Implementing Selection Sort and Quick Sort Algorithms

Homework #3

By Logan Miles

1. Objectives

The goal of this assignment was to implement the selection sort algorithm and two variations of the quick sort algorithm to test their performance utilizing a high-level programming language.

2. Program Design

In order to successfully implement the required sorting algorithms, as well as test and compare their performance, several functions were implemented in addition to the actual sorting algorithms themselves. Several of these functions were modified from the last assignment to improve the testing process. These are:

1) main() – This driver function calls readFile() on each different text file.
2) readFile() – This function contains the code to read the text files, implement the sorting algorithms, calculate their execution time, and print the results.
3) selectionSort() – Function containing the selection sort algorithm.
4) printArray() – This function adds each element of the sorted array to a string and prints it for testing purposes.
5) quickSort() – Function containing the quick sort algorithm.
6) partition() – Called in quickSort(), this function this function is responsible for selecting a pivot element and partitioning the input array into two subarrays then returns the new pivot.
7) medianOfThreeQuickSort() – Function containing the median of three quick sort algorithm.

8) medianOfThree() – This function uses a series of if statements to determine the median of the lowest, middle, and highest index in the array.
9) partitionMedian() – This function is responsible for partitioning the array at the pivot value determined by medianOfThree() into two subarrays.

main()

This driver function contains a function call of the readFile() function on each text file containing a different unsorted array. There are also empty lines printed in between each call for the sake of easier reading during testing. This was done differently than the last assignment because during testing I found it tedious to have to change the file parameter for each file, so this is much easier and digestible as results.

readFile()

The code contained in this function is responsible for reading the file, adding each element in the represented array from the text file to an actual array, calling each sorting algorithm on the resulting array, then calculating the execution time for each and printing the results. The text file is read using the Scanner class. The trim function is then called on the resulting string to remove the space at the beginning. The string is then split at each space using a regex and added to an array of strings. This array is then iterated through using a for loop and each element is converted to an integer using parseInt(). Within this same for loop each integer element is then added to an array and a copy of this array, so that it can be reset after each sorting algorithm is called on it. Each sorting algorithm is then called on the array. But before the function call, the system time is recorded using system.nanoTime(). The time is then once again recorded after. The difference between the two is calculated to give the execution time represented in nanoseconds. This is then printed in addition to a float representation of the time converted to milliseconds and seconds. The file name and the array size are also printed for

understandable results. The printArray() function is also called when not commented out to verify the sorting algorithm sorted the array as expected.

printArray()

This is the final helper function not directly involved in the sorting algorithms. It simply iterates through the array using a for loop after it has been sorted and adds each element to a string. This string is then printed. This function is used only for testing purposes and is commented out in the final submission of this assignment.

selectionSort()

The selection sort algorithm first finds the minimum element in the unsorted portion of the array by comparing each element to the current minimum. The program accomplishes this using a nested for loop, in which the outer for loop initializes the minimum as the current index i. The inner for loop then iterates through the array comparing each element in the array after i to i, assigning the new minimum is an element is less than i. Once a new minimum is found, it is swapped with the first element in the unsorted portion, effectively moving it to the correct position in the unsorted portion of the array. This is accomplished programmatically by assigning i to a temp value, then assigning the new min to i, and then assigning the temp value of i to the min. This is repeated for each element in the array until each element has been moved to the correct position, sorting the array.

quicksort()

The quickSort() function calls partition() to find a pivot index and then recursively calls itself on the resulting left and right subarrays which are defined by the pivot – 1 and pivot + 1 respectively. This continues as long the lowest index is less than the highest index, essentially meaning the function will recursively call itself until the subarrays have been reduced to a size of one.

partition()

This function is responsible for selecting a pivot element and partitioning the array into two subarrays: one containing elements less than or equal to the pivot, and the other containing elements greater than the pivot. After the partition, the function returns the new pivot index. This is accomplished programmatically by iterating through the array and comparing the value of the current index to that of the highest index in the array using an if statement to check if this is true. If it is, then the value of the current index is swapped with that of the pivot. After the for loop is executed, the pivot + 1 is swapped with the highest index in the array to ensure it is also in the correct position. This pivot value is then returned as i + 1.

medianOfThreeQuickSort()

