Joao Paulo dos Santos Ferreira
CSIT 212_02 – Data Structures and Algorithms
Instructor: Boxiang Dong
September 26, 2018

Homework 2 – Heap & Quick & Counting Sort

1 Problem Description

Instructions. You are provided the skeleton code named *Sort2.java*. The source file is available on Canvas in a folder named *HW2*. Please modify the skeleton code to solve the following tasks.

- Task 1 (30 pts). Implement the *Heap Sort* algorithm as discussed in Lecture 3. (Hint: use the function *checked_sorted* to check if your output is indeed sorted.)
- Task 2 (30 pts). Implement the *Quick Sort* algorithm as discussed in Lecture 4. (Hint: use the function *checked_sorted* to check if your output is indeed sorted.)
- Task 3 (30 pts). Implement the *Counting Sort* algorithm as discussed in Lecture 4. (Hint: use the function *checked_sorted* to check if your output is indeed sorted.)
- Task 4 (10 pts). Generate a report to discuss the time performance of the three 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, 2 and 3 (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 Sort2 {

	// returns the index of the left child
	public static int left (int i) {
		return (2*i);
	}

	// returns the index of the right child
	public static int right (int i) {
		return (2*i+1);
	}

	// returns the index of the parent
	public static int parent (int i) {
		return i/2;
	}

	// creates a max heap rooted at i
	public static int[] max_heapify (int[] array, int heap_size, int i) {
```

```
            int l = left(i);
        int r = right(i);
        int largest;             // stores which element is the largest

        // compares parent and child and stores the index of the largest
        if(l <= heap_size && array[l - 1] > array[i - 1]) {
            largest = l;
        } else {
            largest = i;
        }

        if(r <= heap_size && array[r - 1] > array[largest - 1]) {
            largest = r;
        }

        // repeats the process until largest is stored at the top of the tree
        if(largest != i) {
            int temp = array[i - 1];
            array[i - 1] = array[largest - 1];
            array[largest - 1] = temp;
            max_heapify(array, heap_size , largest);
        }
        return array;
}

// Calls max_heapify in a bottom-up manner to build a max heap
public static int[] build_heap (int[] array) {
      for(int i = array.length/2; i > 0; i--) {
            max_heapify(array,array.length ,i);
      }
      return array;
}

/* extracts the largest element in array[0]
 * and reorganizes the remaining element into a max heap
 */
public static int[] heap_sort (int[] array) {
  build_heap(array);
  for(int i = array.length - 1; i > 0; i--) {
      int temp = array[0];
      array[0] = array[i];
      array[i] = temp;
      max_heapify(array, i, 1);
  }
      return array;
}

/*  Divides and array into two sub-arrays with one containing
 * elements smaller than or equals to the value at array[q] and
 * the other contain the elements that are larger than array[q]
 */
public static int[] quick_sort (int[] array, int p, int r) {
      if (p < r) {
            int q = partition(array, p, r);
            quick_sort(array, p, q-1);
```

```
            quick_sort(array, q+1, r);
        }
        return array;
}

/* Collects the element at the end of the array, compares it
 * with every element in the array and organize both sub-arrays.
 * Returns the new index of the element being compared.
 */
public static int partition (int[] array, int p, int r) {
        int x = array[r];
        int i = p-1;
        for(int j = p; j < r; j++) {
            if (array[j] <= x) {
                i++;
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
        int temp2 = array[r];
        array[r] = array[i+1];
        array[i+1] = temp2;
        return i+1;
}

public static int[] counting_sort (int[] array, int k) {
        int[] C = new int[k+1];
        /*
         * creates an array to count how many times each element
         * is present, initializing the count for each at 0;
         */
        for (int i = 0; i <= k; i++) {
            C[i] = 0;
        }

        /*
         * counts how many times each element is present
         * and stores it in the count array
         */
        for (int j = 0; j < array.length; j++) {
            C[array[j]] = C[array[j]] + 1;
        }

        /*
         * adds the count from the left element to the right one
         * last element should contain the value of array.length
         */
        for (int i = 1; i <= k; i++) {
            C[i] = C[i]+ C[i-1];
        }
        // creates a new array to store the sorted elements
        int[] B = new int[array.length];

        /*
```

```
         * puts each element at the right position
         * using the count array and the original array
         */
        for(int j = array.length-1; j >= 0; j--) {
              B[C[array[j]]-1] = array[j];
              C[array[j]]--;
        }
        return B;
}


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;
}

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 ascendingly 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
                int k = 10000;

                System.out.println("Heap sort starts ------------------");
                for (int n = 100000; n <= 1000000; n=n+100000) {
                        int[] array = Sort2.generate_random_array(n);
                        long t1 = System.currentTimeMillis();
                        array = Sort2.heap_sort(array);
                        long t2 = System.currentTimeMillis();
                        long t = t2 - t1;
                        boolean flag = Sort2.check_sorted(array);
                        System.out.println(n + "," + t + "," + flag);
                }
                System.out.println("Heap sort ends ------------------");

                System.out.println("Quick sort starts ------------------");
                for (int n = 100000; n <= 1000000; n=n+100000) {
                        int[] array = Sort2.generate_random_array(n);
                        long t1 = System.currentTimeMillis();
                        array = Sort2.quick_sort(array, 0, n-1);
                        long t2 = System.currentTimeMillis();
                        long t = t2 - t1;
                        boolean flag = Sort2.check_sorted(array);
                        System.out.println(n + "," + t + "," + flag);
                }
                System.out.println("Quick sort ends ------------------");

                System.out.println("Counting sort starts ------------------");
                for (int n = 100000; n <= 1000000; n=n+100000) {
                        int[] array = Sort2.generate_random_array(n, k);
                        long t1 = System.currentTimeMillis();
                        array = Sort2.counting_sort(array, k);
                        long t2 = System.currentTimeMillis();
                        long t = t2 - t1;
                        boolean flag = Sort2.check_sorted(array);
                        System.out.println(n + "," + t + "," + flag);
                }
                System.out.println("Counting sort ends ------------------");
        }
}
```

Task 4:

The graph bellow shows the time spent by Heap, Quick and counting sort algorithms. From the graph we see that the Heap sort took the longest to execute, the Quick sort executed with a shorter time than the Heap sort, and the Counting sort was the fastest one to execute. This result can be explained by looking at the time complexity of each algorithm. The time complexity for the Heap Sort algorithm is $O(n * log\ n)$, and the time complexity for the Quick sort algorithm varies from $O(n * log\ n)$ in the best case scenario to $O(n^2)$, although in practice Quick Sort is generally faster than the Heap Sort algorithm, specially on a randomly generate array, that is why they are not too far apart in the graph.. The time complexity of the counting sort is $O(n + k)$ where k is just a constant and could be ignored when comparing it with other algorithms in some cases. Since $n * logn\ >\ n$ this means that the Counting sort algorithm should have the best time performance which can be proved by the graph.