**Tiodor Dimitrov**

**619-50 Project 1**

**03/20/17**

**#1 Theoretical Analysis**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Sorting Algorithm | Best | | Average | | Worst | |
|  | input | complexity | input | complexity | input | complexity |
| Insertion Sort | asc | Ω(n) | rand | Θ(n^2) | desc | O(n^2) |
| Merge Sort | rand | Ω(n log(n)) | rand | Θ(n log(n)) | rand | O(n log(n)) |
| Quick Sort | rand | Ω(n log(n)) | rand | Θ(n log(n)) | asc | O(n^2) |
| Heap Sort | rand | Ω(n log(n)) | rand | Θ(n log(n)) | rand | O(n log(n)) |
| Radix Sort | rand | Ω(nk) | rand | Θ(nk) | rand | 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. "Divide-and-conquer" algorithms sort by recursively dividing the list into smaller sublists which are then sorted. The typical default sort implementation for most languages is either Mergesort or Quicksort. 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 input is an array that is already sorted. With this input each iteration compares the first remaining element with the right-most element of the sorted subsection of the array. The simplest worst case input is an array sorted in reverse order. The average case is quadratic, as when the input is random, 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) regardless of the input. 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) and in the best case The best case occurs when each time we perform a split we divide the list into two nearly equal pieces. 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). In the [worst case](https://en.wikipedia.org/wiki/Best,_worst_and_average_case), quick sort makes O(n2) comparisons. This occurs when the pivot divides the list into two sublists of sizes 0 and n − 1.

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 regardless of the input..

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) regardless of the input.; this makes it at least as equally efficient as the other sorts.

**Hypothesis**

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

Because the way the algorithms work, only insertion sort and quick sort should have any difference in run time as a result of using a different input type (asc/desc/rand).

The rest of the algorithms should run in about the same time regardless if the input type. Their run times should grow linearly as the input size increases.

**#2 Data generation and experimental setup**

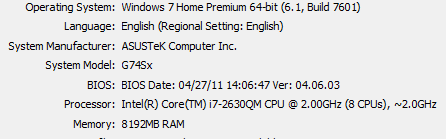
input types: ascending, descending, random

input sizes: 1,000,10,000, 100,000, 1,000,000, 10,000,000

experiment repeat: 3

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

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.

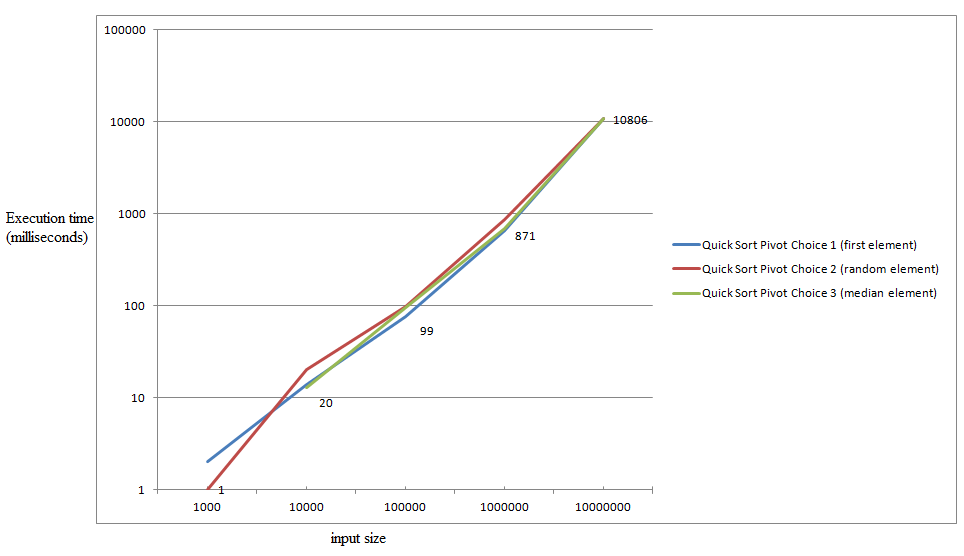
Did you use the same inputs for all sorting algorithms?

