Edwin Quintuna, Wanyi Chen, Rakib Hassan

MAC 286: Data Structures in Java

Professor Muller

Abstract

The sorting algorithm is relevant to Java programmers, and programmers in general, because sorting helps programmers organize a set of data. This can be potentially time consuming, depending on the sorting type. This project content gives an explanation for three sorting types in Java: bubble sort, insertion sort, and selection sort. The algorithm for which the sorting type uses will be explained in both plain English as well as in Java code. The efficiency of the sorting type will also be explained using "Big-O."

The goal for this project is to see which sorting type is most efficient and reliable for the user. To do this, multiple tests will be run with unique seed numbers and arrays of different sizes to see how the sorting type runs with data of larger size. The elapsed time will be recorded for the different test runs and the data will be exported into Microsoft Excel to easily grasp the data presented to the user. Using this graph, it can be concluded which sorting type is best out of the three being compared.

<u>Introduction</u>

The sorting problem is one that can be answered in many different ways. Sorting algorithms have attracted the interests of many scientists, but that interest especially grew during the 1950's. It was during this time that technology was taking huge leaps forward and computers increased in popularity. There are many different sorting algorithms that have been developed since. In 1945, the merge sort was developed by John von Neumann. This sort used the "divide and conquer" algorithm to split a list into many lists and put them back together until everything is sorted back into a single list.[5][7] Another common sort that is still seen today is quicksort, developed by C. A. R. Hoare in 1962.[5] Quicksort uses a "less-than" relation as it compares different values.[6] Technology must always improve as time moves forward. Anything that is invented to replace old technology must be faster, otherwise it defeats the purpose of replacing the old. Quicksort is two to three times faster than merge sort. Even to this day, new sorting types are being developed. This can be seen with the "Timsort dating to 2002, and the library sort being first published in 2006." [4] Sorting algorithms are essential to anyone interested in the computer science field. Some of the fundamental sorts that every student should know are the **selection sort**, **insertion sort** and **bubble sort**. These three sorts have similar attributes, but vary in efficiency.

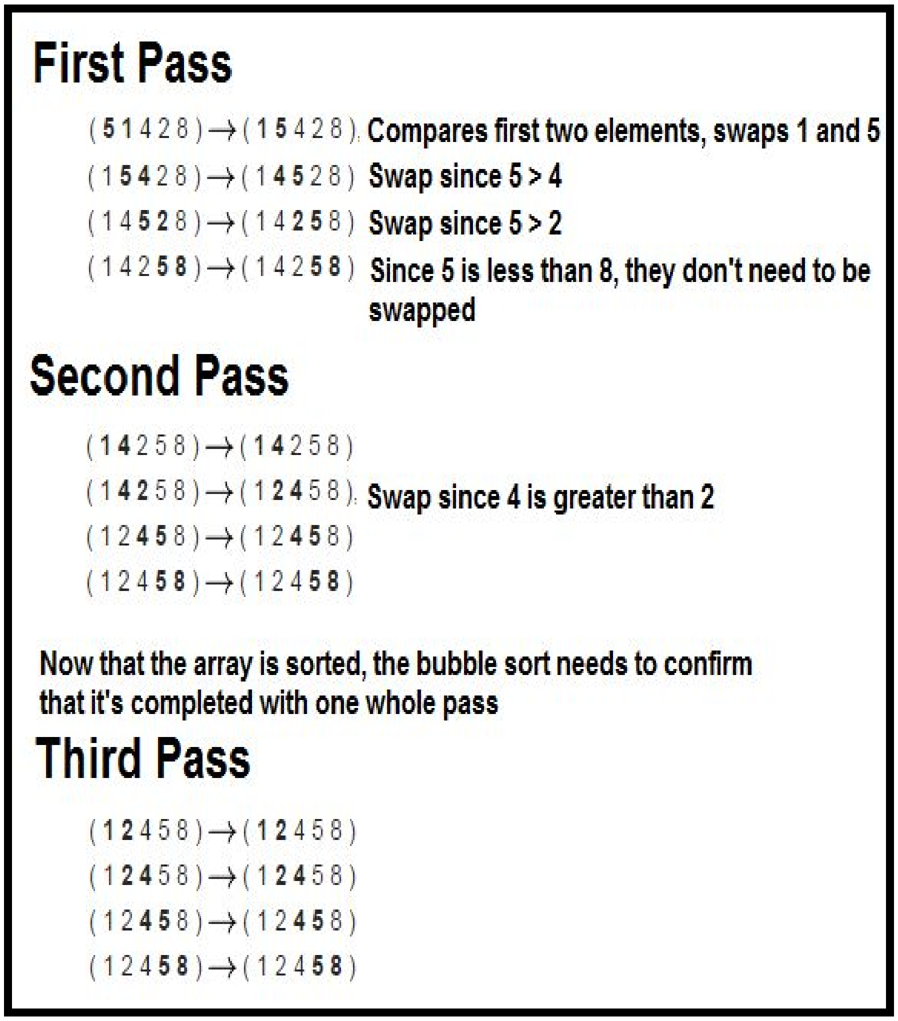
The algorithm for bubble sort is the simplest out of the three sorting types. Bubble sort takes the first item and compares it to the item on the right. If the item is larger than the item to the right, then the two items are swapped and the focus is changed to the next item and the same comparison is done. By the time the end of the list is reached, the item on the very right will be the largest item on the list. This will be procedure will be repeated several times, but with one less comparison each time since the items on the right will not need to be sorted since it will already be in the correct position from the last comparison run.

