CS303 Lab 5 – Quick Sort

Problem Specification:

The goal of this assignment was to create and test the quick sort algorithm. We read in 7 different files with random numbers ranging 1 – n, n being the number of random numbers in the files. The 7 files had the following number random numbers: 100, 1000, 5000, 10000, 50000, 100000, 500000. I expected this to sort to be close to as fast as insertion sort for smaller arrays and to be close to or faster to merge sort since both use the divide and conquer technique. Considering the algorithms for heap and quick sort, it can be said that quick sort should be faster than heap for most cases.

Program Design:

The quick sort algorithm has two functions, one called sort() and the other called Partition(). The sort() function takes in an int[] array, int p, and int r. The int[] array is the array that needs to be sorted, int p is the starting point of the array that needs to be sorted, int r is the ending point of the array that needs to be sorted. The Partition() function takes and compares all of the numbers to the pivot and swapping when necessary. There is also a median() function that is implemented later that takes three values, based on the pseudocode, which is the first, middle, and last elements in the array and finds the median of the three to get the pivot.

Within the AllSortDriver, which is the file with the driver that tests all of the sorts, I created two functions. The updateArray() function, is a function that takes in a file, reads it and assigns all the numbers in the txt file and puts it into arrays and returns the size of the arrays so that the next function, getTimes(), can use it. The getTimes() function calls the sorting algorithm on the array and times it and uses the size of the array to print out a statement with the array size and time it took to sort.

Testing Plan:

The testing plan is the same as the previous labs. I read each of the files into string arrays and then copy those arrays into integer arrays for sorting. I did change up my code this time because I realized that after sorting once all the other sorts would be sorting a sorted array. This time I added an array for each sort, so this ensures that each sort gets its own unsorted array. I also added an extra array because I created a new file for the median of 3 version of quick sort.

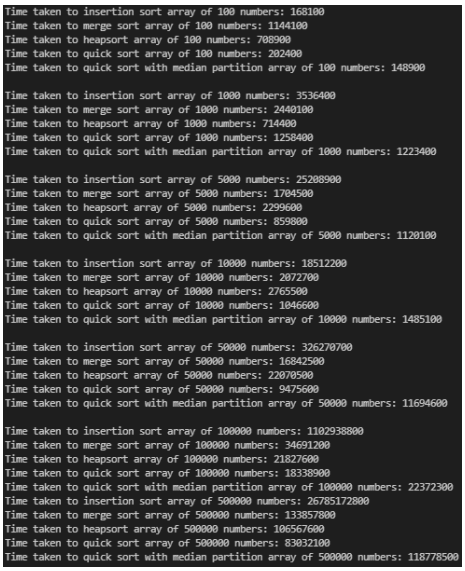
There was also another driver that only tests the quick sort and the quick sort with median of 3 implemented. It tests 100 – 500000 and the three extra tests of random values, reverse sorted, and sorted arrays. This was implemented the same way as the all sort driver.

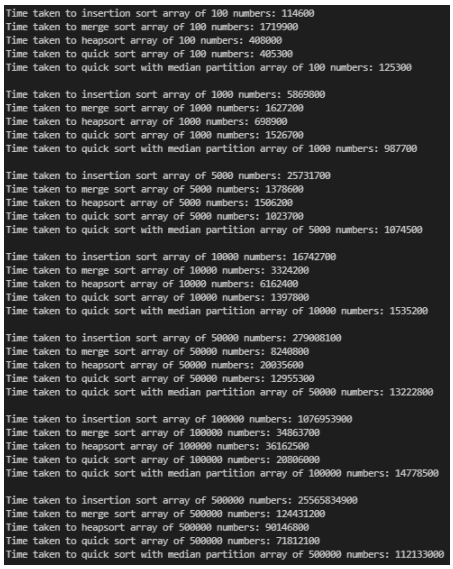
Test Cases:

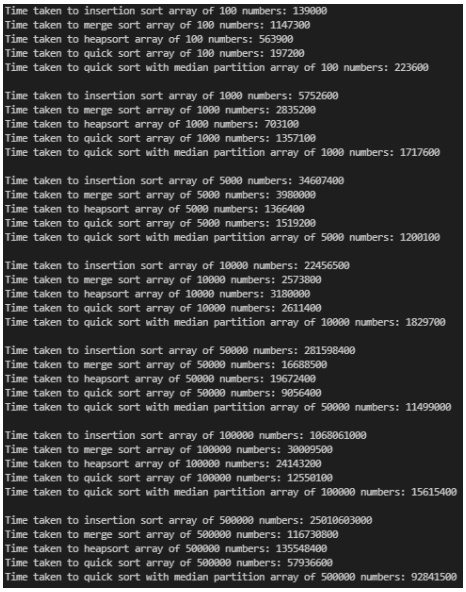
Array Size	Quick Sort w/ Median Time(ns)	Quick Sort Time(ns)	Heap Sort Time(ns)	Merge Sort Time(ns)	Insertion Sort Time(ns)
100	126600	98500	281100	605900	59900
1000	2139400	741800	295600	1652000	4442400
5000	1075600	883900	1344600	2208500	19231300
10000	1099000	1157100	2500800	1742000	15074200
50000	6143200	4747900	11613400	9038700	188761500
100000	1150100	7776900	15882900	15710000	740171900
500000	55509600	48242799	60013300	68584500	16990845200

Array Type	Quick Sort Time(ns)
Random Numbers	902500
Reverse Sorted	3309400
Sorted	2831300

This is the test case that is mainly being used for analysis, the rest below are just showing example outputs.







