**Tiodor Dimitrov**

**619-50 Project 1**

**03/10/17**

**Theoretical Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| Sorting Algorithm | Best | Average | Worst |
| Insertion Sort | Ω(n) | Θ(n^2) | O(n^2) |
| Merge Sort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) |
| Quick Sort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) |
| Heap Sort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) |
| Radix Sort | Ω(nk) | Θ(nk) | O(nk) |

Some algorithms (selection, bubble, heapsort) work by moving elements to their final position, one at a time. Some algorithms (insertion, quicksort, counting, radix) put items into a temporary position, close to their final position. “Comparison sorts” make no assumptions on the data and compare all elements against each other. The typical default sort implementation for most languages is either Mergesort or Quicksort. These "divide-and-conquer" algorithms sort by recursively dividing the list into smaller sublists which are then sorted. O(N lg N) time is the ideal “worst-case” scenario of a sorting algorithm.

Insertion Sort is efficient for small data sets . The best case input is an array that is already sorted. The simplest worst case input is an array sorted in reverse order. The average case is quadratic, which makes insertion sort impractical for sorting large arrays.

Merge sort has an [average](https://en.wikipedia.org/wiki/Average_performance) and [worst-case performance](https://en.wikipedia.org/wiki/Worst-case_performance) of [O](https://en.wikipedia.org/wiki/Big_O_notation)(n log n). Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially. Merge sort is the standard routine in [Perl](https://en.wikipedia.org/wiki/Perl) and a variation of it is used as the standard sort method in Python, Java, and Android as well.

Quick sort takes [O](https://en.wikipedia.org/wiki/Big_O_notation)(n log n) comparisons to sort n items [on average](https://en.wikipedia.org/wiki/Best,_worst_and_average_case). In the [worst case](https://en.wikipedia.org/wiki/Best,_worst_and_average_case), it makes O(n2) comparisons, though this behavior is rare. When implemented well, it can be about two or three times faster than its main competitors, [merge sort](https://en.wikipedia.org/wiki/Merge_sort) and [heapsort](https://en.wikipedia.org/wiki/Heapsort).

Heap sort is a much more efficient version of [selection sort](https://en.wikipedia.org/wiki/Selection_sort). It works by determining the largest (or smallest) element of the list, placing it at the end (or beginning) of the list, then continuing with the rest of the list, by using a [heap](https://en.wikipedia.org/wiki/Heap_%28data_structure%29) data structure. Creating the heap is O(N lg N). Popping items is O(1), and fixing the heap after the pop is lgN. There are N pops, so there is another O(N lgN) factor, which is O(N lg N) overall.

Radix sort is an algorithm that sorts numbers by processing individual digits. n numbers consisting of k digits each are sorted in O(n · k) time. if all n keys are distinct, then k has to be at least log n for a random-access machine to be able to store them in memory, which gives at best a time complexity O(n log n); this makes it at least as equally efficient as the other sorts.

**Hypothesis**

Based on the time complexity analysis, I expect for Insertion sort to run the slowest, merge/quick/heap sorts to be about equal, and for radix sort to run the fastest during experimentation.

**Data generation and experimental setup**

input types: ascending, descending, random, ascending\_shuffled.

input sizes: 10000, 100000, 1000000, 10000000

experiment repeat: 3

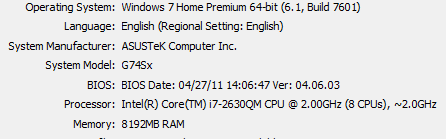
algorithm types: Insertion, merge, quick, heap, radix sorts

of different inputs: 4\*4=16

# of experiments per algorithm: 4\*4\*3=48

total # of experiments: 4\*4\*3\*5=240

What kind of machine did you use?



What timing mechanism? How does your machine measure time?

ANSI C routine clock().

How many times did you repeat each experiment?

3

What times are reported?

Milliseconds of run time

How did you select the inputs?

I wanted the input size to be large enough to run for a good amount of time in order to be comparable between algorithms but less than 1 minute. I ran a test to gauge this number.

Did you use the same inputs for all sorting algorithms?

yes