The median of three quick sort algorithm functions essentially the same as the quick sort algorithm apart from several key differences. The pivot value is not determined by partition(), instead it is the median of the lowest, middle, and highest index in the array found in the medianOfThree() function. The array is then partitioned using partitionMedian() which functions similarly to partition(), partitioning the array into two subarrays.

medianOfThree()

This function utilizes a series of if statements to determine pivot value at which partition median splits the array into the two subarrays. The function first calculates the median index by adding the lowest index to the difference of the highest and lowest index divided by two. The function then uses a series of if statements to determine which of the three indices has the median value and returns that index as the pivot value. The if statement logic first determines if the lowest index's value is higher than the highest index, causing the middle index to be the median if it is lower than the highest index, the highest to be the median if the lowest index value is greater than the middle, and the lowest to be the median if

neither. If the lowest isn't greater than the highest, then the function returns the lowest as the pivot. The lowest is also returned as the pivot if the middle value is greater than the highest, and if none else the middle index is returned as the pivot.

partitionMedian()

The partitionMedian() function partitions the array into two subarrays: one containing elements less than or equal to the pivot, and the other containing elements greater than the pivot, then returns the new pivot value. First the function swaps the temp value for the pivot and then swaps the pivot index for the highest index in the array. The function then iterates through the array from the lowest to highest index in the array and checks with an if statement if the value of the current index is less than or equal to the pivot. If this is true then the function replaces the current index value with temporary pivot value. After the for loop has completed, the value of the highest index is then swapped with the value of the pivot, essentially placing the pivot in the correct position in the array. The function then returns this pivot value.

```
You, 40 seconds ago | 3 authors (imiles1511 and others)
1  import java.io.IOException;
2  import java.nio.file.Paths;
3  import java.util.Scanner;
4
   You, 40 seconds ago | 3 authors (imiles1511 and others)
5  public class HW3 {
6
7      private static int[] originalArray;
8      private static int[] array;
9
       Run | Debug
10     public static void main(String[] args) {
11         readFile(file:"input_16.txt");
12         System.out.println();
13         readFile(file:"input_32.txt");
14         System.out.println();
15         readFile(file:"input_64.txt");
16         System.out.println();
17         readFile(file:"input_128.txt");
18         System.out.println();
19         readFile(file:"input_256.txt");
20         System.out.println();
21         readFile(file:"input_512.txt");
22         System.out.println();
23         readFile(file:"input_1024.txt");
24         System.out.println();
25         readFile(file:"input_2048.txt");
26         System.out.println();
27         readFile(file:"input_4096.txt");
28         System.out.println();
29         readFile(file:"input_8192.txt");
30         System.out.println();
31         readFile(file:"Input_Random.txt");
32         System.out.println();
33         readFile(file:"Input_ReversedSorted.txt");
34         System.out.println();
35         readFile(file:"Input_Sorted.txt");
36     }
```

main()

```java
/*
Description: this function reads the file using the file parameter, converts strings from text file to ints and adds them to an array, then runs the algorithms and prints execution time and outputs
Parameters:
String file - this string represents the file name to be read
Returns: nothing
 */
public static void readFile(String file) {
    try (Scanner reader = new Scanner(Paths.get(file))) {
        while (reader.hasNextLine()) {
        String intString = reader.nextLine();
        intString = intString.trim(); //removes extra spaces from the beginning
        String[] stringArray = intString.split(regex:"\\s"); //splits the string at each space and adds each individual string to an array
        originalArray = new int[stringArray.length]; //initializes array to size of the string array read from the file
        array = new int[stringArray.length]; //initializes a copy of the original array
        for(int i = 0; i < stringArray.length; i++) { //converts each string in the array to an int and adds to two new identical arrays
            String string = stringArray[i];
            originalArray[i] = Integer.parseInt(string);
            array[i] = Integer.parseInt(string);
        }
    }
    reader.close();

    long timeInit = System.nanoTime(); //records initial system time in nanoseconds
    selectionSort(array);
    long timeFinal = System.nanoTime(); // records final system time in nanoseconds
    long time = timeFinal - timeInit; //calculates time taken for insertion sort algorithm
    // printArray(array);
    System.out.println("File: " + file);
    System.out.println("Array Size: " + (array.length));
    System.out.println("Insertion Sort Time: " + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
    array = originalArray; //resets the array

    timeInit = System.nanoTime(); //records initial system time in nanoseconds
    quickSort(array, low:0, array.length-1);
    timeFinal = System.nanoTime(); // records final system time in nanoseconds
    time = timeFinal - timeInit; //calculates time taken for insertion sort algorithm
    // printArray(array);
    System.out.println("File: " + file);
    System.out.println("Array Size: " + (array.length));
    System.out.println("Quick Sort Time: " + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");
    array = originalArray; //resets the array

    timeInit = System.nanoTime(); //records initial system time in nanoseconds
    medianOfThreeQuickSort(array, low:0, array.length-1);
    timeFinal = System.nanoTime(); // records final system time in nanoseconds
    time = timeFinal - timeInit; //calculates time taken for insertion sort algorithm
    // printArray(array);
    System.out.println("File: " + file);
    System.out.println("Array Size: " + (array.length));
    System.out.println("Median of 3 Quick Sort Time: " + time + " nanoseconds, " + (float)time/1000000 + " milliseconds, or " + (float)time/1000000000 + " seconds");

    } catch (IOException e) {
        e.printStackTrace();
```

