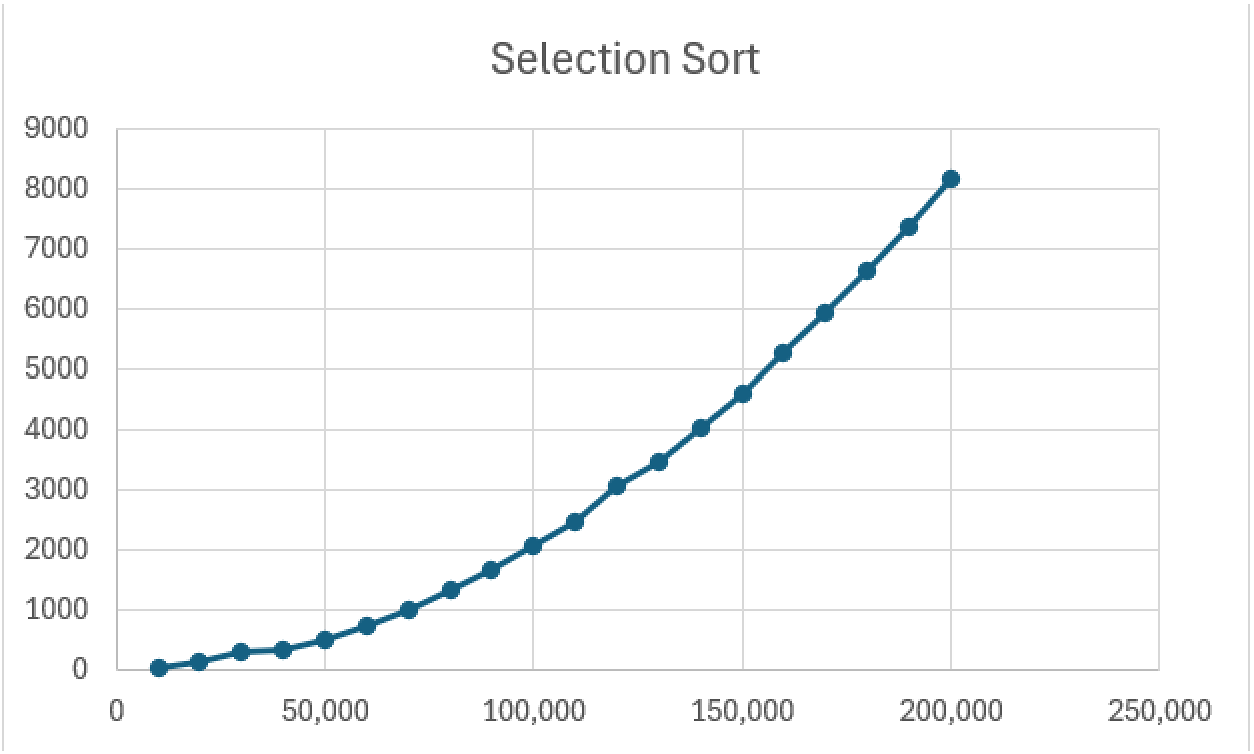
The time in milliseconds needed to sort each increment of the automatically created array.



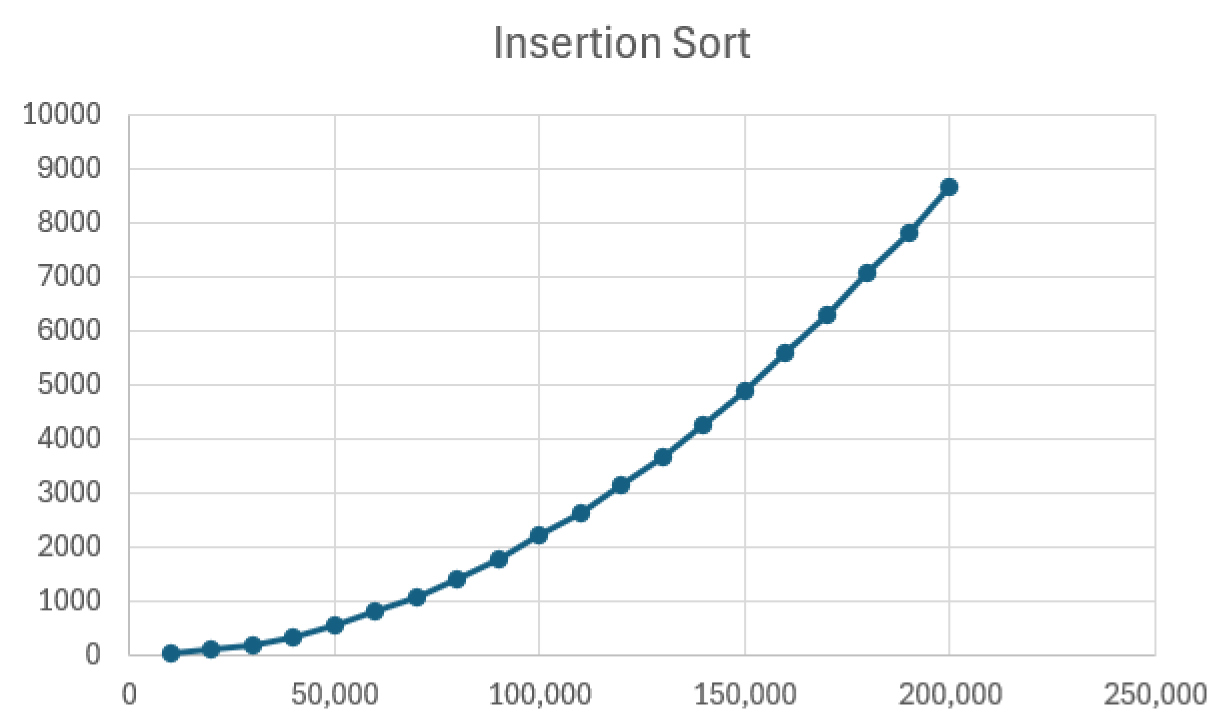
A graphed representation of the total time it takes to sort through 10-200 thousand values for each sorting method.



