Design and Analysis of Algorithms (CS-3364-001) - Project 1

Goal: Find source reliability by counting inversions during sorting

Contributors:

1. Alfonso Islas	(R11623068)
2. Nathan Rissman	(R11759554)
3. Mohammad Seeam	(R11771075)
4. Lucas Woldegiorgis	(R11884106)

Role of Contributors:

Alfonso	- Implemented Merge Sort
Nathan	- Implemented Bubble Sort
Lucas	- Implemented Quick Sort
Mohammad	- Programmed main() function and file handling

Problem Interpretation:

Sorting Algorithms are used to organize collections of data in multiple applications such as Search Engines, Machine Learning, Deep Learning, Artificial Intelligence, Product Rankings, Inventories, Databases and so on. The raw data enters unsorted, which introduces time complexity when organizing. The level of "unsorted-ness" in the data can be represented by the number of inversions present in the data list. This problem can be observed when understanding lists of combined rankings. Some sources will provide better combined ranking results. In order to categorize the efficacy of these sources, we can count the number of inversions in their combined rankings.

Source reliability is inversely proportional to the number of source inversions. Thus, if a source's inversion count is known, the source's reliability can be known as well.

Another common usage of search lists is sorting. Sorting is simply the correction of inversions. Since a sorting algorithm is already automatically correcting these inversions, a counter can be added to key points of the sorting process to keep track of the number of inversions encountered when sorting the data.

The objective of this project is to analyze Quick-Sort (O(nlogn)), Merge-Sort (O(nlogn)), Bubble Sort (O(n^2)) and modify these algorithms to count the inversions in the data supplied to them. Using these modified algorithms, 5 sources of combined rankings are to be sorted and have their inversions counted. The resulting inversion counts will be used to determine source reliability with higher numbers of inversions being less reliable and lower inversion counts being more reliable.

It is important to recognize from the outset that all 3 algorithms should return the same inversion counts and sorted data for each respective source since these are properties of the data sets and should not be affected by which algorithm is used.

Solution Methodology:

The inversionCount.java file contains a class called inversionCount. Inside the inversionCount class, we have the main function and other necessary functions to perform different sorting algorithms and calculate inversions.The program starts from the main(String[] args) function.

We made an int named ‘inversions’ to count inversion.

```
static int inversions = 0;
```

Then we made two ArrayList.

```
static   ArrayList<Integer>   inversionCompariosnCounts   =   new
ArrayList<Integer>();
static    ArrayList<String>    inversionCompariosnFiles    =    new
ArrayList<String>();
```

We used these ArrayLists to store all inversions and file names so that later we can find the least number of inversion and the file.

Inside the main() function, we run a for loop to read all 5 files which were provided. The “FileReader” function simply reading a specific file and filling ranklist[] array with all values. We made an array called ranklist where we store all 10000 int values of a specific file on each iteration of the for loop. Then we called the CountInversions() function to perform sorting algorithms to find inversion. We compare inversions and finally return the file with the least number of inversions.

The CountInversions() function, calls either QuickSortInvertCount(), BubbleSort() or MergeSort() based on user given arguments.

Merge Sort:

The Merge Sort algorithm, a divide-and-conquer strategy for sorting, is implemented by the MergeSort function. It accepts as input an array ranklist and its length. First, it is necessary to examine the base case: if the length is 1, the function returns right away because there is nothing to sort. If the initial array length is odd, it divides the array into two sub-arrays, L and R, while ensuring that L gets an extra element. Iterating through the ranklist initializes and populates these sub-arrays. After that, the function iteratively breaks down the sub-arrays L and R until it reaches the base case. Finally, it combines and sorts the sub-arrays L and R back into the main array using the merge function.

Two sorted sub-arrays, L and R, are combined into a single sorted array ranklist using the merge function. To keep track of the elements in the three arrays—L, R, and the merged array—it employs three counters—l, r, and i. Elements from each sub-arrays are compared in a while-loop, and then they are assigned to the ranklist in ascending order. If an element from R is assigned, it also determines the number of inversions, which is a measure of how many elements in L are greater than elements in R, and increases the inversions counter in line with that determination. The remaining elements from the other sub-array are immediately copied into the ranklist after one sub-array runs out of elements. The ranklist is kept ordered and the amount of inversions is tracked as needed thanks to this merging process. Overall, by effectively merging and sorting two sub-arrays, the merge function plays a significant part in the Merge Sort algorithm.