readFile()

```java
/*
Description: finds the minimum element in the unsorted portion of the array then swaps it with the first element in the sorted portion
Parameters:
int[] array - the array of integers to be sorted
Returns: nothing
References:
https://www.geeksforgeeks.org/java-program-for-selection-sort/
https://chat.openai.com/share/53cb093d-7b78-46ff-8701-d87c4cede31e
 */
public static void selectionSort(int[] array) {
    int size = array.length; //sets size equal to  number of elements
    for (int i = 0; i < size - 1; i++) { //iterates over the whole array
        int min = i;  //sets smallest value to current index
        for (int j = i + 1; j < size; j++) { //iterates over the unsorted portion of the array
            if (array[j] < array[min]) { //if the current iteration value is smaller than the original minimum
                min  = j; //set new minimum to new index
            }
        }
        int temp = array[i]; //creates temp index copy of the value of the current index
        array[i] = array[min]; //swaps the index of the original min to new min
        array[min] = temp; //sets the index of new min to original min
    }
}
```

selectionSort()

```java
/*
Description: called in main to print out each element in the array after it is sorted for testing purposes
Parameters:
int[] array - array that the function prints
Returns: nothing
 */
public static void printArray(int[] array) {
    String s = "";
    for (int i = 0; i < array.length; i++) {
        s = s + array[i] + " ";
    }
    System.out.println(s);
}
```

printArray()

```java
/*
Description: this function recursively calls itself on the array and the resulting subarrays, causing them to be partitioned at the pivot value each time which is determined by partition()
Parameters:
int[] array - the array to be sorted
int low - the lowest index in the array
int high - the highest index in the array
Returns: nothing
References:
https://www.geeksforgeeks.org/java-program-for-quicksort/
https://chat.openai.com/share/854d75ed-87c1-4826-bd4c-6c0f9135c9f9
 */
public static void quickSort(int[] array, int low, int high) {
    if (low < high) { //recursively call function on subarrays until they are of size one
        int pivot = partition(array, low, high);
        quickSort(array, low, pivot - 1); //calls quicksort on the left subarray
        quickSort(array, pivot + 1, high); //calls quicksort on the right subarray
    }
}
```

quickSort()

```java
/*
Description: this function is responsible for selecting a pivot element and partitioning the array into two subarrays: one containing elements less than or equal to the pivot, and the other containing elements greater than the
Parameters:
int[] array - the array to be sorted
int low - the lowest index in the array
int high - the highest index in the array
Returns:
int i + 1 - the new pivot value for the next call of partition()
References:
https://www.geeksforgeeks.org/java-program-for-quicksort/
https://chat.openai.com/share/854d75ed-87c1-4826-bd4c-6c0f9135c9f9
 */
public static int partition(int[] array, int low, int high) {
    int x = array[high]; //set x as the highest value in the array
    int i = low - 1; //set pivot to the lowest value - 1
    for (int j = low; j < high; j++) { //iterate from the lowest index of the array to the highest
        if (array[j] <= x) { //if the value of the current index is less than or equal to the value of the highest index
            i++; //forward iterate the pivot value by 1
            int temp = array[i]; //next few lines swap the current index and the pivot
            array[i] = array[j];
            array[j] = temp;
        }
    }
    int temp = array[i + 1]; //next few lines swap the highest index with the pivot + 1
    array[i + 1] = array[high];
    array[high] = temp;
    return i + 1; //return new pivot
}
```

