Ross Grogan-Kaylor
CS 201 (Eric Alexander)

Search for the Optimal Quicksort Base Case

Introduction

As we learned in class, it is common practice to call insertion sort once subarrays being passed recursively to a quicksort method have become "small enough." Here, I investigate what "small enough" really means, and set out to determine what the size of subarrays that trigger insertion sort should be.

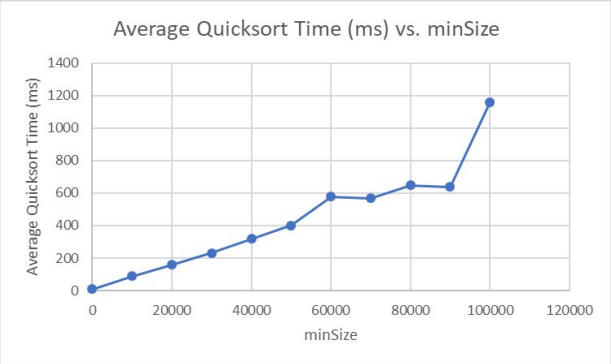
In the following analysis, the size of an array that triggers the base case of quicksort will be referred to as `minSize`.

Timing Methodology

In order to analyze the runtime of quicksort for various different `minSize`s, I wrote two methods (along with some other helper methods) to encapsulate the functionality of testing. The first method, `timeAverageQuickSort(int arrSize, int minSize, int n)`, returns the average, over `n` trials, of the times it took to quicksort randomly-generated arrays with the specified `minSize`. The second method, `writeCSVFileWithMinSizeTestData(int arrSize, int divisions, int finalMinSize, int numTrials, String file)`, writes to the specified filepath `file` a CSV table that varies `minSize` over the range [`0, finalMinSize`] and compares each value of `minSize` to the associated value returned by `timeAverageQuicksort`.

I used `writeCSVFileWithMinSizeTestData` to create CSV tables and Excel plots for quicksort time tests done on an array of length 100,000, and gradually decreased the `finalMinSize` parameter specified to this method in order to "zoom" in on the parts of the plots that were interesting.

Test 1- Broad Overview (`minSize` ranges from 0 to 100,000)



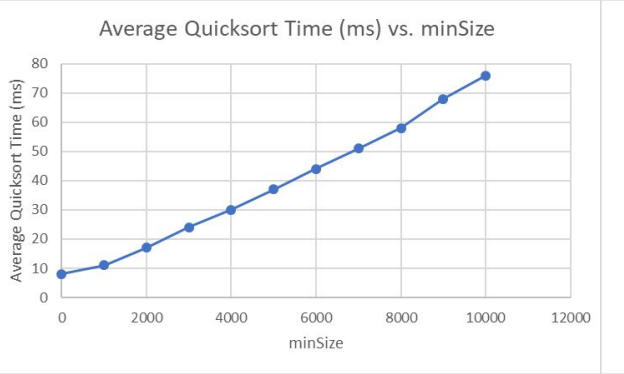
Because the largest value of `minSize` tested was so large (100,000), 10 trials were used to compute each average quicksort time for this test.

According to this plot, average quicksort time depends linearly on `minSize` until `minSize` is approximately 50% of the length of the array being sorted.

Quicksort becomes much less efficient after `minSize` passes this threshold because a `minSize` that is 50% of the original array's length causes the quicksort algorithm to fall back on the insertion sort base case almost immediately. Subarrays passed to the first set of recursive calls to quicksort are obtained by splitting the original array in half, so at least one of the subarrays will have a length that is 50% of the original array's length. Even in this best worst case, where only one of the subarrays has a length that triggers the insertion sort base case, it will only take one more set of recursive calls for insertion sort to be handling the sorting of all subarrays.

Evidently, it is a bad idea to use a `minSize` that is greater than or equal to half the size of the array being sorted. If `minSize` is restricted so that it cannot become this large, then examining how increasing `minSize` linearly increases sort time leads to the conjecture that a `minSize` that is small compared to the size of the original array is most efficient.