On line 127, we are counting inversion for merge sort.

```
inversions+=lengthL-l; //***inversions are based on if smaller value is in R
```

Bubble Sort:

The “BubbleSort()” function takes an array of int which is full of int of a specific file. It also takes the length of the array. Inside the nested loop of bubble sort, when we find $i < j$ $(i,j \geq 0)$ but $A[i]>A[j]$, condition, which is an inversion, we count inversion and return it lastly. That is how we count inversion for bubble sort.

When we meet below conditions on bubble sort, we count inversion.

```
if (ranklist[j]>ranklist[j+1]) {
```

Quick Sort:

Quick sort algorithm requires that a collection of data be sorted as it is being divided. In order to accomplish this, quick sort requires the selection of a pivot point. All values in the array or list are then compared to this pivot point and are binned into their respective sub arrays: less-than, equal, and greater. Once the base case is reached through recursion, the function pops back a merged array. Because the values are sorted as the array is separated, each merging is guaranteed to be in order with no inversions present.

The trick to counting inversions with this algorithm is in the binning process. Suppose we have an array of 7 numbers. The exact values are unimportant, only their position with respect to the pivot:

🟦	🟢	🟦	⭐	🟢	⭐	🟦

🟦 - Less Than
🟢 - Greater Than
⭐ - Equal To

Now we can create the respective sub-arrays:

Less Than

🟦	🟦	

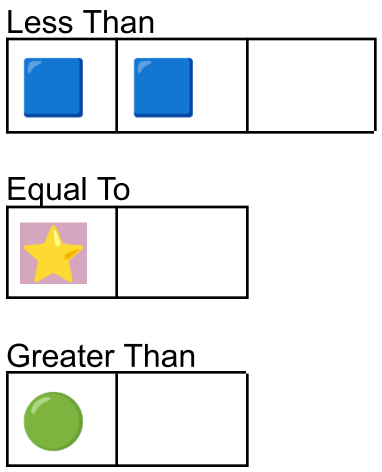
Equal To

Greater Than

🟢	

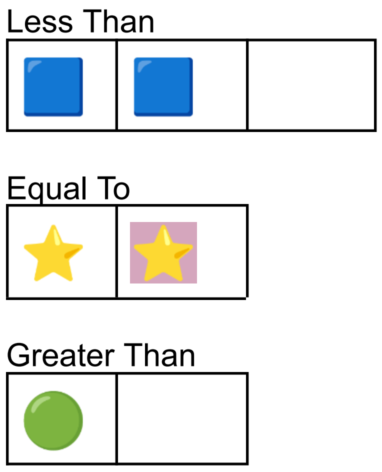
As we place the elements, their order in the unsorted array determines if there are associated inversions. Notice above when the second “less-than” element is placed, there is an inversion between it and the already sorted “greater than” element. This indicates that the number of inversions for every “less-than” element sorted is at least equal to the number of sorted “greater-than” elements.

We can make the same observation when we sort our pivot element (shown below). The number of associated inversions is equal to the number of “greater-than” elements placed.



Notice when we sort another “greater-than” element, still no inversions are resolved. Therefore, inversion resolution is independent of the sorting of “greater-than” elements.

Again, when sorting an element “equal to” the pivot element, then we see that the number of inversions resolved increases by the number of sorted “greater-than” elements.

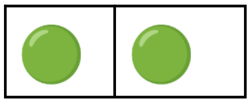


Finally, when we observe our final “less-than” element get sorted, we see that the number of resolved inversions increases by the number of “greater-than” elements AND the number of “equal-to” elements that have been sorted. Thus, we can modify our earlier observation: When a “less-than” element is sorted, the number of resolved inversions increases by the “greater-than” and “equal-to” elements that have been sorted.