partition()

```java
/*
Description: this function is essentially the same as quickSort(), except it uses the medianOfThree() function to determine the pivot value, then calls partitionMedian() on that value
Parameters:
int[] array - the array to be sorted
int low - the lowest index in the array
int high - the highest index in the array
Returns: nothing
References:
https://stackoverflow.com/questions/27284619/selecting-the-pivot-from-the-median-of-three-items-quicksort-java
https://chat.openai.com/share/854d75ed-87c1-4826-bd4c-6c0f9135c9f9
 */
public static void medianOfThreeQuickSort(int[] array, int low, int high) {
    if (low < high) { //recursively call function on subarrays until they are of size one
        int pivot = medianOfThree(array, low, high);
        int partitionIndex = partitionMedian(array, low, high, pivot); //partition at the pivot value
        quickSort(array, low, partitionIndex - 1); //recursively call on left subarray
        quickSort(array, partitionIndex + 1, high); //recursively call on right subarray
    }
}
```

medianOfThreeQuickSort()

```java
    /*
    Description: this function determines the median of the lowest, middle, and highest indices using a series of if statements and returns that value as the pivot
    Parameters:
    int[] array - the array to be sorted
    int low - the lowest index in the array
    int high - the highest index in the array
    Returns:
    int low, int mid, or int high depending on the if conditions which all function as the pivot value
    References:
    https://stackoverflow.com/questions/27284619/selecting-the-pivot-from-the-median-of-three-items-quicksort-java
    https://chat.openai.com/share/854d75ed-87c1-4826-bd4c-6c0f9135c9f9
     */
    public static int medianOfThree(int[] array, int low, int high) {
        int mid = low + (high - low) / 2; //calculate middle index
        if (array[low] > array[high]) { //if the value of the lowest index is greater than the value of the highest index
            if (array[mid] > array[high]) { //if the value of the middle index is greater than the value of the highest index
                return mid;
            }
            else if (array[low] > array[mid]){ //if the value of the lowest index is greater than the value of the middle index
                return high;
            }
            else {
                return low;
            }
        }
        else {
            if (array[low] > array[high]) { //if the value of the lowest index is greater than the value of the highest index
                return low;
            }
            else if (array[mid] > array[high]){ //if the value of the middle index is greater than the value of the highest index
                return low;
            }
            else {
                return mid;
            }
        }
    }
```

medianOfThree()

```java
    /*
    Description: this function is responsible for partitioning the array into two subarrays at the pivot: one containing elements less than or equal to the pivot, and the other containing elements greater than the pivot, then retur
    Parameters:
    int[] array - the array to be sorted
    int low - the lowest index in the array
    int high - the highest index in the array
    int pivot - the value at which the array is partitioned
    Returns:
    int i + 1 - the new pivot value for the next call of partition()
    References:
    https://stackoverflow.com/questions/27284619/selecting-the-pivot-from-the-median-of-three-items-quicksort-java
    https://chat.openai.com/share/854d75ed-87c1-4826-bd4c-6c0f9135c9f9
     */
    public static int partitionMedian(int[] array, int low, int high, int pivot) {
        int x = array[pivot]; //set x to value of pivot
        int temp = array[pivot]; //next few lines swap the pivot index and the highest index
        array[pivot] = array[high];
        array[high] = temp;
        int i = low - 1;
        for (int j = low; j < high; j++) { //iterate from lowest to highest index in array
            if (array[j] <= x) { //if the value of the current index is less than the value of the pivot
                i++; //iterate lowest index forward 1
                temp = array[i]; //next few lines swap the current index and the pivot
                array[i] = array[j];
                array[j] = temp;
            }
        }
        temp = array[i + 1]; //next few lines swap the highest index in the array with the pivot
        array[i + 1] = array[high];
        array[high] = temp;
        return i + 1;
    }
}
```

partitionMedian()