The algorithm for selection sort is a bit more complex. Selection sort goes through the list of items and looks for the smallest item. This smallest item is taken to the beginning of the list and is considered “sorted”. Then the same procedure is repeated for the array once more, but this time, the sorted portion of the array is excluded from the search. This continues to be done until everything is sorted.

The algorithm for insertion sort is the most difficult to grasp, and especially to code, but it is the most efficient sorting type of the three being compared. For insertion sort, the first item is considered sorted. After this the next item is taken and compared to the items in the sorted section. The items in the sorted list that are larger than the item are shifted to the right and the item is placed into its rightful position. This process is repeated until the list is completely sorted.

Bubble Sort

All sorts must satisfy two conditions and these are that “the output is in nondecreasing order” and “that the output is a permutation of the input.”[4] This can be accomplished in many different ways, but some ways are more efficient than others. The name “bubble sort” has its origins dating back to Kenneth Iverson’s work, *A Programming Language*, in 1962. Bubble sort is a very popular algorithm and is classified as a O(n^2) algorithm. If bubble sort is given a list that it is out of order, sorting will be accomplished through multiple iterations. The algorithm goes through the list multiple times and while it is doing so, neighboring values are swapped if it is not in the correct order. In the example, each iteration is called a pass and since 5 is followed by a 1, these two values need to be swapped. Here is where the problem can be seen with bubble sort as its complexity can lead to large amounts of comparisons. In the worst case scenario, the sort takes O(n^2) to sort a list, with the best case scenario taking O(n).[3] On average, it takes the sort O(



First Pass

```
( 5 1 4 2 8 ) → ( 1 5 4 2 8 ), Compares first two elements, swaps 1 and 5
( 1 5 4 2 8 ) → ( 1 4 5 2 8 )  Swap since 5 > 4
( 1 4 5 2 8 ) → ( 1 4 2 5 8 )  Swap since 5 > 2
( 1 4 2 5 8 ) → ( 1 4 2 5 8 )  Since 5 is less than 8, they don't need to be
                               swapped
```

Second Pass

```
( 1 4 2 5 8 ) → ( 1 4 2 5 8 )
( 1 4 2 5 8 ) → ( 1 2 4 5 8 ), Swap since 4 is greater than 2
( 1 2 4 5 8 ) → ( 1 2 4 5 8 )
( 1 2 4 5 8 ) → ( 1 2 4 5 8 )
```

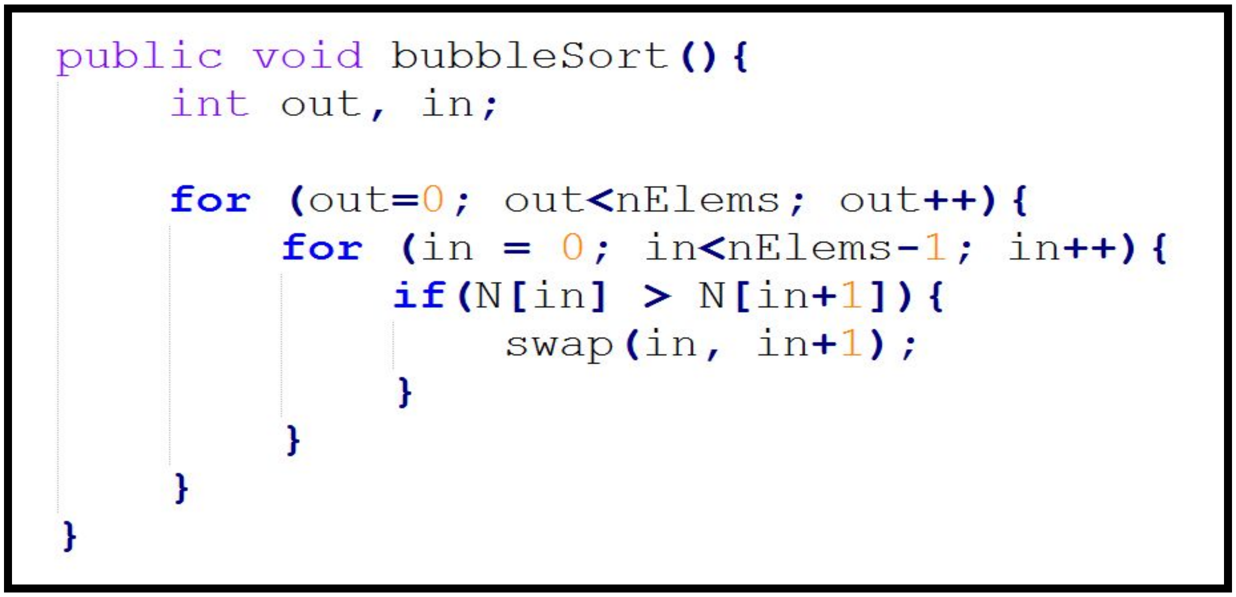
Now that the array is sorted, the bubble sort needs to confirm that it's completed with one whole pass