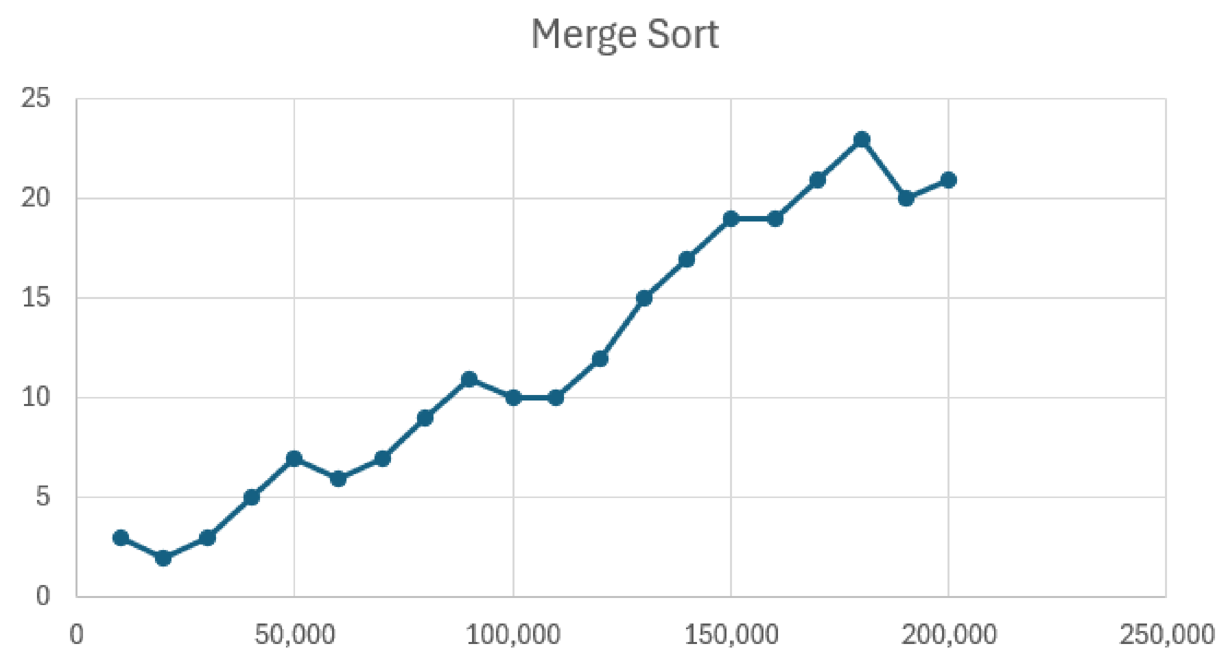
A graphed representation of bubble sort.