3. Testing

Testing of the sorting algorithms and their supporting functions was done using several text files containing integers separated by spaces ranging from a size of 16 integers to 8192 integers. There are also three special case text files for testing as well. These are:

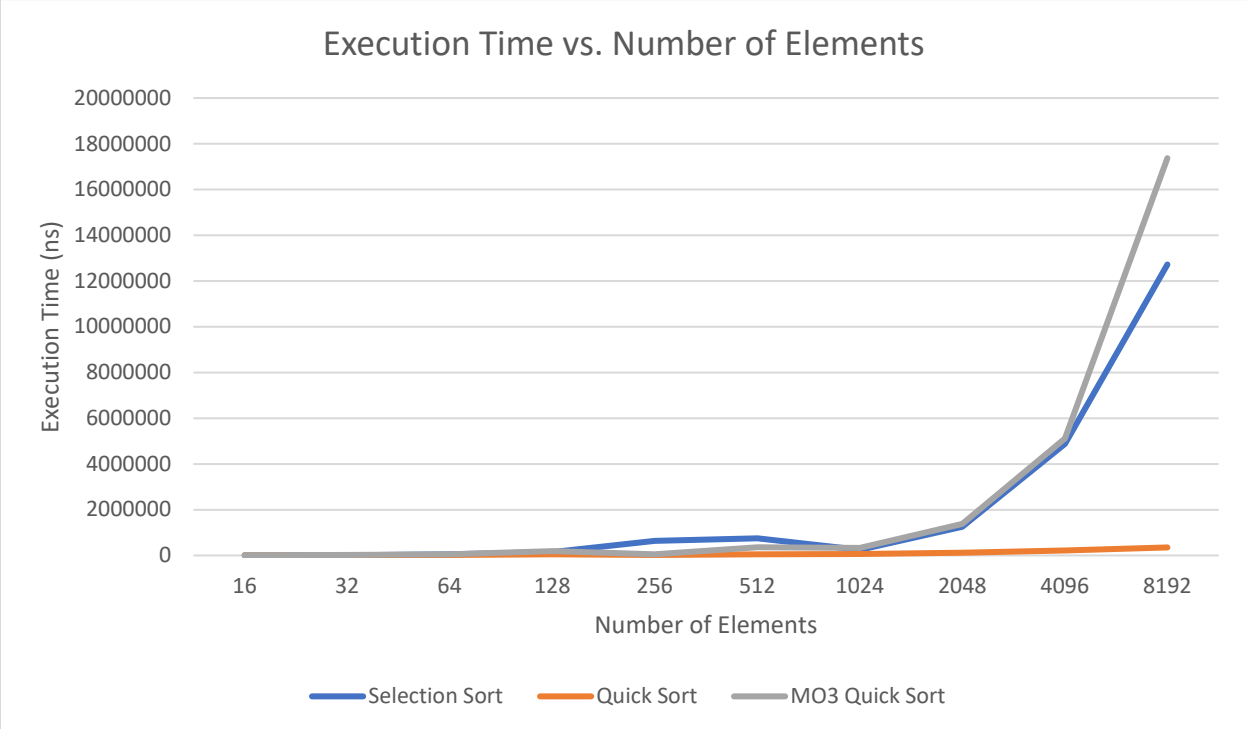
1) Input_Random.txt – Contains random integers.
2) Input_Random.txt – Contains integers in reversed sorted order.
3) Input_Sorted.txt – Contains integers sorted in ascending order.

Each different text file is called in readFile(), and the print statements in main() provide the file name, execution time, and array size.