Third Pass

```
( 1 2 4 5 8 ) → ( 1 2 4 5 8 )
( 1 2 4 5 8 ) → ( 1 2 4 5 8 )
( 1 2 4 5 8 ) → ( 1 2 4 5 8 )
( 1 2 4 5 8 ) → ( 1 2 4 5 8 )
```

n^2) to sort a list, which makes it difficult to implement bubble sort into an efficient program. [3] Bubble sort is best suited for lists that are close to being sorted.

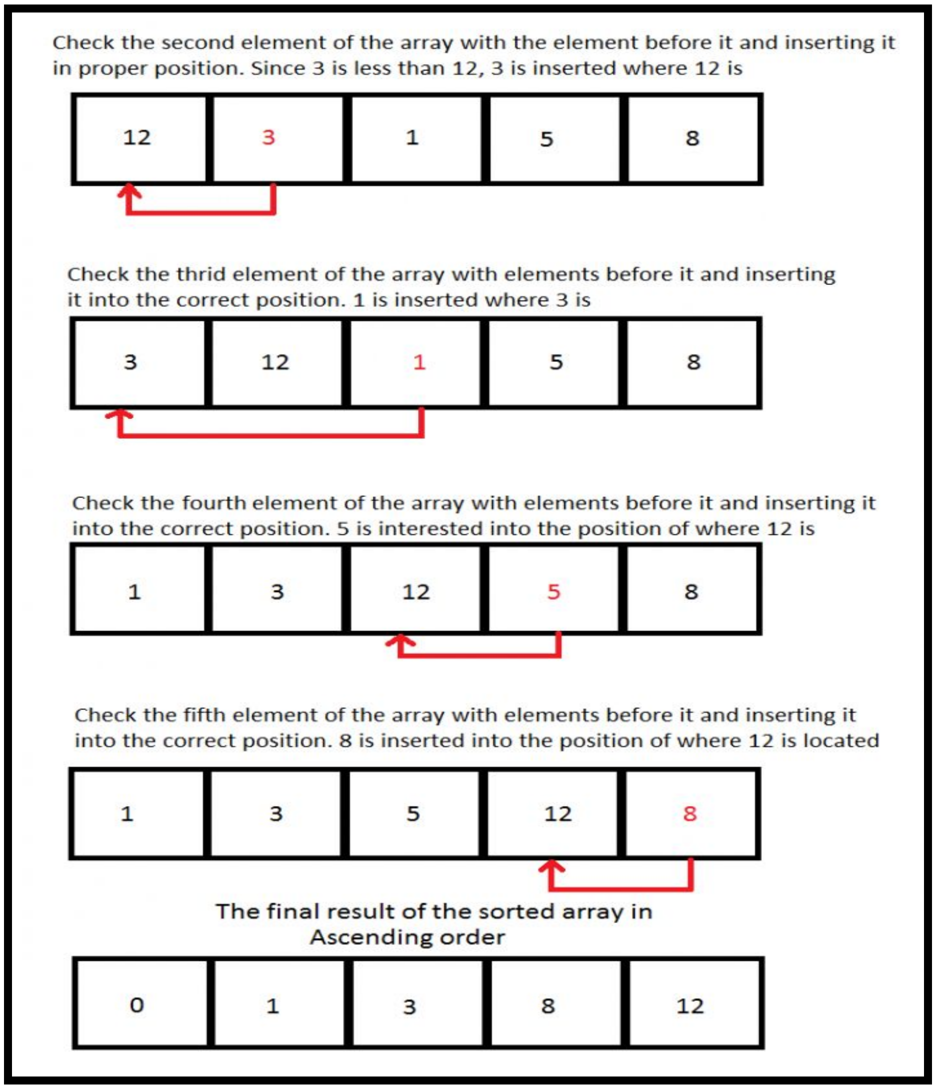
To implement **bubble sort** in Java, two for-loops, an if statement, and a swap function are used. In the inner for-loop, “in” and “in+1” are two adjacent items in that order. If “in” is greater than “in+1”, then the two items are swapped. Then “in” is changed to the next value in the list and the same procedure is done. This will continue to happen until the end of the list is reached. By the end of the inner for-loop, the largest item will be on the right side of the array. To sort the entire array, the outer for-loop is required to make the test repeat itself. With each incrementation, the number of sorted items on the right will increase until the entire array is properly sorted.



```
public void bubbleSort(){
    int out, in;

    for (out=0; out<nElems; out++){
        for (in = 0; in<nElems-1; in++){
            if(N[in] > N[in+1]){
                swap(in, in+1);
            }
        }
    }
}
```

