Joao Paulo dos Santos Ferreira
CSIT 212_02 – Data Structures and Algorithms
Instructor: Boxiang Dong
September 19, 2018

Homework 1 – Insertion and Merge Sort

1 Problem Description

Instructions. You are provided the skeleton code named *Sort.java*. The source file is available on Canvas in a folder namedHW1. Please modify the skeleton code to solve the following tasks.

- Task 1 (40 pts). Implement the *Insertion Sort* algorithm as discussed in Lecture 1. (Hint: use the function *checked_sorted* to check if your output is indeed sorted.)
- Task 2 (40 pts). Implement the *Merge Sort* algorithm as discussed in Lecture 2. (Hint: use the function *checked_sorted* to check if your output is indeed sorted.)
- Task 3 (20 pts). Generate a report to discuss the time performance of the two algorithms. Compare it with their theoretical time complexity as discussed in the lecture. Plots and figures are encouraged to help draw the conclusion.

Task 1 and 2 (source code):

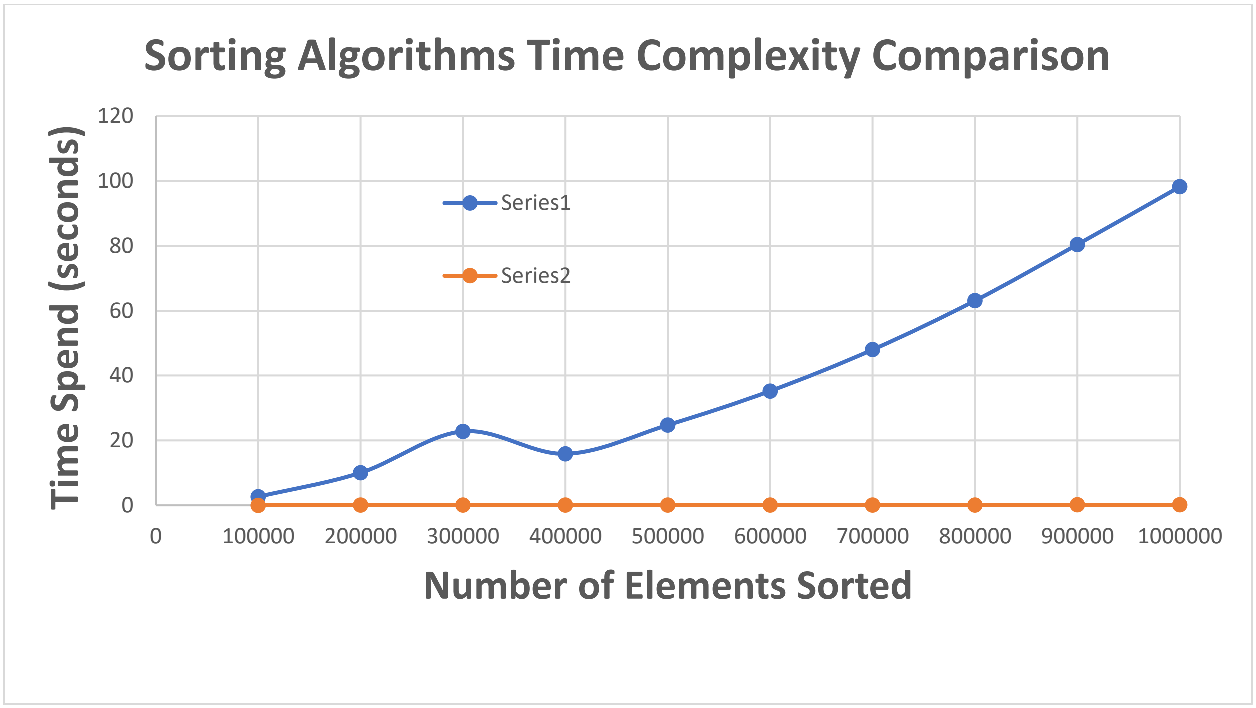
```
// CSIT 212 HW1 - Topic: Insertion and Merge Sort
// This program sorts large arrays using the insertion
// and merge sort algorithm to compare their time performance.

package sorting;
import java.util.*;

public class Sort {
      public static int[] insertion_sort (int[] array) {

             //
             for (int j = 1; j < array.length; j++)
            {
                   // stores the number that is being sorted
               int key = array[j];
               int i = j;

               // Shift greater values than the
               // number being sorted to the right
               while (i > 0 && key<(array[i-1]))
               {
                  array[i] = array[i-1];
                  i--;
               }

               // puts the number into its sorted position
               array[i] = key;
            }
```

```
		return array;
	}

	public static int[] merge_sort (int[] array, int p, int r) {
		//divide array until it contains sub-arrays  with only one element.
		if (p < r) {
			int q = ((p+r)/2);
			merge_sort(array,p,q);
			merge_sort(array,q+1,r);
			merge(array, p, q, r);
		}
		return array;
	}

	public static int[] merge (int[] array, int p, int q, int r) {
		// Algorithm to merge the unsorted sub-arrays into a sorted sub-array
		int n1 = q - p + 1;		// number of elements on left array
		int n2 = r - q;				//  number  of  elements  on  right
array

		int[] L = new int[n1+1];  // left sub-array
		int[] R = new int[n2+1];  // right-sub-array
		int i = 0, j = 0;

		// populates the left and right sub-arrays
		for(i = 0; i < n1; i++) {
			L[i] = array[p+i];
		}
		for(j = 0; j < n2; j++) {
			R[j] = array[q+j+1];
		}
		L[n1] = Integer.MAX_VALUE;
		R[n2] = Integer.MAX_VALUE;
		i = 0;
		j = 0;

		// merges the left and right array into a sorted array
		for(int k = p; k <= r; k++) {
			if(L[i] <= R[j]) {
				array[k] = L[i];
				i++;
			} else {
				array[k] = R[j];
				j++;
			}
		}
		return array;
	}

	/*
	 * n: the size of the output array
	 * k: the maximum value in the array
	 */
	public static int[] generate_random_array (int n, int k) {
		List<Integer> list;
		int[] array;
```

```
        Random rnd;

        rnd = new Random(System.currentTimeMillis());

        list = new ArrayList<Integer> ();
        for (int i = 1; i <= n; i++)
            list.add(new Integer(rnd.nextInt(k+1)));

        Collections.shuffle(list, rnd);