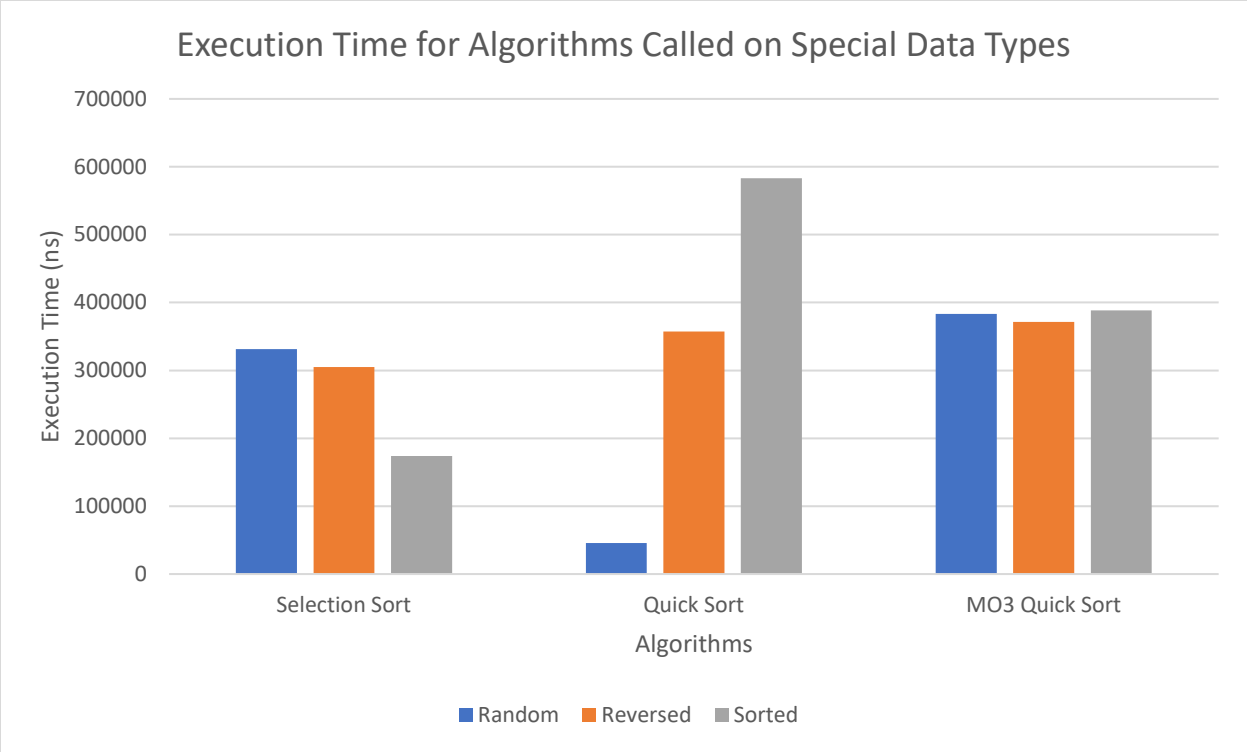
```
\CS-303-HW3>  c:; cd 'c:\Users\logan\OneDrive - UAB - The University of Alabama at Birmingham\UAB File
s\FA 2023\CS-303\HW\CS-303-HW3'; & 'C:\Program Files\Java\jdk-19\bin\java.exe' '-XX:+ShowCodeDetailsIn
ExceptionMessages' '-cp' 'C:\Users\logan\OneDrive - UAB - The University of Alabama at Birmingham\UAB
Files\FA 2023\CS-303\HW\CS-303-HW3\bin' 'HW3'
File: input_16.txt
Array Size: 16
Selection Sort Time: 4500 nanoseconds, 0.0045 milliseconds, or 4.5E-6 seconds
File: input_16.txt
Array Size: 16
Quick Sort Time: 9199 nanoseconds, 0.009199 milliseconds, or 9.199E-6 seconds
File: input_16.txt
Array Size: 16
Median of 3 Quick Sort Time: 10400 nanoseconds, 0.0104 milliseconds, or 1.04E-5 seconds

File: input_32.txt
Array Size: 32
Selection Sort Time: 10700 nanoseconds, 0.0107 milliseconds, or 1.07E-5 seconds
File: input_32.txt
Array Size: 32
Quick Sort Time: 6600 nanoseconds, 0.0066 milliseconds, or 6.6E-6 seconds
File: input_32.txt
Array Size: 32
Median of 3 Quick Sort Time: 14099 nanoseconds, 0.014099 milliseconds, or 1.4099E-5 seconds

File: input_64.txt
Array Size: 64
Selection Sort Time: 35300 nanoseconds, 0.0353 milliseconds, or 3.53E-5 seconds
File: input_64.txt
Array Size: 64
Quick Sort Time: 14300 nanoseconds, 0.0143 milliseconds, or 1.43E-5 seconds
File: input_64.txt
Array Size: 64
Median of 3 Quick Sort Time: 65299 nanoseconds, 0.065299 milliseconds, or 6.5299E-5 seconds

File: input_128.txt
Array Size: 128
Selection Sort Time: 133900 nanoseconds, 0.1339 milliseconds, or 1.339E-4 seconds
File: input_128.txt
Array Size: 128
Quick Sort Time: 48401 nanoseconds, 0.048401 milliseconds, or 4.8401E-5 seconds
File: input_128.txt
Array Size: 128
Median of 3 Quick Sort Time: 134699 nanoseconds, 0.134699 milliseconds, or 1.34699E-4 seconds

File: input_256.txt
Array Size: 256
Selection Sort Time: 466001 nanoseconds, 0.466001 milliseconds, or 4.66001E-4 seconds
File: input_256.txt
Array Size: 256
Quick Sort Time: 20800 nanoseconds, 0.0208 milliseconds, or 2.08E-5 seconds
File: input_256.txt
Array Size: 256
Median of 3 Quick Sort Time: 50900 nanoseconds, 0.0509 milliseconds, or 5.09E-5 seconds
```

```
File: input_512.txt
Array Size: 512
Selection Sort Time: 752200 nanoseconds, 0.7522 milliseconds, or 7.522E-4 seconds
File: input_512.txt
Array Size: 512
Quick Sort Time: 31300 nanoseconds, 0.0313 milliseconds, or 3.13E-5 seconds
File: input_512.txt
Array Size: 512
Median of 3 Quick Sort Time: 216799 nanoseconds, 0.216799 milliseconds, or 2.16799E-4 seconds

File: input_1024.txt
Array Size: 1024
Selection Sort Time: 314601 nanoseconds, 0.314601 milliseconds, or 3.14601E-4 seconds
File: input_1024.txt
Array Size: 1024
Quick Sort Time: 66300 nanoseconds, 0.0663 milliseconds, or 6.63E-5 seconds
File: input_1024.txt
Array Size: 1024
Median of 3 Quick Sort Time: 234700 nanoseconds, 0.2347 milliseconds, or 2.347E-4 seconds

File: input_2048.txt
Array Size: 2048
Selection Sort Time: 877200 nanoseconds, 0.8772 milliseconds, or 8.772E-4 seconds
File: input_2048.txt
Array Size: 2048
Quick Sort Time: 92800 nanoseconds, 0.0928 milliseconds, or 9.28E-5 seconds
File: input_2048.txt
Array Size: 2048
Median of 3 Quick Sort Time: 818000 nanoseconds, 0.818 milliseconds, or 8.18E-4 seconds

File: input_4096.txt
Array Size: 4096
Selection Sort Time: 3001399 nanoseconds, 3.001399 milliseconds, or 0.003001399 seconds
File: input_4096.txt
Array Size: 4096
Quick Sort Time: 196300 nanoseconds, 0.1963 milliseconds, or 1.963E-4 seconds
File: input_4096.txt
Array Size: 4096
Median of 3 Quick Sort Time: 5063500 nanoseconds, 5.0635 milliseconds, or 0.0050635 seconds

File: input_8192.txt
Array Size: 8192
Selection Sort Time: 13060399 nanoseconds, 13.060399 milliseconds, or 0.013060399 seconds
File: input_8192.txt
Array Size: 8192
Quick Sort Time: 349600 nanoseconds, 0.3496 milliseconds, or 3.496E-4 seconds
File: input_8192.txt
Array Size: 8192
Median of 3 Quick Sort Time: 18622700 nanoseconds, 18.6227 milliseconds, or 0.0186227 seconds
```

```
File: Input_Random.txt
Array Size: 1024
Selection Sort Time: 200900 nanoseconds, 0.2009 milliseconds, or 2.009E-4 seconds
File: Input_Random.txt
Array Size: 1024
Quick Sort Time: 47500 nanoseconds, 0.0475 milliseconds, or 4.75E-5 seconds
File: Input_Random.txt
Array Size: 1024
Median of 3 Quick Sort Time: 367500 nanoseconds, 0.3675 milliseconds, or 3.675E-4 seconds

File: Input_ReversedSorted.txt
Array Size: 1024
Selection Sort Time: 495300 nanoseconds, 0.4953 milliseconds, or 4.953E-4 seconds
File: Input_ReversedSorted.txt
Array Size: 1024
Quick Sort Time: 355899 nanoseconds, 0.355899 milliseconds, or 3.55899E-4 seconds
File: Input_ReversedSorted.txt
Array Size: 1024
Median of 3 Quick Sort Time: 357299 nanoseconds, 0.357299 milliseconds, or 3.57299E-4 seconds

File: Input_Sorted.txt
Array Size: 1024
Selection Sort Time: 221200 nanoseconds, 0.2212 milliseconds, or 2.212E-4 seconds
File: Input_Sorted.txt
Array Size: 1024
Quick Sort Time: 689101 nanoseconds, 0.689101 milliseconds, or 6.89101E-4 seconds
File: Input_Sorted.txt
Array Size: 1024
Median of 3 Quick Sort Time: 330200 nanoseconds, 0.3302 milliseconds, or 3.302E-4 seconds
PS C:\Users\logan\OneDrive - UAB - The University of Alabama at Birmingham\UAB Files\FA 2023\CS-303\HW
\CS-303-HW3>
```