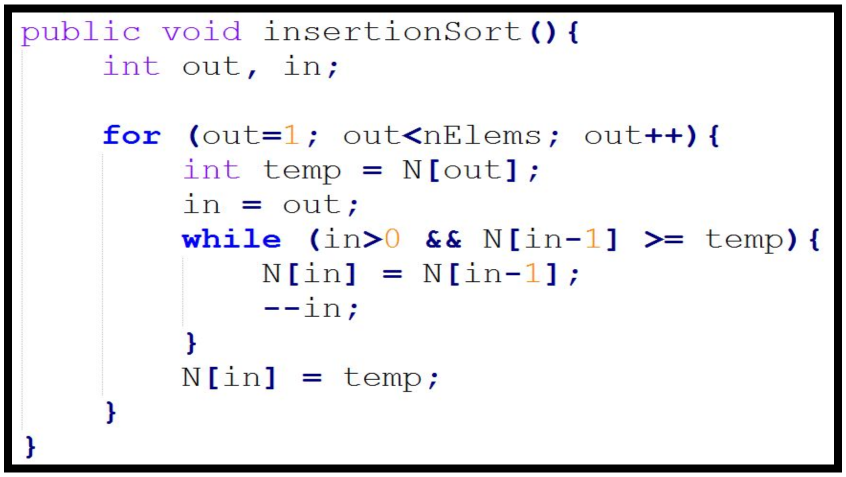
Insertion Sort

The **insertion sort** algorithm improves on the bubble sort in several different ways, making it more suitable for implementation in programs that have the need for a sort. Like bubble sort, it still has a worst case scenario of O(n^2), a best case scenario of O(n) and an average case of O(n^2) [2]. However, this sort can adapt to be more effective depending on the circumstances. When sorting a list the insertion sort looks at the first item and then proceeds to read down the list until it finds one of lower value. After it is found, these two items are swapped. The sort does this until there are no items of lower value. This process is done for the rest of the values on the list.[2] Insertion sort is adaptive and is considered an "online algorithm, which means it can start sorting before it


Check the second element of the array with the element before it and inserting it in proper position. Since 3 is less than 12, 3 is inserted where 12 is
12 3 1 5 8
Check the thrid element of the array with elements before it and inserting it into the correct position. 1 is inserted where 3 is
3 12 1 5 8
Check the fourth element of the array with elements before it and inserting it into the correct position. 5 is interested into the position of where 12 is
1 3 12 5 8
Check the fifth element of the array with elements before it and inserting it into the correct position. 8 is inserted into the position of where 12 is located
1 3 5 12 8
The final result of the sorted array in Ascending order
0 1 3 8 12

