University of Georgia

Assignment 4: Sorting

Jacob Maraffi and Reed Sexton

Data Structures: CSCI 2720

Sachin Meena

7/25/18

To properly test our implementations of the four sorting algorithms we ran the program with each of the input files many times and compared the results. The collection of the results follows this introduction.

Explanation of our Quick Sort Implementation:

Our implementation of Quick Sort started with us using a pivot point of the first index. This implementation resulted in a large increase in number of comparisons needed to complete the sort. The implementation of Quick Sort uses a helper method called “split” to create two partitions of the integer array at the pivot point “sPoint”. If the pivot is greater than or less than all other elements of the array (or subarray) all other elements must be compared to the pivot. Additionally, only one partition will be created for recursive calls. In an ordered data set, all pivot points meet this condition. Therefore, each partition contains only one fewer elements than the previous, and another n-1 comparison are performed.

We decided to test our implementation of quick sort using a random pivot point and determined that it was more efficient. This is since it is more likely to pick a pivot point with data in the wrong place on both sides of the pivot point, resulting in fewer comparisons being made.

Additional Memory/ Data Structure Usage:

Our implementations of the sorting algorithms did not utilize data structures other than the integer array. The integer array “inArr” which is a private member of the sorting class was used to store all the values from the input files; the sorting algorithms were executed on this array. The way we implemented Merge Sort required a temporary integer array to be created. We chose this since it was easier to copy the contents of the “inArr” into a temporary array as an intermediate step between

splitting the array into two parts, sorting them both, and remerging it into a sorted array.

Analysis of sorting algorithms implemented on an integer array:

i. Ordered file as input (number of comparisons)

Insertion Sort	Merge Sort	Heap Sort	Quick Sort (fixed)	Quick Sort (random)
9,999	69,008	244,460	50,004,999	165,621

ii. Reverse file as input (number of comparisons)

Insertion Sort	Merge Sort	Heap Sort	Quick Sort (fixed)	Quick Sort (random)
49,995,000	64,608	226,682	49,999,999	158,201

iii. Random.txt file as input (number of comparisons)

Insertion Sort	Merge Sort	Heap Sort	Quick Sort (fixed)	Quick Sort (random)
25,154,665	120,414	235,430	159,534	166,085

iv. Mean number of comparisons for each sorting method

Insertion Sort	Merge Sort	Heap Sort	Quick Sort (fixed)	Quick Sort (random)
25,053,222	84,677	235,524	33,388,178	163,303

Explain what sorting algorithm works best in each situation:

1. Ordered.txt

When sorting an ordered data set, we found that insertion sort was most effective. This is because insertion sort can simply iterate through every element in the array, comparing every element to the next element in the array. Since the data was already in order it would find that each element was already in its correct position.

2. Reverse.txt

When sorting a reversed data set, we found that merge sort was most effective. This occurs due to the nature of merge sort itself. Splitting the array into multiple parts increases the likeliness that an element in the array is smaller or larger than all elements in another split of the array. This means that once one comparison is complete there is no need to compare that same element with every element in the adjacent split.

3. Random.txt

When sorting a randomly arranged data set, we found that merge sort was most effective, for the same reasons for its efficiency with the reversed set. Merge sort saves additional comparisons when splitting the array into smaller and smaller pieces.

4. Most consistent sorting algorithm

We calculated the mean number of comparisons for each of the sorting algorithms across the 3 different input types and determined that overall merge sort was the most effective algorithm. Merge sort reduces the number of necessary comparisons by not needing to compare a lone element with every member of its corresponding split partition. However, the downside of merge sort is that it is the only sorting algorithm that required additional memory in the form of the temporary integer array that was used to copy the sorted elements from “inArr”.