Based on the representation of the execution time versus the number of elements per sorting algorithm shown in the above graph, there is a distinct difference between each of the three algorithms. The execution time for selection sort suffered from a noticeable change in execution time after the number of elements increased from 1024 to 2048 and then increased approximately quadratically with each doubling of the number of elements. This quadratic increase is present throughout each increase in the number of elements; however, it is much more noticeable the greater the number of elements increases. For example, a difference of 245200 nanoseconds to 1257601 nanoseconds is much more significant than an increase from 5799 nanoseconds to 11700 nanoseconds. Overall, the time complexity of selection sort is $O(n^2)$. This is because the algorithm utilizes a nested for loop. The median of three quick sort algorithm also suffered from an increase in execution time after an increase in elements. The worst-case time complexity for quick sort is $O(n^2)$ and the best-case scenario is O(nlog(n)). This algorithm avoids the worst case time-complexity by choosing the median of the first, middle, and last elements, however this also sacrifices the possibility of achieving the best-case time-complexity. The quick sort algorithm, however, did not have a dramatic increase in execution time with an increase in the number of elements. Unlike the selection sort algorithm and median of three quick sort algorithms it only increased by approximately nlog(2n) with each array size increase. This means that quick sort has a time complexity of O(nlog(n)). Comparing each of the sorting algorithms, there doesn't seem to be much of a difference between their execution when the number of elements is lower, but as the number of elements increases, this difference becomes more apparent. This is because a quadratic increase in time is much greater than a logarithmic increase in time.