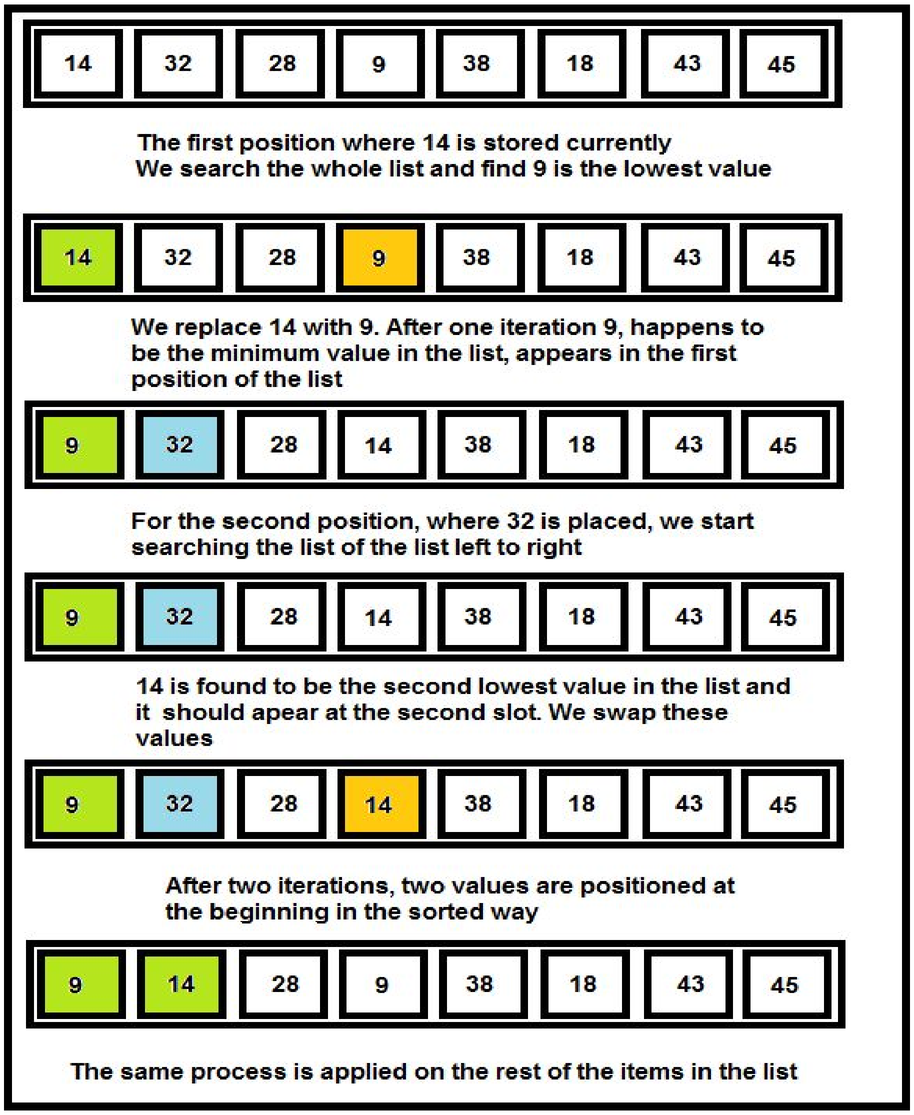
gets all the items and then merge the lists once the partial sorting has completed.”[2] If a list [4,6,8,9,1] is given, all that insertion sort needs to do in order to sort this list is shift [4,6,8,9] one index up, as it is already sorted, and move 1 to the front of the list, giving us the sorted list [1,4,6,8,9]. [2]

When implementing insertion sort into Java, there is no need for two “for-loops”. Only one “for-loop” and one “while-loop” is enough to implement the insertion Sort. Variable “out” is assigned the location of the value of the item that needs to be checked. “N[out]” has the actual value using “out” as a reference. Variable “temp” stores this value temporarily and location “out” is stored in variable “in.” The while statement will only execute if the previous value from location “in” is smaller than the value stored in location “in.” If this condition is met, an item with a lower value than the item in location “in” is found. If this is the case, the smaller value is assigned into “N[in].” If that condition is not met, then there is no smaller or equal value in value to “in,” which is just the variable “out.” On its own, insertion sort is only efficient when it comes to smaller data sets, partially sorted lists, and live data. It is for this reason that insertion sort is implemented on larger sort programs such as shell sort. Shell sort was invented by Donald Shell in 1959. This type of sort does an insertion sort on all the iterations of the list. [10] Insertion sort is a good building block for developing larger and more powerful types of sort algorithms.