yes

**#3 Multiple Versions of QuickSort**

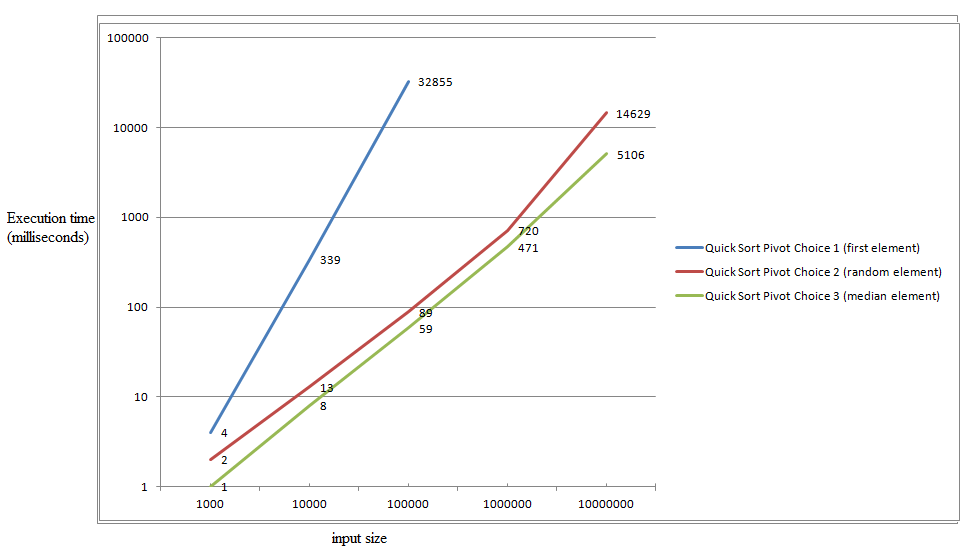
**Quick Sort Best and average case input (random):**

**3 runs averaged of input sizes 1000, 10000, 100000, 1000000, 10000000**

****

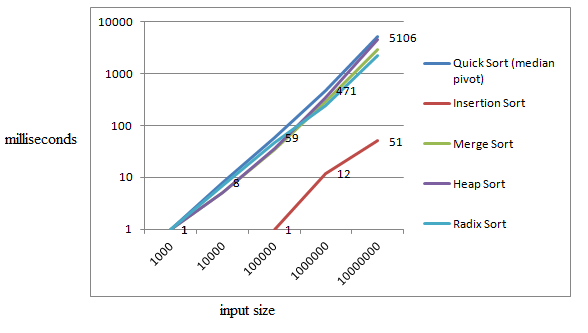
**Quick Sort Worst case input (ascending):**

**3 runs averaged of input sizes 1000, 10000, 100000, 1000000, 10000000**

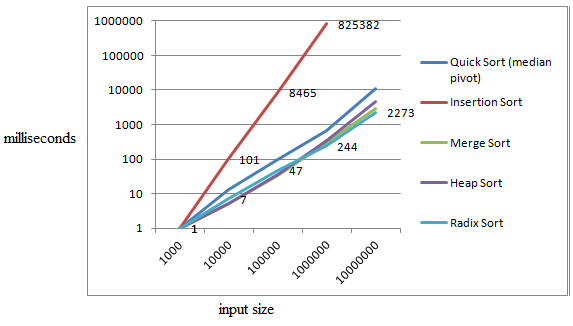
****

**#4 Sort Comparison**

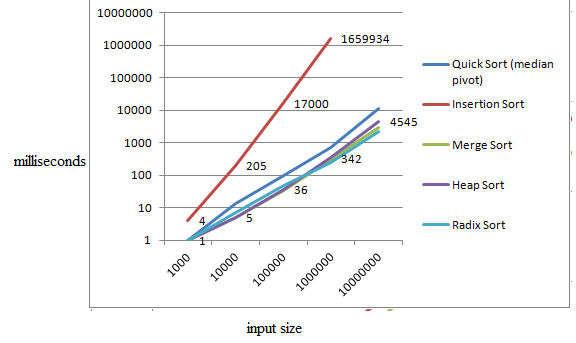
**Best case input: 3 runs averaged of input sizes 1000, 10000, 100000, 1000000, 10000000**

****

**Average case input: 3 runs averaged of input sizes 1000, 10000, 100000, 1000000, 10000000**

****

**Worst case input: 3 runs averaged of input sizes 1000, 10000, 100000, 1000000, 10000000**

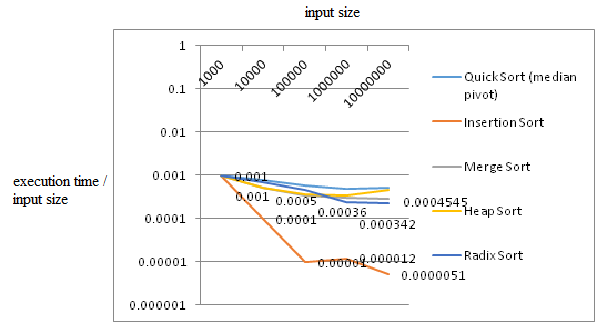
****

**#5 Time complexity analysis**

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

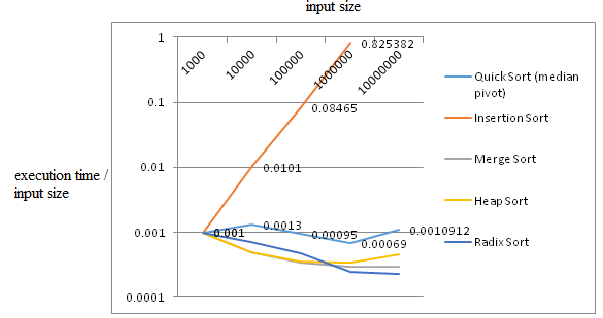
**Best Case**

By dividing the time with different input sizes, you can see that as predicted, Insertion sort grows at a rate slower than the other sorts in the best case.