The graph above compares three specific scenarios for each of the sorting algorithms. Each array has 1024 elements and represents one of the following:

1) The array consists of randomized integers ranging from 1 to 1024.
2) The array consists of integers sorted in descending order.
3) The array consists of integers sorted in ascending order.

The difference between the way each of these algorithms handles these scenarios is shown by the difference in execution time taken for each. The selection sort algorithm handled the random and reversed arrays with little difference in execution time, however the sorted array was operated on in approximately half the time. This is because the algorithm does not have to swap any of the elements in the array resulting in a best-case time complexity of O(n). The quick sort algorithm achieved its own best-case time complexity of O(nlog(n)) in the randomly sorted array, but also sunk to its worst-case scenario of $O(n^2)$ when the array is already sorted. The best-case time complexity is a result of even array

partitioning when choosing the pivot in partition(), while the worst-case time complexity is a result of uneven array partitioning due to the fact that the array was already sorted causing the pivot to be chosen as the first element. The reason that the time complexity of the algorithm did not suffer when the array was reverse sorted is because the first element needed to be in the last position, causing more even partitioning. The median of three quick sort algorithm avoided this worst-case time complexity by choosing the pivot based of the median of the first, middle, and last elements, but at the cost of also not achieving the algorithms best-case time complexity.

4. References

https://www.geeksforgeeks.org/java-program-for-selection-sort/

https://chat.openai.com/share/53cb093d-7b78-46ff-8701-d87c4cede31e

https://www.geeksforgeeks.org/java-program-for-quicksort/

https://chat.openai.com/share/854d75ed-87c1-4826-bd4c-6c0f9135c9f9

https://stackoverflow.com/questions/27284619/selecting-the-pivot-from-the-median-of-three-items-quicksort-java