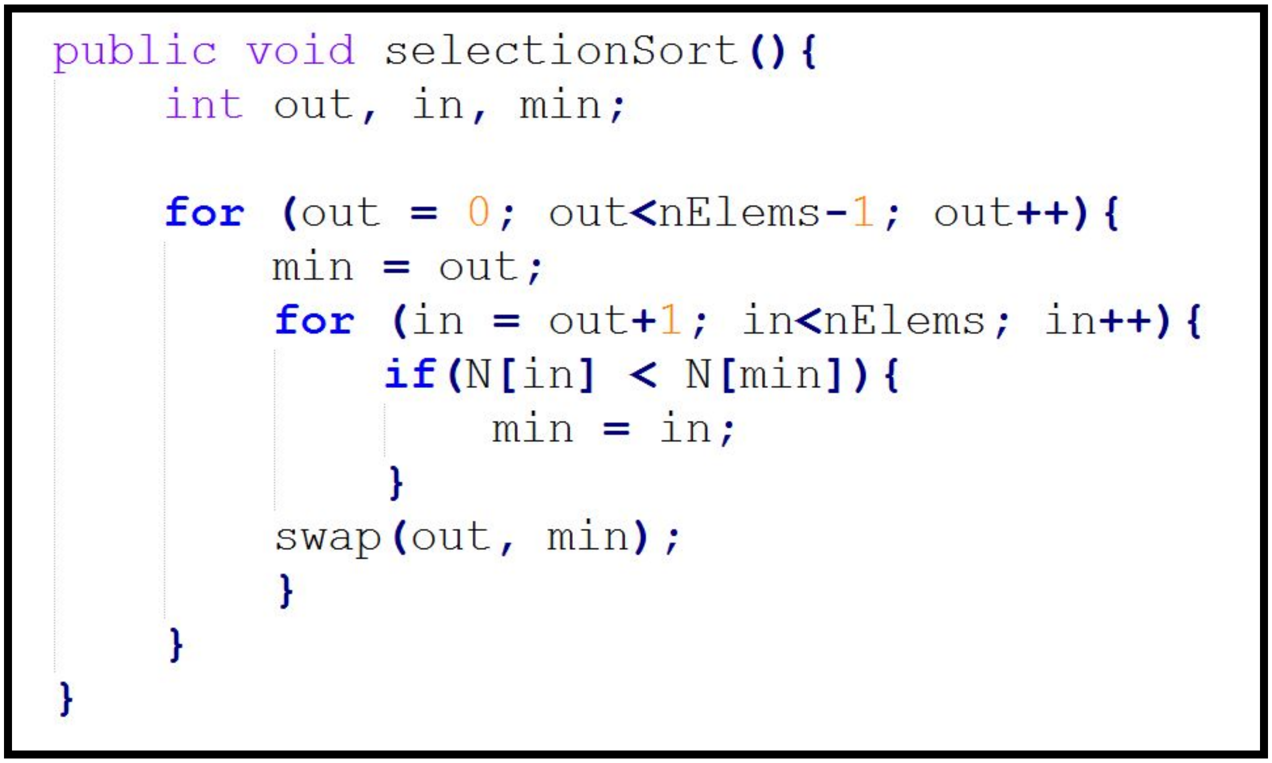

```
public void insertionSort(){
    int out, in;

    for (out=1; out<nElems; out++){
        int temp = N[out];
        in = out;
        while (in>0 && N[in-1] >= temp){
            N[in] = N[in-1];
            --in;
        }
        N[in] = temp;
    }
}
```

Selection Sort

Selection sort is a step down from Insertion sort as its sorting time is much slower. Selection sort has a worst case scenario of O(n^2), a best case scenario of O(n^2) and an average case scenario of O(n^2).[1] Insertion had a best case scenario of O(n), but because of the way selection sort is structured, it's worst and best scenario take the same amount of time and comparisons. Selection sort goes through the list and finds the item of least value.[1] Once this value is found, it is swapped with the first value of the list. On the next iteration, the first value of the list is no longer taken into account as it should have the smallest value.


14 32 28 9 38 18 43 45
The first position where 14 is stored currently
We search the whole list and find 9 is the lowest value
14 32 28 9 38 18 43 45
We replace 14 with 9. After one iteration 9, happens to be the minimum value in the list, appears in the first position of the list
9 32 28 14 38 18 43 45
For the second position, where 32 is placed, we start searching the list of the list left to right
9 32 28 14 38 18 43 45
14 is found to be the second lowest value in the list and it should apear at the second slot. We swap these values
9 32 28 14 38 18 43 45
After two iterations, two values are positioned at the beginning in the sorted way
9 14 28 9 38 18 43 45
The same process is applied on the rest of the items in the list

When implementing this to Java, the code is very similar to that of bubble Sort. There are still two for-loops to keep track of iterations and specific elements, but for selection sort, “min=in;” is included. “min” stores the location of the smallest value of the list. As for “in,” it has the current location of the value being compared to. N[min] gives the value of the most current smallest value in the list. If N[in], which has the value of some other item on the list, is smaller than N[min], then a smaller value is found, thereby it becomes the new min. Since “in” needs to be the new “min,” “min” is made equal to “in.” Then a swap function is used to swap the location of “out” and “min.” After all of this the loop starts again, but now with the new “out,” “in,” and “min” values.



