Son Huynh
ITCS 2215-003
November 2, 2015

Report

Implementation – Pseudo-code:

Random Array Generator

Either in main or its own class, using a for loop.

```
for (x = 0 to array.length)
{
        Array[x] = randomGenerator(to 99999);
        outputFile.println(Array[x]);
}
```

Bubble Sort

Start from the first element, compare with the right element. Swap the elements if the right is smaller. Repeat until sorted.

Note: when Bubble Sort reaches 500,000 (the max size I'm using), it takes around 10 minutes for it to complete. Once that's completed, then the program will run the remaining sorts.

```
BubbleSort(A[n])
for (x = 0 to array.length)
{
        for (y = 0 to length – 1)
        {
                if (A[y] > A[y + 1])
                        swap(A[y],A[j+1]);
        }
}
```

Insertion Sort

Look at the first two elements. Look at the second element: if it's less than the first, put it before the first element, if it's greater than the first, leave it where it is. Now look at the first three elements, use the 3rd element to compare and put it in the correct index. Continue until sorted.

```
InsertionSort(A[n])
for (x = 2 to array.length)
{
        tempVar = A[x];
        counter = x – 1;
        while (counter >= 0 and A[counter] > tempVar)
        {
                A[counter + 1] = A[counter]
                counter--;
        }
        A[counter + 1] = tempVar;
}
```

Merge Sort

Look at first 2 elements of each array and compare them. The smaller element goes first into merged array. Now look at the 2nd element of the 1st array and compare again. Whenever an element is moved into the merged array, look at the next element of the array it’s from.

```
Merge(A[n], B[n])
{
        x = y = z = 0;
        while (x <= n and y < m)
        {
                if (A[x] <= B[y])
                {
                        M[z] = A[x];
                        x++;
                }
                else
                {
                        M[z] = B[y];
                        y++;
                }
                z++;
        }
}

Split(C[n])
{
        Array1[C.length / 2];
        Array2[C.length – C.length/2];

        Split(Array1[n]);
        Split(Array2[n]);

        Merge(array1, array2);
}
```

Quick Sort

Given an arbitrary element of an array A, as the pivot, partition the array into 2 arrays such that one is bigger than the pivot and the other one is smaller.

```
QuickSort(A[n], start, end)
{
    if (A.length <= 1)
        return A;

    pivot = A[0];
    r = start + 1;
    l = end;

    for (k = 0 to A.length)
    {
        if (A[k] < pivot)
            r++;
        else
        {
            swap(r,l);
            l--;
        }
    }
    swap(start,pivot);

    QuickSort(A[0 to m-1]);
    QuickSort(A[m+1 to n-1]);
}
```

Time Evaluation and Efficiency Measure

Time Evaluation

Starting at 50, each subsequent input size was the previous size multiplied by 10. Thus, my different input sizes were 50, 500, 5000, 50000, and 500000. For each of the sorting algorithms used (bubble, insertion, merge, and quick sort), as the input size grows, so did the total runtime it takes to complete each iteration.

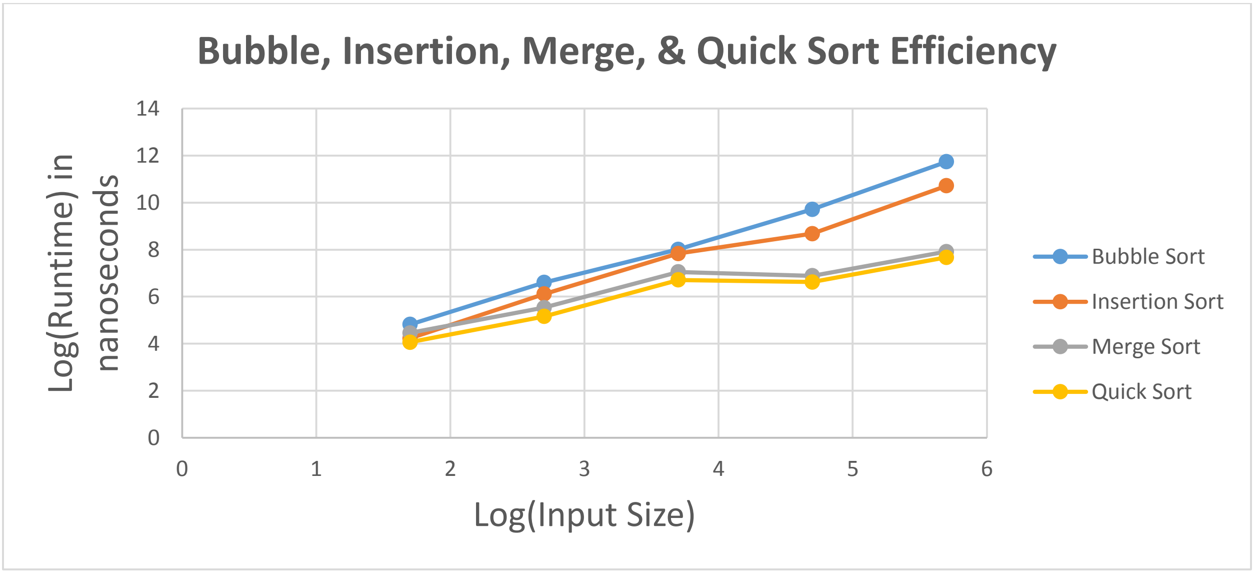
Son Huynh
ITCS 2215-003
November 2, 2015

Efficiency Measure

To visualize the efficiency of each sorting algorithm, I used Excel to plot the input size vs. runtime and the result can be seen below:

Combined Chart Data Points

	┌-----------------------------	*Log(Runtime)*	------------------------------┐	
Log(Input Size)	**Bubble Sort**	**Insertion Sort**	**Merge Sort**	**Quick Sort**
1.698970004	4.812933401	4.22214397	4.4571701	4.06243155
2.698970004	6.597326976	6.11339312	5.529267335	5.15613404
3.698970004	8.008149843	7.834937162	7.049127393	6.71034546
4.698970004	9.716415748	8.676042147	6.889704709	6.62382197
5.698970004	11.73540915	10.7133728	7.917321019	7.67128258



As seen above, as the input size increases, so did the runtime of each sorting algorithm. As expected, Bubble Sort took the most time to sort at each input size vs. the other, since its complexity is O(n^2). Similarly, because Insertion Sort is also O(n^2), we see that its line closely resembles that of the Bubble Sort. However, it was still more efficient because once the last element is inserted, algorithm guarantees that array is sorted while Bubble Sort has to go through for each element.

Merge Sort runs much faster than the other two sorting algorithms, with significant differences at the larger input sizes. This corresponds with what we discussed in class with how Merge Sort is the fastest out of the three here, with its complexity being O(n log(n)). However, despite having the same complexity of O(n log(n)), Quick Sort is still faster than Merge Sort because of its in-place characteristics, meaning it doesn't need to create a temporary array for storage to perform sorting, which Merge Sort has.

Son Huynh
ITCS 2215-003
November 2, 2015

Conclusion

This program was designed to show the efficiency of each sorting algorithms. Quick Sort emerged as the fastest out of the four, which was not surprising since it matches with what we talked about in class. The data, along with its graph, showcase the inferiority of Bubble Sort and the superiority that Quick Sort. There was no error in this program as everything runs as expected.