****

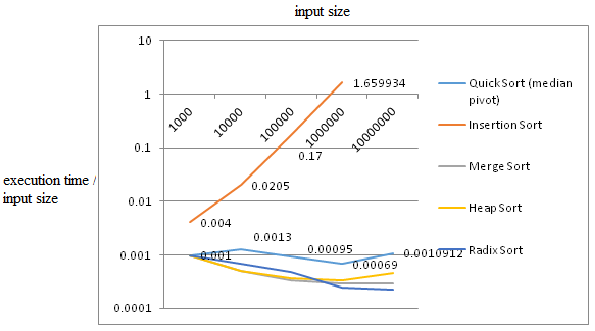
**Average Case**

The horizontal line for 4 of the 5 sorts indicates that the run time increases at about the same rate as the input size, n, increases. This correlates to the theoretical time complexity of the sorts in the average case of Θ(n log(n)). Insertion sort on the other hand has a linear curve which indicates that the run time scales with input size at a n^2 pace.



**Worst Case**

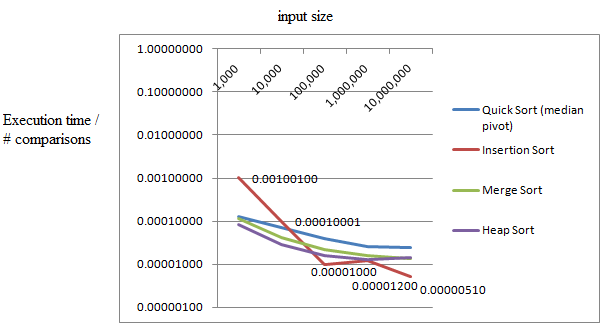
My experimental results for the sort algorithms also correlate with the theoretical Big Oh complexity as can be seen by the chart. 4/5 of the sorts grow in a linear fashion whereas insertion sort grows exponentially.

****

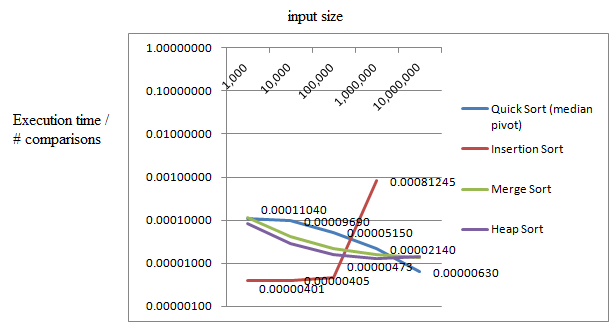
**#6 Number of Comparisons**

I found that for the comparison sort algorithms, comparisons is a good predictor of the execute time as there appeared to be correlation between the two. I found that the # of comparisons grew about linearly (as evidenced by the horizontal line for time/comparison) for algorithms with a linear time complexity and exponentially (increasing line for time/comparison) when the complexity was n^2.

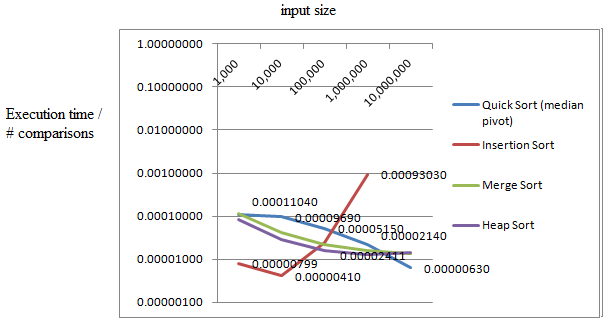
**Best Case**

****

**Average Case**

****

**Worst Case**

****

**#7 Radix Sort Experiment**

Implement several variations of the radix sort algorithm using different values for the number of passes, r. Then for each variation use a different bit length for the input. Run the experiments using different values for b and r and note the running time for each. For example:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | b | | | |
| 8 | 16 | 32 | 64 |
| r | 2 |  |  |  |  |
| 4 |  |  |  |  |
| 8 |  |  |  |  |
| 16 |  |  |  |  |

Then show that the running time is smallest when T(n,b) is minimized.