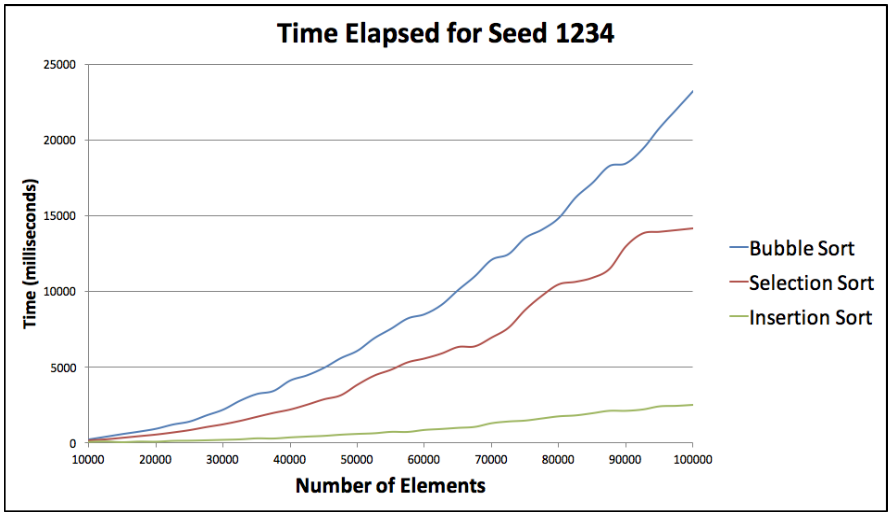
```java
public void selectionSort(){
    int out, in, min;

    for (out = 0; out<nElems-1; out++){
        min = out;
        for (in = out+1; in<nElems; in++){
            if(N[in] < N[min]){
                min = in;
            }
        swap(out, min);
        }
    }
}
```

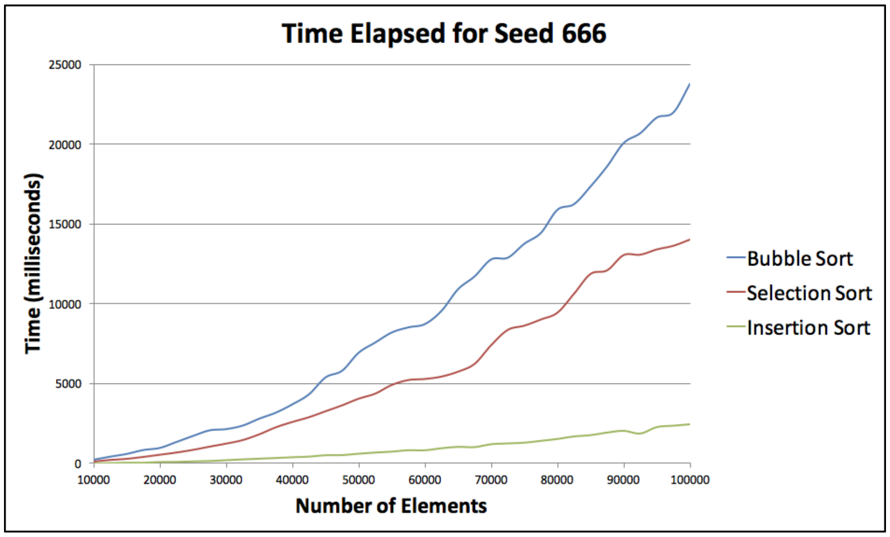
Experiment and Results

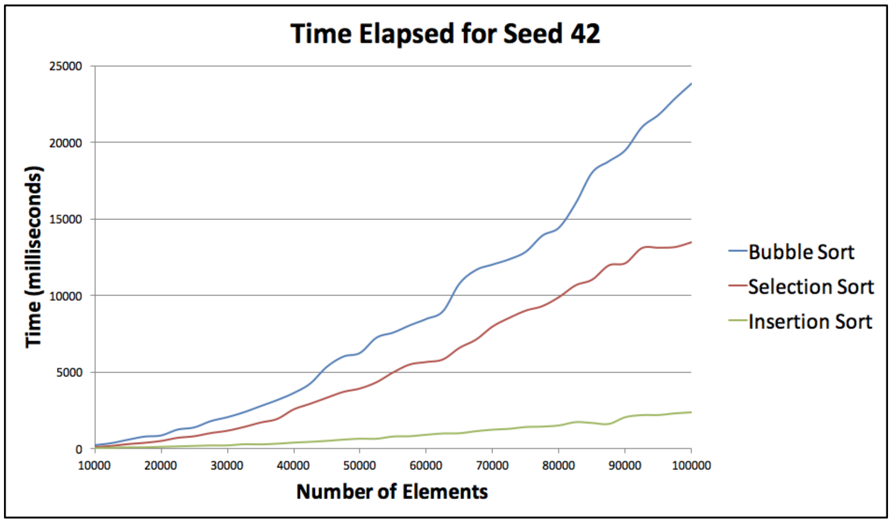
To test which sorting type is best, a for-loop is used to create an integer array with the initial index size of 10,000. The array is then filled with random integers using the "Random setSeed" function. The array is then sorted using bubble sort and the time is recorded in milliseconds. The time is recorded by taking the current time before the array is sorted, and taking the current time after the array is sorted and subtracting the two values. This will be recorded as "elapsed-time." After this value is recorded, the array size is incremented by 2,500, and the same test is done until 100,000 elements is reached. This for-loop is redone using selection sort and insertion sort. After the different elapsed times for different array sizes is recorded for all three sorting times, the entire test is redone using different seed numbers to ensure the validity in the results.