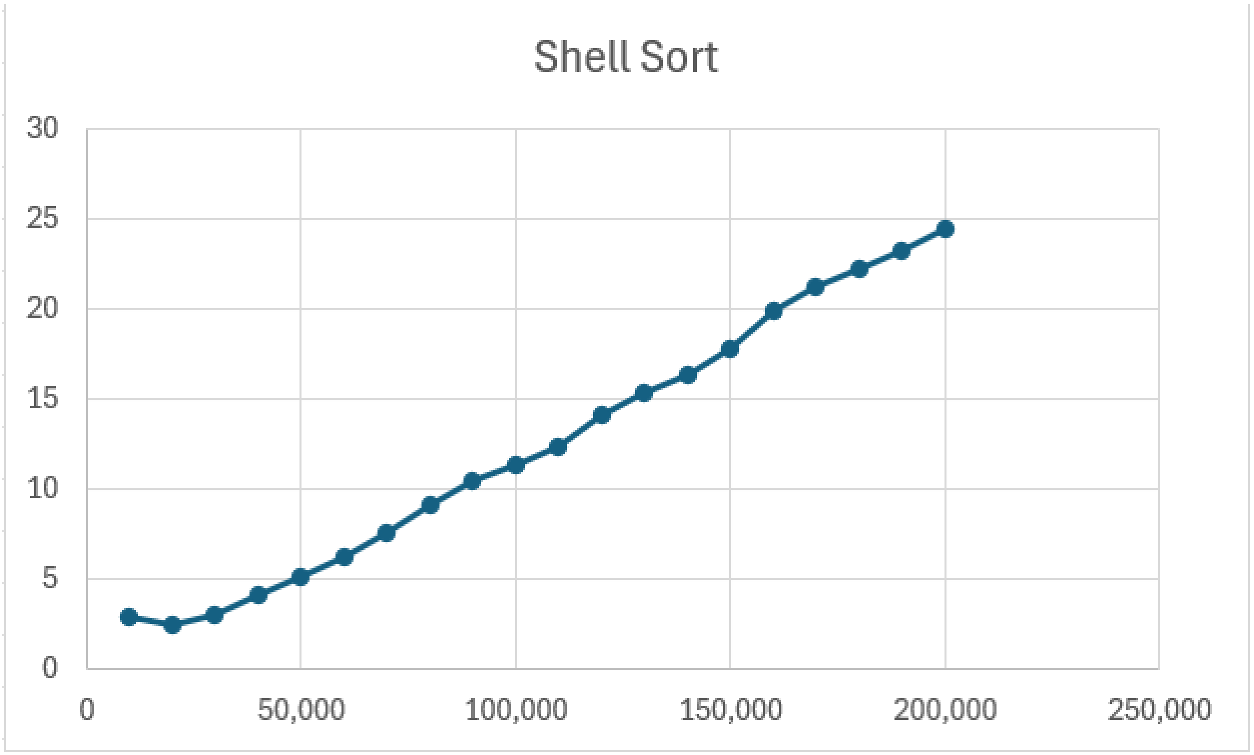
A graphed representation of selection sort.



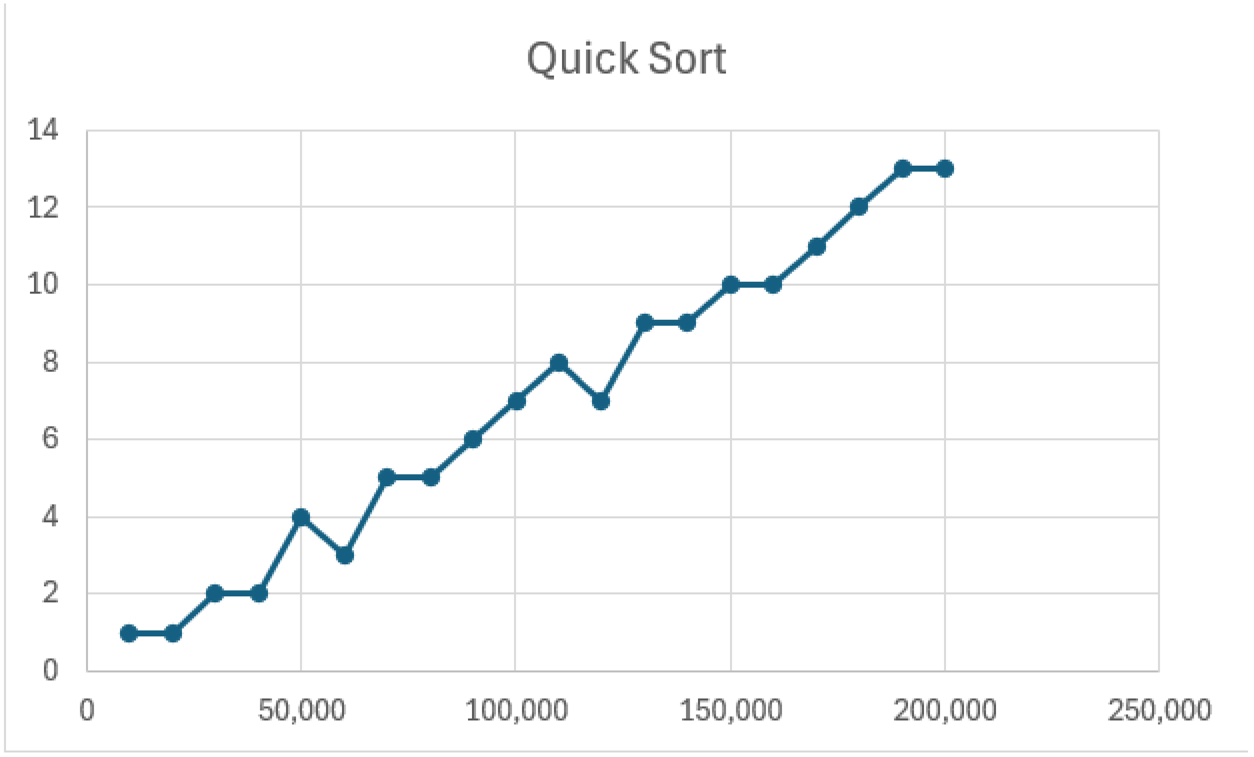
A graphed representation of insertion sort.