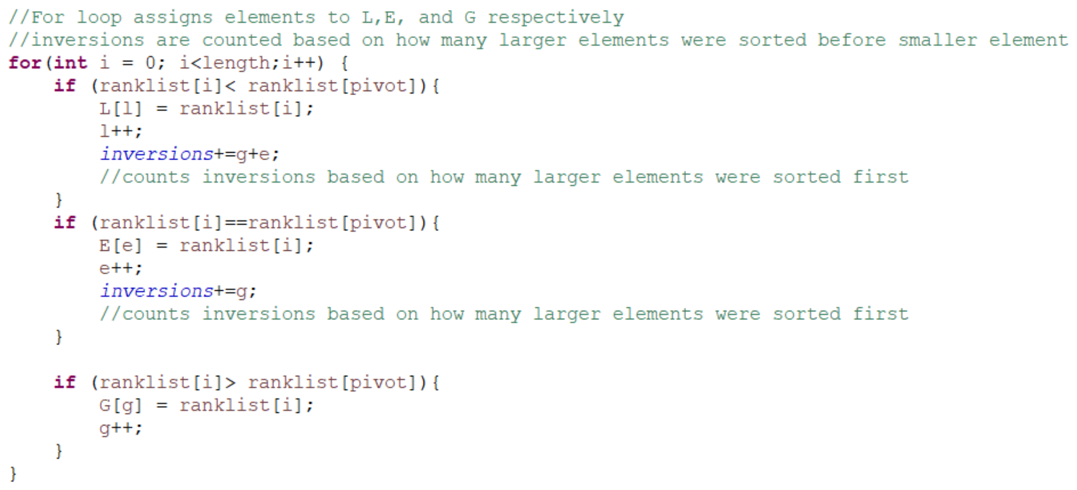
Less Than

Equal To

Greater Than



Now, using these 2 rules associated with inversion resolution for less-than and equal-to elements. In our quick-sort algorithm, we implement this observation by increasing our inversion counter by array lengths of the greater-than array (and equal-to array in the case of the less-than elements) whenever less-than and equal-to elements are sorted.



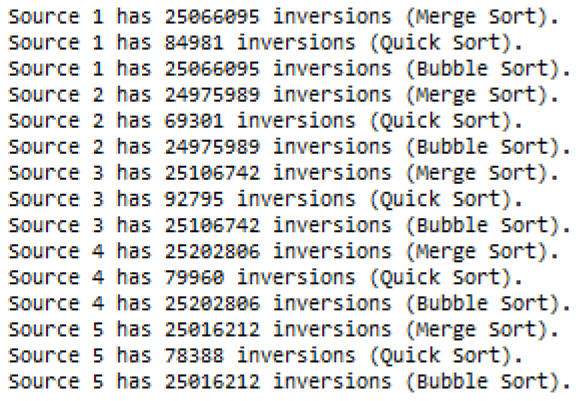
```
//For loop assigns elements to L,E, and G respectively
//inversions are counted based on how many larger elements were sorted before smaller element
for(int i = 0; i<length;i++) {
    if (ranklist[i]< ranklist[pivot]){
        L[l] = ranklist[i];
        l++;
        inversions+=g+e;
        //counts inversions based on how many larger elements were sorted first
    }
    if (ranklist[i]==ranklist[pivot]){
        E[e] = ranklist[i];
        e++;
        inversions+=g;
        //counts inversions based on how many larger elements were sorted first
    }

    if (ranklist[i]> ranklist[pivot]){
        G[g] = ranklist[i];
        g++;
    }
}
```

Experimental Results:

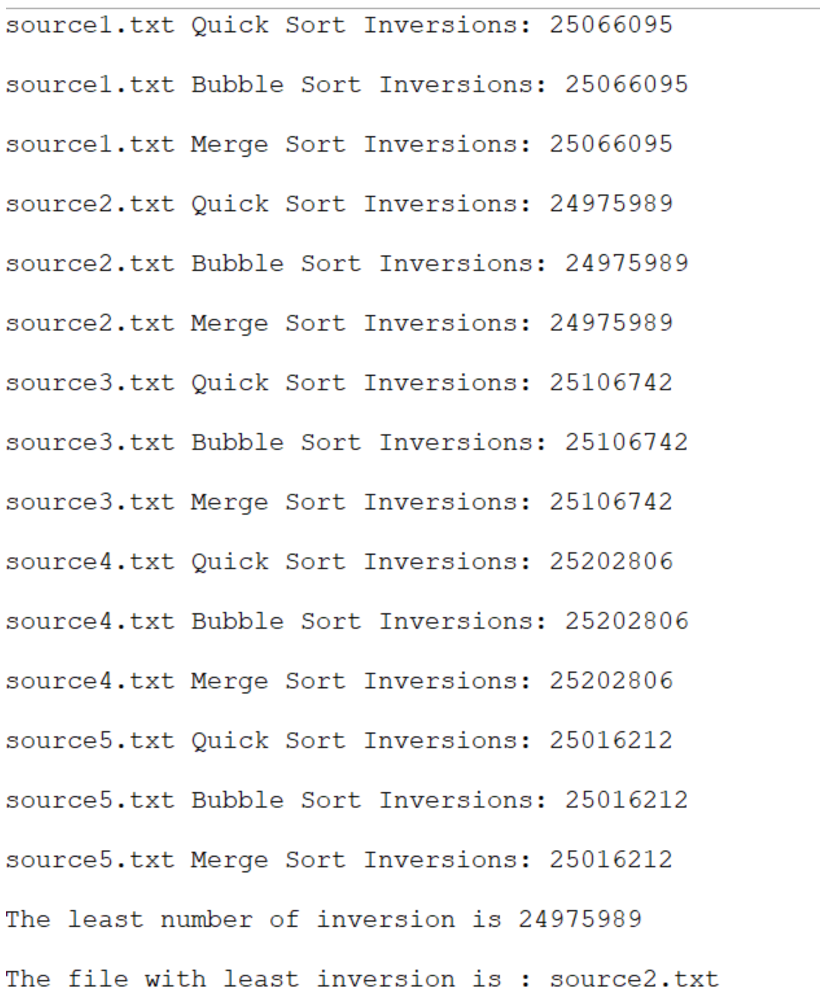
While running the algorithms each of them were able to properly sort the sources provided, but quicksort did not have the matching inversion count that mergesort and bubblesort had. The number of inversions that the quicksort counter was showing was significantly less than the results for the other two. Our first method for the quicksort was incorrect as they were all supposed to have the same inversion. For example

Source 1 had 25066095 inversions for merge and bubble sort while quicksort only had 84981. With this we were able to tell that our methodology for counting the quicksort inversions were off, so we needed to change our thinking.



```
Source 1 has 25066095 inversions (Merge Sort).
Source 1 has 84981 inversions (Quick Sort).
Source 1 has 25066095 inversions (Bubble Sort).
Source 2 has 24975989 inversions (Merge Sort).
Source 2 has 69301 inversions (Quick Sort).
Source 2 has 24975989 inversions (Bubble Sort).
Source 3 has 25106742 inversions (Merge Sort).
Source 3 has 92795 inversions (Quick Sort).
Source 3 has 25106742 inversions (Bubble Sort).
Source 4 has 25202806 inversions (Merge Sort).
Source 4 has 79960 inversions (Quick Sort).
Source 4 has 25202806 inversions (Bubble Sort).
Source 5 has 25016212 inversions (Merge Sort).
Source 5 has 78388 inversions (Quick Sort).
Source 5 has 25016212 inversions (Bubble Sort).
```

After changing how we counted the quicksort inversions the numbers were able to match the values that we got for the other two inversion counts for each source. As this is the situation that we were expecting, we were able to conclude our counting experiment knowing that source 2 had the least number of inversions.



```
source1.txt Quick Sort Inversions: 25066095

source1.txt Bubble Sort Inversions: 25066095

source1.txt Merge Sort Inversions: 25066095

source2.txt Quick Sort Inversions: 24975989

source2.txt Bubble Sort Inversions: 24975989

source2.txt Merge Sort Inversions: 24975989

source3.txt Quick Sort Inversions: 25106742

source3.txt Bubble Sort Inversions: 25106742

source3.txt Merge Sort Inversions: 25106742

source4.txt Quick Sort Inversions: 25202806

source4.txt Bubble Sort Inversions: 25202806

source4.txt Merge Sort Inversions: 25202806

source5.txt Quick Sort Inversions: 25016212

source5.txt Bubble Sort Inversions: 25016212

source5.txt Merge Sort Inversions: 25016212

The least number of inversion is 24975989

The file with least inversion is : source2.txt
```

<u>Conclusions:</u>

Modifying algorithms is a common practice in computer science and the choice of modifications depends on many different factors such as specific problem requirements, time complexity, and what context the algorithm will be used in. Careful consideration of these factors and through multiple testing instances are completely essential to help ensure the accuracy and efficiency of any modified algorithm. We had to make numerous changes to our code as we continued to test it until we managed to get the results that we needed. It is important to understand the source code before doing any modifications.

Because of what we accomplished here in our project, we believe that we have the potential to tackle higher level problems regarding sorting algorithms. Learning about many different sorting algorithms also gave us more insights into the principles of algorithm design such as divide and conquer strategies and best and worst case scenarios.