Analysis/Conclusion:

Based on the new driver class and all of the timings that came from the testing, the results are as expected. Each sort takes longer as the number of elements increases. The insertion sort takes the longest while the quick sort seems to be fastest and most efficient by the time you test 500,000 elements. Comparing the regular quick sort and the quick sort that uses median of 3, the results are as expected, sometimes the regular version does better and other times the median of 3 does better. Regarding the random numbers, reversed sorted, and sorted files, it does not seem as though it made a difference. The reversed sorted did take longer than the sorted but it did not so much longer than expected.

All files such as input files, insertion sort, merge sort, and heap sort came from the previous labs and have been copied and pasted for testing and comparing purposes.

References:

I did not use any outside resources; all the code came from my own creation. The lab report is based off my previous reports causing similarities between the two. I am citing my own previous work in order to avoid self-plagiarism.

All input files were provided to us with the lab instructions.

Below are the pictures of my code.

```
private static int[] iArray = new int[]{};
private static int[] mArray = new int[]{};
private static int[] hArray = new int[]{};
private static int[] qsArray = new int[]{};
private static int[] qmArray = new int[]{};
private static int[] tempArray = new int[]{};

private static InsertionSort is = new InsertionSort();
private static Merge ms = new Merge();
private static HeapSort hp = new HeapSort();
private static QuickSort qs = new QuickSort();
private static QuicksortM qm = new QuicksortM();
```

```
public static int updateArray(File f)
{
    try
    {
        Scanner sc = new Scanner(f);
        String[] sArray = sc.nextLine().trim().split(" ");

        iArray = new int[sArray.length];
        mArray = new int[sArray.length];
        hArray = new int[sArray.length];
        qsArray = new int[sArray.length];
        qmArray = new int[sArray.length];

        int sLength = sArray.length;
        for(int i = 0; i < sLength; i++)
        {
            iArray[i] = Integer.parseInt(sArray[i]);
            mArray[i] = Integer.parseInt(sArray[i]);
            hArray[i] = Integer.parseInt(sArray[i]);
            qsArray[i] = Integer.parseInt(sArray[i]);
            qmArray[i] = Integer.parseInt(sArray[i]);
        }
        sc.close();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }

    tempArray = new int[mArray.length];
    for(int x = 0; x < mArray.length; x++)
    {
        tempArray[x] = mArray[x];
    }

    return tempArray.length;
}
```

```
private static QuickSort qs = new QuickSort();
private static QuicksortM qm = new QuicksortM();

private static int[] qsArray = new int[]{};
private static int[] qmArray = new int[]{};

private static int updateArray(File f)
{
    try
    {
        Scanner sc = new Scanner(f);
        String[] sArray = sc.nextLine().trim().split(" ");
        qsArray = new int[sArray.length];
        qmArray = new int[sArray.length];
        int sLength = sArray.length;
        for(int i = 0; i < sLength; i++)
        {
            qsArray[i] = Integer.parseInt(sArray[i]);
            qmArray[i] = Integer.parseInt(sArray[i]);
        }
        sc.close();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }

    return qsArray.length;
}

private static void getTimes(int size)
{
    long sTime = System.nanoTime();
    qs.sort(qsArray, 0, qsArray.length-1);
    long eTime = System.nanoTime() - sTime;
    System.out.println("Time taken to quick sort array of " + size + " numbers: " + eTime);

    long mTime = System.nanoTime();
    qm.sort(qmArray, 0, qmArray.length-1);
    long emTime = System.nanoTime() - mTime;
    System.out.println("Time taken to quick sort with median partition array of " + size + " numbers: " + emTime + "\n");
}
```

```
public static void getTimes(int size)
{
    long iTime = System.nanoTime();
    is.iSort(iArray);
    long eiTime = System.nanoTime() - iTime;
    System.out.println("Time taken to insertion sort array of " + size + " numbers: " + eiTime);

    long mTime = System.nanoTime();
    ms.mergeSort(mArray, tempArray, 0, mArray.length-1);
    long emTime = System.nanoTime() - mTime;
    System.out.println("Time taken to merge sort array of " + size + " numbers: " + emTime);

    long hTime = System.nanoTime();
    hp.sort(hArray);
    long ehTime = System.nanoTime() - hTime;
    System.out.println("Time taken to heapsort array of " + size + " numbers: " + ehTime);

    long sTime = System.nanoTime();
    qs.sort(qsArray, 0, qsArray.length-1);
    long eTime = System.nanoTime() - sTime;
    System.out.println("Time taken to quick sort array of " + size + " numbers: " + eTime);

    long qmTime = System.nanoTime();
    qm.sort(qmArray, 0, qmArray.length-1);
    long eqmTime = System.nanoTime() - qmTime;