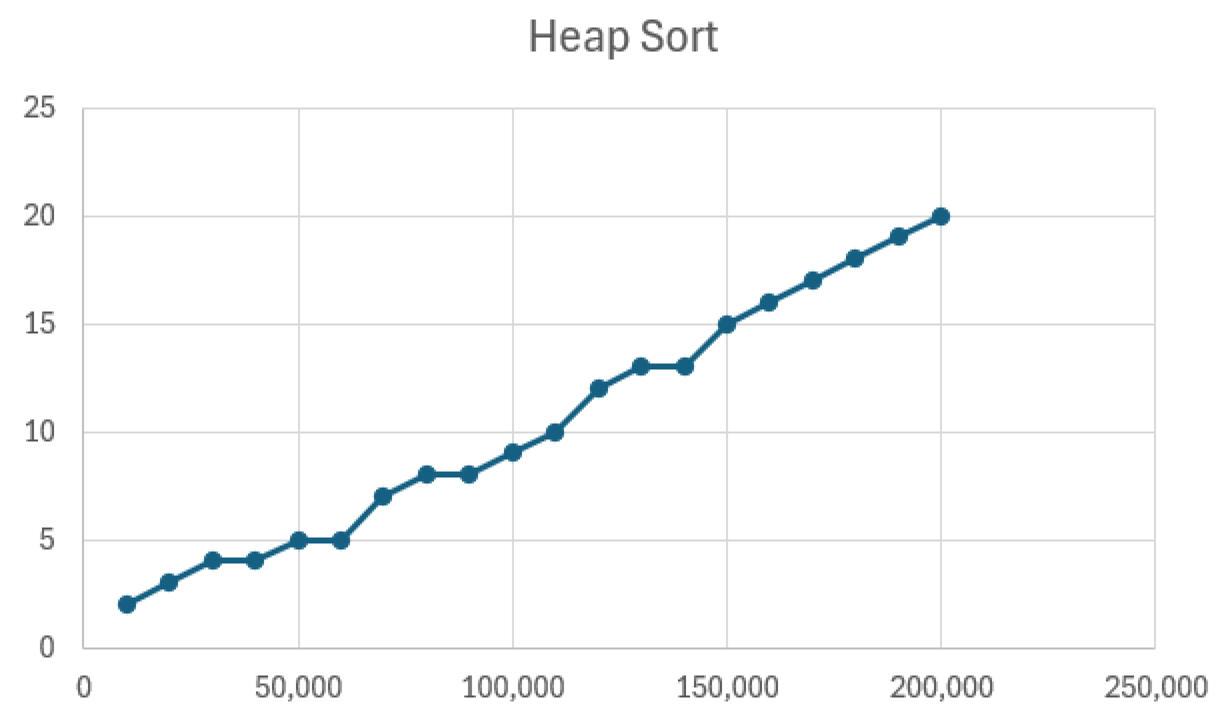
A graphed representation of merge sort.



A graphed representation of shell sort.



A graphed representation of quick sort.



A graphed representation of heap sort.

Overall, many of the sorting methods seem to increase timing in a similar, linear fashion looking across each of them. I still am unsure as to why my findings show an incorrect timing method, and I am unsure if I could find a way to fix it compared to my first three sorting methods. Based on my findings though, quick sort is the most optimal sorting method, and will provide the best and fastest times to sort even a larger set of data given.

Section 10: Addendum

I would like to give credit to GeeksForGeeks in creation of some of these sorting algorithms, as I was unfamiliar with many of them before starting this project. More specifically, the four methods that timing broke for, along with insertion as I was unable to remember from class for insertion, I had viewed GeeksForGeeks to aid in creating these methods. Links below are posted for the sorting algorithm information turned to:

Insertion Sort: https://www.geeksforgeeks.org/insertion-sort-algorithm/

Merge Sort: https://www.geeksforgeeks.org/merge-sort/

Shell Sort: https://www.geeksforgeeks.org/shell-sort/

Quick Sort: https://www.geeksforgeeks.org/quick-sort-algorithm/

Heap Sort: https://www.geeksforgeeks.org/heap-sort/