Esteban Valle
Algorithms
Prof. Harmon
30 October 2015

Paige™

Page-Accessed Information Gateway Experiment

Part I `JavaSortedPager` Run-Time Analysis:

See below for comparison with ScoredResultsPager.

Run-Time Performance for Key Methods:

1. `JavaSortedPager(T [] vals, int pageSize)`

   O(n•log(n)) worst case performance, guaranteed by Java's Arrays.sort() method, which uses either an optimized "stable, adaptive, iterative" merge sort or a heavily optimized twin-pivot quicksort depending on which datatype is sorted. The documentation states that the sort most often requires "far fewer than n*lg(n) comparisons".

2. `int pageSize();`

   Constant. Accessor method.

3. `int pages();`

   Constant. Accessor method with one calculation.

4. `int size();`

   Constant. Accessor method.

5. `T [] page(int i);`

   Linear on first call (visits first '`pageSize()`' elements). Constant time thereafter, if caching is enabled. Guaranteed.

Part II `ScoredResultsPager`

Predictive Analysis

See below

Run-Time Analysis:

See below.

Run-Time Performance for Key Methods:

6. `JavaSortedPager(T [] vals, int pageSize)`

    Runtime cannot be guaranteed better than O(n•log(n)). In the best case scenario, using the Improved Flash Sort, runtime could be given in O(n) time. This is assuming uniformly distributed data with no repeating values. This type of data set would result in n placement operations, and the resulting insertion sort would not move any objects.

7. `int pageSize();`

    Constant time operation. Accessor method.

8. `int pages();`

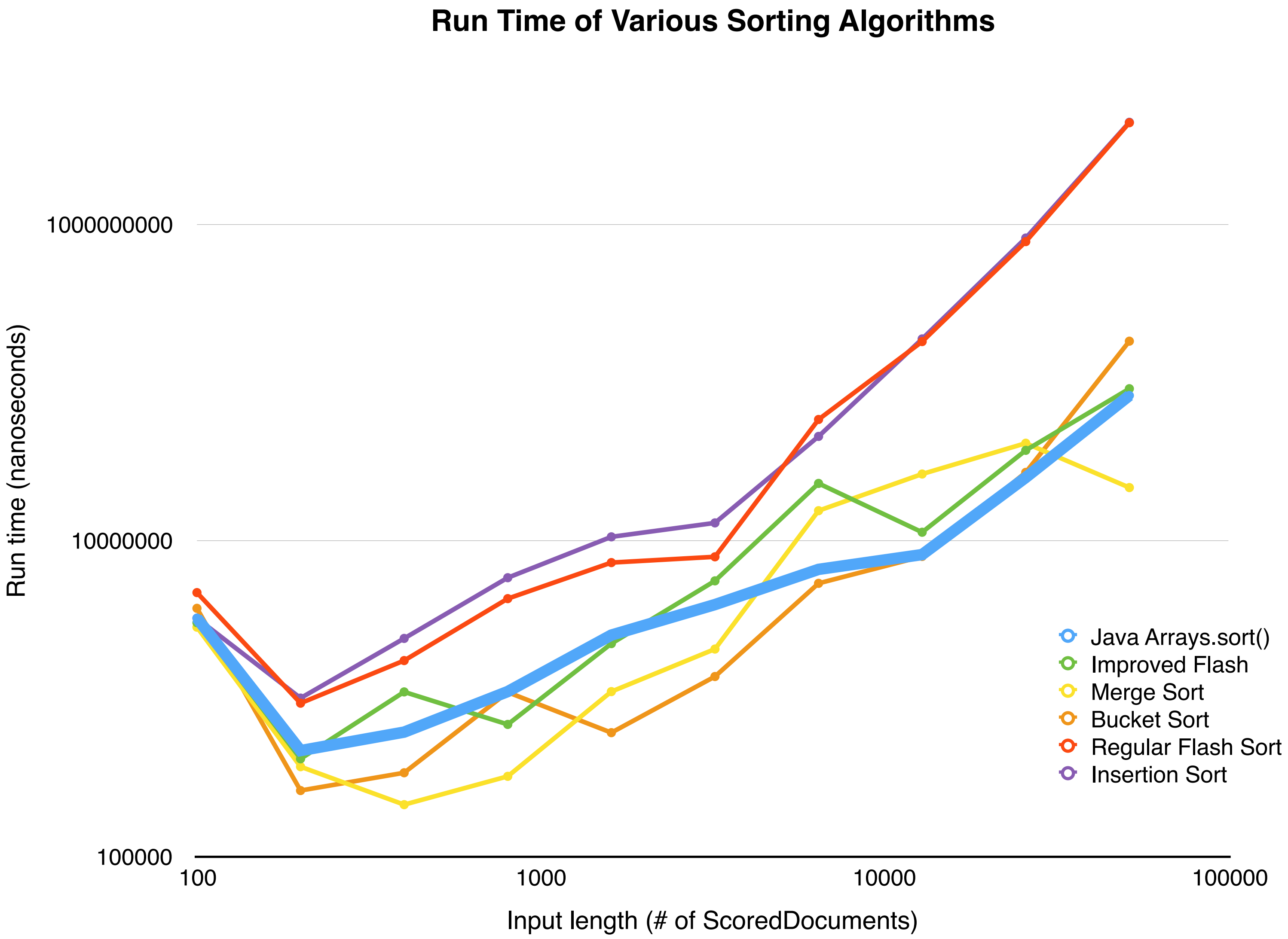
    Constant time operation. Returns simple mathematical expression.

