**Assignment 6**

MET CS 526 – Data Structures and Algorithms

Michael Melvin

**Overview**

In this exercise we look at four popular sorting algorithms: insertion sort, merge sort, quick sort, and heap sort. We will experimentally estimate the running time of these algorithms over varying input sizes, ranging from 10,000 to 100,000, in increments of 10,000. We do this by clocking the starting time and end time before and after execution of each algorithm, for each input size. Then, we plot the elapsed time (end time – start time) against the input size.

Insertion sort proceeds by considering one element at a time and placing that element in the correct order relative to those before it. Insertion sort of an array has best case running time of Ω(n), and worst case running time of O(n). The best case realized if the array is already sorted, and worst case is realized if the array is sorted in reverse order. A high-level pseudocode for insertion sort is shown below, and the implementation used is from our text book.

**Algorithm**: InsertionSort(A):

**Input**: An array A of n comparable elements.

**Output**: The array A with elements rearranged in nondecreasing order.

**For** *k* from 1 to *n-1* **do**

Insert A[k] at its proper location within A[0], A[1],…, A[k].

Unlike insertion sort, merge sort makes use of the divide and conquer strategy, which can be broken down as follows. Divide: if the array has one or no elements it is already sorted, otherwise put each half of the array in to two separate arrays. Conquer: perform the divide step recursively. Then we recombine the arrays into one. A high-level pseudocode for merge sort is shown below. Merge sort is O(n log n) for best and worst case, assuming the elements can be compared in O(1) time. The implementation used is a slightly modified version of our text books.

**Algorithm:** mergeSort(A, c):

**Input:**  An array of n comparable elements, and a Comparator c.

**Output:**  The array A with elements rearranged in nondecreasing order.

1. Find the middle point for the array
   1. Copy first half into a separate array
   2. Copy second half into a separate array
2. Call merge sort on each half.
3. Combine the arrays into original array.

Much like merge sort, our next algorithm, quick sort, utilizes the divide and conquer strategy, but in a slightly different way than merge sort. Where in merge sort, all of the ‘hard work’ is done before making the recursive calls. Quick sort has best case running time of O(n log n) and a worst case running time of O(n^2), which occurs if the last element in the array (the pivot) is the largest element of the array. A high-level pseudocode for quick sort is shown below. The implementation used is a slightly modified version of our text books.

**Algorithm:** quicksort(A, c, s, e):

**Input:** An array A of n comparable elements. A Comparator c, and a starting and ending index, s and e, respectively.

**Output:** The array A with elements rearranged in nondecreasing order.

1. If s has at least two elements, make the last element the pivot, and put other elements into one of three sequences.
   1. L, storing the elements of A less than the pivot.
   2. E, storing the elements of A equal to the pivot.
   3. G, storing the elements of A greater than the pivot.
2. Recursively sort L and G.
3. Combine the results back into the original array.

While the implementation of heap sort is more involved than the other three sorting algorithms in this exercise, conceptually it is rather simple. We will omit a pseudocode for this reason, but we will still discuss it. Heap sort begins by inserting all elements into a heap, then swaps the maximum element with the minimum element. Because this breaks the max-heap property, so we ‘heapify’, recursively fixing the children, until the heap property is satisfied. The best and worst case running time of heap sort is O(n log n).

**Results**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **n/Algorithm** | **10000** | **20000** | **30000** | **40000** | **50000** | **60000** | **70000** | **80000** | **90000** | **100000** |
| **Insertion Sort** | 15 | 64 | 102 | 176 | 359 | 400 | 494 | 626 | 774 | 950 |
| **Merge Sort** | 18 | 10 | 17 | 23 | 29 | 14 | 15 | 16 | 20 | 23 |
| **Quick Sort** | 22 | 8 | 16 | 17 | 10 | 7 | 10 | 10 | 11 | 12 |
| **Heap Sort** | 8 | 3 | 5 | 6 | 7 | 8 | 10 | 11 | 11 | 13 |

Clearly insertion sort is the slowest of the four algorithms, except at the 10,000-input size. There it ranks second, still almost twice as slow as heap sort, but is closely followed by merge sort. Then, heap sort keeps the lead until around the 60,000-input size, where quick sort catches it, and remains level with, or faster than it for the remainder of the input sizes. As for merge sort, it is never faster than heap sort, but it does outperform quick sort at the 10,000-input size.

**Conclusions**

As can be seen by the running time of the algorithms, insertion sort should have, and did, have the worst performance. Merge sort, quick sort, and heap sort, should have been relatively close in performance, with quick sort having the slowest worst case running time. We saw that heap sort consistently outperformed all of the other algorithms, but the implementation of heap sort is much more involved than the rest of the algorithms. Also, it needs to be noted that this experiment is not totally conclusive. Many factors come into play when running these algorithms, including the software and hardware environments of the computer they are ran on, as well as the object types that are being compared, and many other factors.