After running the tests, it is clear that insertion sort is, by far, the superior sorting type. With all three sorting types, as the number of elements increased, so did the time required to sort, as expected. The time required for bubble sort and selection sort increased drastically, yet the insertion sort required a significantly lesser time. Even with 100,000 elements to sort, insertion sort did not take over 2500 milliseconds in any of the tests. This is especially astounding considering the fact that bubble sort and selection sort would take about 24,000 milliseconds and 14,000 milliseconds respectively for 100,000 elements. Looking at the graph, bubble sort is the slowest sorting type, followed by selection sort, and then insertion sort; however, bubble sort and selection sort appear to have similar results, still taking a long time to sort an integer array of a large size. Neither of the two sorting types are close to insertion sort.

Elapsed Time Graphs for Bubble Sort, Selection Sort, and Insertion Sort:







Conclusion

Even though bubble, insertion and selection sorts are similar in structure and complexity, it is insertion that provides the most efficient sort. According to the results, bubble sort is the slowest as the number of elements. For 100,000 elements in an array, it took on average about 24,000 milliseconds to sort the array in increasing order. For selection sort, it took about 14,000 milliseconds for the same arrays. Insertion is easily the fastest when compared to bubble sort and selection sort when the elements increase. The tests were even repeated with different seed numbers to ensure validity and consistency between the different test results. Even with 100,000 elements, it took less than 2,500 milliseconds on all tests. With these results, insertion sort is about 10 times faster than bubble sort and 6 times faster than selection sort. When compared to the results of others, it is even more clear that insertion sort is by far, the fastest sorting type of the three. This is especially interesting considering the fact that the average time complexity for all three sorting types are $O(n^2)$. Looking at the Big O alone, one might assume that the three sorting types would take about the same time sorting a similar array; however, the actual tests show otherwise.

It is important to note, however, that the results may vary with each test run. This is because the computer cannot simulate the exact same speed each time. There may be slight hiccups in another run that can slightly alter the numbers. It is also important to note that different computers may present different results. This is because a computer might be faster or slower than another; however, even with these differences, the tests should ultimately show that insertion sort is the superior sorting type.

Bibliography

Imms, Daniel. "Selection sort." *Growing with the Web*, 8 Dec. 2013, www.growingwiththeweb.com/2013/12/selection-sort.html. [1]

Imms, Daniel. "Insertion sort." *Growing with the Web*, 10 Nov. 2012, www.growingwiththeweb.com/2012/11/algorithm-insertion-sort.html. [2]

Imms, Daniel. "Bubble sort." *Growing with the Web*, 20 Feb. 2014, www.growingwiththeweb.com/2014/02/bubble-sort.html. [3]

"Sorting algorithm." *Wikipedia*, Wikimedia Foundation, 4 Dec. 2017, https://en.wikipedia.org/wiki/Sorting_algorithm. [4]

"Timeline of algorithms." *Wikipedia*, Wikimedia Foundation, 6 Nov. 2017, en.wikipedia.org/wiki/Timeline_of_algorithms.[5]

"Quicksort." *Wikipedia*, Wikimedia Foundation, 1 Dec. 2017, https://en.wikipedia.org/wiki/Quicksort [6]

"Merge sort." *Wikipedia*, Wikimedia Foundation, 7 Nov. 2017, https://en.wikipedia.org/wiki/Merge_sort.[7]

PowerPoint [8]
Hassan, Rakib. *Sort Types*. 2017, *Sort Types*.

Garg, Vinayak. "Time Comparison of Quick Sort, Insertion Sort and Bubble Sort." *Tech Blog*, 16 Oct. 2016, https://vinayakgarg.wordpress.com/2011/10/25/time-comparison-of-quick-sort-insertion-sort-and-bubble-sort/ [9]

"Sorting Algorithms." *Back to Joshua Kehn*, 1 Oct. 2010, www.joshuakehn.com/2010/10/1/Sorting-Algorithms.html. [10]