Michael Winkler and John Zimmer

CSC335

Prof. Duvall

4/4/16

Project 3: Sort Detective

For each of the unknown sorts, we recorded in an excel spreadsheet the number of comparisons and movements that resulted from combining each possible initial ordering (InOrder, ReverseOrder, AlmostOrder and Random) with lists of size 16, 64, 256, and 1024. The results are as follows:

****

**Alpha: Merge Sort:**

* We think that Alpha is merge sort because it consistently runs in O(nlgn) time, regardless of whether the array is in-order, reverse-order, or random-order. Merge sort has a best-case, average-case, and worst-case running time of O(nlgn), so the fact that alpha has approximately the same runtime regardless of how the array it runs on is sorted suggests that it’s merges sort.

**Beta: Bubble Sort**

* We noted right away that this sort ran in O(n) time on an ordered list and in O(n2) time on an unordered list, which made us think that it had to be either bubble sort or insertion sort.
* We decided that it was bubble sort instead of insertion sort because when run on a pre-sorted list, beta produces only one movement, which matches the given implementation of bubble sort and not the implementation of insertion sort (which increments movement approximately 2n times no matter whether the list is initially sorted or not.

**Gamma: Insertion Sort**

* We were initially tipped off the Gamma might be insertion sort because we saw that if the list to be sorted is already ordered, then Gamma produces O(n) comparisons and movements, whereas if the list is initially in reverse order, then Game produces O(n2) comparisons and movements. That sounds like insertion sort, which has a best-case running time of O(n) and worst-case of O(n2).
* We also looked at the specific numbers produced by running Gamma on an ordered list and saw that it produces exactly 2n-2 movements and n-1 comparisons, which matches the given implementation of insertion sort.

**Delta: Quicksort**

* We think that delta is quicksort because delta runs significantly faster on a randomly sorted list than it does on an in-order or reverse-order list. Quicksort should run faster on a randomly sorted list than on an in-order or reverse-order list because the given implementation of quicksort uses the first element of the list (and sub-arrays) as its pivot point. Shame on whoever implemented quicksort this way ☹

**Epsilon: Selection Sort**

* Epsilon resulted the same number of comparisons (O(n2)) regardless of whether the list was initially sorted or not, which suggests that it’s selection sort, because selection sort always iterates through the entirety of the unsorted sub-array before choosing a value to add to the sorted sub-array.
* Also, whenever we ran Epsilon on a sorted list, it resulted in zero movements, which again suggests that Epsilon is selection sort because the provided implementation of selection sort only moves “if it must”.

**Zeta: Heap Sort**

* We think that Zeta is heap sort because its best-case running time is running on a reverse-order list and its worst case running time is running on an in-order list. This suggests that Zeta is heap sort because if you think of a list as a heap, a reverse-order list is already heapified, whereas an in-order list will have to be completely heapified before the rest of the sorting can begin.

**Citations**

[www.wikipedia.org](http://www.wikipedia.org) (looked at the pages for each sort)

<https://www.youtube.com/watch?v=D_B3HN4gcUA>

<https://www.youtube.com/watch?v=kPRA0W1kECg>

<https://www.youtube.com/watch?v=onlhnHpGgC4>

http://betterexplained.com/articles/sorting-algorithms/