    System.out.println("Time taken to quick sort with median partition array of " + size + " numbers: " + eqmTime + "\n");
}
```

```java
public static void main(String[] args)
{
    int size;

    File f1 = new File("input_100.txt");
    size = updateArray(f1);
    getTimes(size);

    File f2 = new File("input_1000.txt");
    size = updateArray(f2);
    getTimes(size);

    File f3 = new File("input_5000.txt");
    size = updateArray(f3);
    getTimes(size);

    File f4 = new File("input_10000.txt");
    size = updateArray(f4);
    getTimes(size);

    File f5 = new File("input_50000.txt");
    size = updateArray(f5);
    getTimes(size);

    File f6 = new File("input_100000.txt");
    size = updateArray(f6);
    getTimes(size);

    File f7 = new File("input_500000.txt");
    size = updateArray(f7);
    getTimes(size);
}
```

```java
public static void main(String[] args)
{
    File f = new File("Input_Random.txt");
    File f2 = new File("Input_ReversedSorted.txt");
    File f3 = new File("Input_Sorted.txt");

    int[] random = new int[]{};
    int[] reversed = new int[]{};
    int[] sorted = new int[]{};

    try
    {
        Scanner sc = new Scanner(f);
        String[] sArray = sc.nextLine().trim().split(" ");
        random = new int[sArray.length];
        int sLength = sArray.length;
        for(int i = 0; i < sLength; i++)
        {
            random[i] = Integer.valueOf(sArray[i]);
        }
        sc.close();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }

    try
    {
        Scanner sc = new Scanner(f2);
        String[] sArray = sc.nextLine().trim().split(" ");
        reversed = new int[sArray.length];
        int sLength = sArray.length;
        for(int i = 0; i < sLength; i++)
        {
            reversed[i] = Integer.valueOf(sArray[i]);
        }
        sc.close();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }
```

```java
    try
    {
        Scanner sc = new Scanner(f3);
        String[] sArray = sc.nextLine().trim().split(" ");
        sorted = new int[sArray.length];
        int sLength = sArray.length;
        for(int i = 0; i < sLength; i++)
        {
            sorted[i] = Integer.valueOf(sArray[i]);
        }
        sc.close();
    }
    catch(IOException e)
    {
        e.printStackTrace();
    }

    long rdTime = System.nanoTime();
    qs.sort(random, 0, random.length-1);
    long erdTime = System.nanoTime() - rdTime;
    System.out.println("Time taken to quick sort array of random numbers: " + erdTime);

    long rsTime = System.nanoTime();
    qs.sort(reversed, 0, reversed.length-1);
    long ersTime = System.nanoTime() - rsTime;
    System.out.println("Time taken to quick sort array of reversed sorted numbers: " + ersTime);

    long sTime = System.nanoTime();
    qs.sort(sorted, 0, sorted.length-1);
    long esTime = System.nanoTime() - sTime;
    System.out.println("Time taken to quick sort array of sorted numbers: " + esTime + "\n");
```

```java
public class QuickSort
{
    public void sort(int[] A, int p, int r)
    {
        if(p < r)
        {
            int q = Partition(A, p, r);
            sort(A, p, q-1);
            sort(A, q+1, r);
        }
    }

    public int Partition(int[] A, int p, int r)
    {
        int x = A[r];
        int i = p - 1;

        for(int j = p; j <= r - 1; j++)
        {
            if(A[j] <= x)
            {
                i++;
                int temp = A[i];
                A[i] = A[j];
                A[j] = temp;
            }
        }

        int temp = A[i + 1];
        A[i + 1] = A[r];
        A[r] = temp;

        return i + 1;
    }
}
```

```java
public class QuicksortM
{
    public void sort(int[] A, int p, int r)
    {
        if(p < r)
        {
            int N = r - p + 1;
            int m = median(A, p, p + N/2, r);

            int temp = A[m];
            A[m] = A[r];
            A[r] = temp;

            int q = Partition(A, p, r);
            sort(A, p, q-1);
            sort(A, q+1, r);
        }
    }

    public static int median(int[] A, int i, int j, int k)
    {

        if((A[i] <= A[j] && A[j] <= A[k]) || (A[k] <= A[j] && A[j] <= A[i]))
        {
            return j;
        }
        else if((A[j] <= A[i] && A[i] <= A[k]) || (A[k] <= A[i] && A[i] <= A[j]))
        {
            return i;
        }
        else if((A[j] <= A[k] && A[k] <= A[i]) || (A[i] <= A[k] && A[k] <= A[j]))
        {
            return k;
        }
        else
        {
            return -1;
        }

    }

    public int Partition(int[] A, int p, int r)
    {
        int x = A[r];
        int i = p - 1;

        for(int j = p; j <= r - 1; j++)
        {
            if(A[j] <= x)
            {
                i++;
                int temp = A[i];
                A[i] = A[j];
                A[j] = temp;
            }
        }

        int temp = A[i + 1];
        A[i + 1] = A[r];
        A[r] = temp;

        return i + 1;
    }
}
```