9. `int size();`

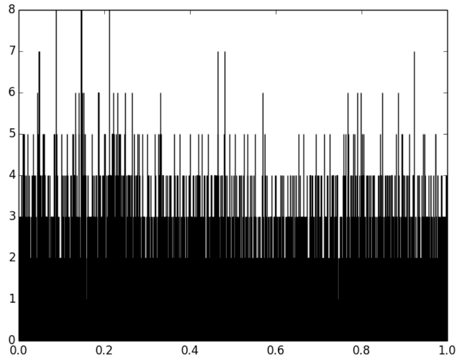
    Constant time operation. Accessor method.

10.`T [] page(int i);`

    Linear on first call (visits first '`pageSize()`' elements). Constant time thereafter, if caching is enabled. Guaranteed.



Because both JavaSortedPager and ScoredDocumentPager are identical classes, with the only exception being a differing implementation of the sortInput() (and related) method, runtime analysis was performed comparatively on both classes, with sorting algorithm serving as independent variable. The tests were conducted on instantiating a pager object—a process which automatically initiates a sort—and returning the first page of pageable data. Testing parameters can be seen in the attached PAIGE.java.



Test data were identical across sorting algorithms, and were originally calculated using the provided methods. Statistical analysis (left) shows a very highly uniform distribution for 1,000,000 doubles [0,1) generated using the StdRandom.uniform() method.

Runtime of Java’s built in Arrays.sort() method is shown in the above graph in bright blue, and is highlighted as the thicker of the lines. As stated above, this algorithm, when used with objects inheriting from Object utilizes a heavily optimized merge-sort which is able to take advantage of pre-ordered subsections, both ascending and descending, to deliver “performance requiring far fewer than n•log(n) comparisons on average.” This test serves as the baseline performance for the custom methods written for the specific dataset.

Owing to the uniform distribution of the data (see above) and the a priori knowledge that the range of the data was between 0 and 1, the first algorithm I wrote was a bucket sort. Although the particular bucket sort I wrote was particularly inefficient, suffering from massive instantiation overhead, and using unconscionable amounts of stack space memory for recursion, it provided some of the fastest run times of any of the algorithms. For these tests, five buckets were used in each level of recursion. The bucket sort would ideally require less than n•log(n) comparisons; the major complexity comes from the final use of insertion sort to sort the buckets.

Following in that line, the insertion sort that I used is fairly naïve, and in the worst case scenario would result in $O(n^2)$ complexity, although in the best case could deliver linear performance. It is shown in the graph as one of the highest-order runtime algorithms.

Because in a best-case scenario, insertion sort would deliver linear time complexity, I decided to set the sort up with as best-case data as was possible: nearly perfectly sorted already. In order to do this, I again referenced the a priori knowledge of the range and uniformity of the data, and decided to create a flash sort. Flash sort uses the principle of uniformity to state that an object of value x will be liable to end up near slot x * n, normalized for range. The first in-place implementation of this sort is very naive, and simply follows values to their destinations, picks up another value from where the first was placed, and repeats. This algorithm was one of the slowest, delivering complexity in the worst case of O(n•log(n)), likely due to the fact that the actual flash algorithm was not perfect at pre-selecting the data for the ultimate insertion sort, so the algorithm effectively took the form of this insertion sort, but without the possibility of worst-case reverse-sorted data.

In order to improve this algorithm, I wrote a new method called Fuckin’ Fast™ Flash Sort (hereafter referred to as, ‘Improved Flash Sort’), a name whose vulgarity stems from my belief that the stronger I presume the algorithm to be fast, the faster it will be. For this reason, the source code for this implementation contains two poems, a prayer to the Java Compiler, and racing stripes. I would color it red, but I couldn’t figure out a way. This algorithm leverages a second data structure to enhance performance at the expense of memory. It also utilizes an online, in-place insertion sort to actively place elements in their eventual position as early as possible. The improvements in this algorithm are substantial over the naïve flash sort, although performance still lags behind due to the final insertion sort which is needed to ultimately sort the roughly ordered data. Irrespective of data set, the Improved Flash Sort seems to inevitably land five or ten percent of values very far away from their ultimate home, which produces significant overhead for the insertion sort. In the best case scenario, this algorithm can deliver n time and comparison complexity. The best case scenario for this algorithm is perfectly uniform data with no repeating values. The average case appears to mimic Arrays.sort() at O(n•log(n)), with worst-case performance escalating to O(n2) if the final insertion operation is the only effective means of sorting, which would happen if the data were highly skewed towards a particular value, or had many repeating values.

I also wrote a merge sort for fun. I wrote it because my program only uses the optimized algorithms if it is known that the abstract data type is a ScoredDocument. Because the interface does not specify this, I needed to write a custom algorithm for every possible T, so I wrote the merge sort which relies only on the natural order of elements via Comparable interface, and not the specific values, as are needed for the more advanced algorithms above. Merge Sort ended up being one of the fastest algorithms in the program, despite its totally unspecific implementation, and total lack of any optimizations, assuming nothing about the input data. Because of the lack of assumptions, average, worst case, and best case performance for this algorithm is O(n•log(n)).