        array = new int[n];
        for (int i = 0; i < n; i++)
            array[i] = list.get(i).intValue();

        return array;
    }

    /*
     * n: the size of the output array
     */
    public static int[] generate_random_array (int n) {
        List<Integer> list;
        int[] array;

        list = new ArrayList<Integer> ();
        for (int i = 1; i <= n; i++)
            list.add(new Integer(i));

        Collections.shuffle(list, new Random(System.currentTimeMillis()));

        array = new int[n];
        for (int i = 0; i < n; i++)
            array[i] = list.get(i).intValue();

        return array;
    }

    /*
     * Input: an integer array
     * Output: true if the array is acsendingly sorted, otherwise return false
     */
    public static boolean check_sorted (int[] array) {
        for (int i = 1; i < array.length; i++) {
            if (array[i-1] > array[i])
                return false;
        }
        return true;
    }

    public static void print_array (int[] array) {
        for (int i = 0; i < array.length; i++)
            System.out.print(array[i] + ", ");
        System.out.println();
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("Insertion sort starts ------------------");
        for (int n = 100000; n <= 1000000; n=n+100000) {
            int[] array = Sort.generate_random_array(n);
            long t1 = System.currentTimeMillis();
            array = Sort.insertion_sort(array);
            long t2 = System.currentTimeMillis();
            long t = t2 - t1;
            boolean flag = Sort.check_sorted(array);
            System.out.println(n + "," + t + "," + flag);
        }
        System.out.println("Insertion sort ends ------------------");


        System.out.println("Merge sort starts ------------------");
        for (int n = 100000; n <= 1000000; n=n+100000) {
            int[] array = Sort.generate_random_array(n);
            //Sort.print_array(array);
            long t1 = System.currentTimeMillis();
            array = Sort.merge_sort(array, 0, n-1);
            long t2 = System.currentTimeMillis();
            long t = t2 - t1;
            //Sort.print_array(array);
            boolean flag = Sort.check_sorted(array);
            System.out.println(n + "," + t + "," + flag);
        }
        System.out.println("Merge sort ends ------------------");
    }

}
```

Task 3:

The graph and the table bellow shows the time spent by the insertion and the merge sort algorithms. It is clear that the merge sort algorithm is much faster when it comes to sorting an array with 100,000 or more elements, with the longest time being 0.156 seconds to sort 1,000,000 elements, while the shortest time for the insertion sort algorithm was 2.641 seconds to sort 100,000 elements (see table 1). These results can be confirmed by looking at the time complexity of both algorithms. The time complexity of the insertion sort algorithm is $O(n^2)$ and the merge sort algorithm is $O(n * \log(n))$ and since $n^2 > n * \log(n)$ for $n > 0$, it makes sense to have the merge sort algorithm being much faster than the insertion sort.



Number of Elements	Insertion Sort time spent(s)	Merge Sort time spent (s)
100000	2.641	0.015
200000	10.045	0.032
300000	22.813	0.047
400000	15.88	0.062
500000	24.765	0.078
600000	35.257	0.094
700000	47.999	0.125
800000	63.108	0.125
900000	80.432	0.156
1000000	98.243	0.156