Kevin Linnane
Data Structures 2720B

# Sorting Algorithms

Background

**Insertion-Sort:**

Concept: Insertion Sort is a simple sorting algorithm that takes the array/data structure at face value. It compares two elements and swaps the value at each location if the element on the left is greater than the one on the right. After the swap, it then ensures that it is in the right location by checking the left once more of the recently swapped value and if not, repeats the swapping/comparison process. We obtain the worst-case scenario when the list is reversed because we traverse and swap each value with the above rule.

Best Case Scenario: O(N)

Average/Worst Case Scenario: O(N^2)

**Merge-Sort:**

Concept: First we divide the given array recursively into two halves. Once there are only single elements in an array, we can start merging because such arrays are sorted if it contains one element. To merge, we combine two sorted sub arrays. We obtain the worst-case scenario when the second biggest and the biggest elements are in two different sub-arrays. For example if we had an array containing [4, 1, 2, 9] 4 and 9 are in separate sub-arrays and the number of comparisons between {4,1} and {2,9} would be three.

Worst/Average/Best Case Scenario: O(NLogN)

**Quick-Sort:**

Concept: Quick-sort, much like merge-sort is a divide and conquer algorithm that divides arrays into two partitions. The two partitions are sorted recursively. Prior to sorting, we choose/encode a pivot point for the array to center around. For this experiment, the first iteration's pivot point is the midpoint of the array, then changes with the recursive call to search the left and right side of pivot points. Using the array's value at the pivot point location, we place all items smaller than the pivot value to the left, and all values larger than the pivot point on the right. For each partition, the less we must compare because there will be guaranteed one element in the correct location. We obtain the worst-case scenario when the smallest or the largest values are assigned to the pivot point value.

Average/Best Case Sccenario: O(NLogN)

Worst Case Scenario: O(N^2)

Kevin Linnane
Data Structures 2720B

Experiment

Using three files each containing ten thousand integers of either Ordered, Reversed, or Random numbers, we perform the sorting algorithms discussed above with an Array data structure to determine which algorithm be most optimized for a specified data set.

**Data:**

Figure 1.1

| # of Comparisons | | | |
|---|---|---|---|
| | Ordered.txt | Revered.txt | Random.txt |
| Insertion-Sort | 9999 | 49995000 | 25039782 |
| Merge-Sort | 69008 | 64608 | 120388 |
| Quick-Sort | 119535 | 124535 | 192281 |

Figure 1.2

| Comparison Differences | | | |
|---|---|---|---|
| | Ordered.txt | Reversed.txt | Random.txt |
| Insertion-Merge | -59009 | 49930392 | 24919394 |
| Insertion-Quick | -109536 | 49870465 | 24847501 |
| Merge-Quick | -50527 | -59927 | -71893 |

**Results:**
When ordered input is applied, Insertion sort compares only N amount of times, and is significantly better to use than a Merge or Quick-Sort. It compares 59009 times less than Merge-Sort and 109538 times less than Quick-Sort. With a reversed text, Merge-Sort compares half of what Quick-Sort compares and a substantial less comparison than insertion Sort. With a text document full of random integers, Merge-Sort also does less comparisons than Quick and Insertion Sort.

Insertion-Sort should only be used if and only if a list is already ordered or data set is small, otherwise either Quick or Merge Sort should be applied. Merge-Sort is more consistent with larger input than Quick-Sort, since each mergesort is working on a different portion of the array it has threads working parallel to the sorting. Quick-Sort's pivot will on average give evenly divided arrays and therefore can be optimized towards knowing what data set is being received and change the pivot point accordingly to surpass Merge-Sort.