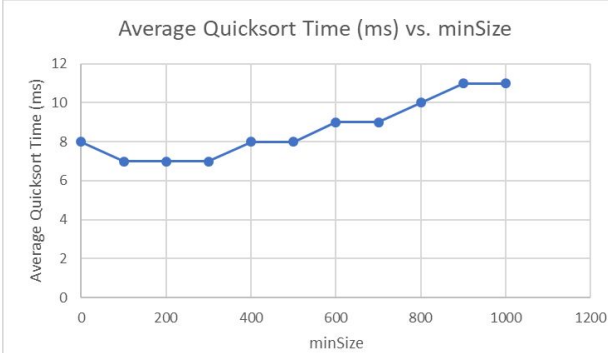
Test 2- Zooming In (`minSize` ranges from 0 to 10,000)



In a second round of testing, `minSize` was varied over a smaller range so as to "zoom in" to the values of minSize that corresponded to small average quicksort times in the previous test. I wanted to see if the plot would look less linear when placed on a smaller scale.

Because the slope of the graph when `minSize` varies from 0 to 1,000 is the smallest slope on the entire graph, I decided to zoom in even more.

Test 3- Zooming In More: (`minSize` ranges from 0 to 1,000)

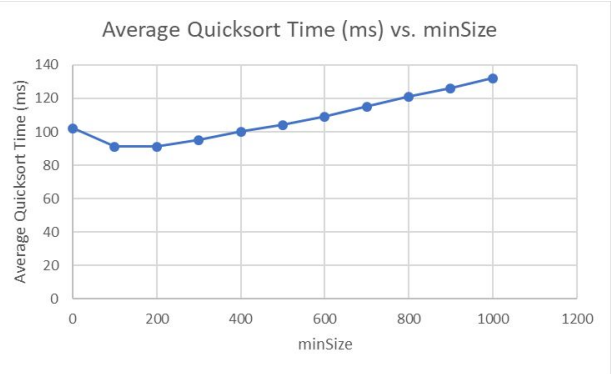


The third round of testing reveals that the fastest quicksort time occurs when `minSize` ranges from 100 to 300.

100 trials were used to compute each average quicksort time for tests 2 and 3.

Test 4- Testing Dependence of Results on Size of Original Array
(`arrSize` = 1,000,000, `minSize` ranges from 0 to 10,000)

In order to see whether the results from test 3 depend on the array size or not, I ran quicksort tests on an arrays with lengths equal to ten times the length of the previous arrays.

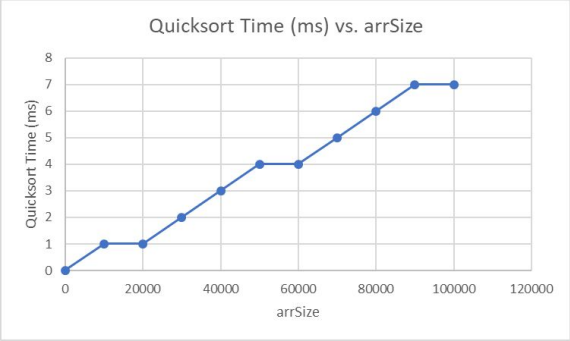


The minima in this plot occur at almost the exact same locations as they do in the plot from test 3. In the plot from test 3, the minima occurred when `minSize` ranged from 100 to 300; here, they occur when `minSize` ranges from 100 to 200. It seems that changing the size of the array shrinks the amount of different `minSize`s that correspond to minima; however, the choice of `minSize` = 200 seems to consistently correspond to a minimum sorting time even as the size of the array being sorted varies.

Conclusion: `minSize` = 200 corresponds to the most efficient quicksort algorithm.

Analysis of Quicksort Runtime

After running `writeCSVFileWithRuntimeTestData`, a method that writes to a CSV the size of an array and its associated quicksort time, the following plot was generated.



100 trials were used to compute each average quicksort time for this test.
As 200 was the value determined to be the most efficient `minSize`, `minSize` = 200 was used for all quicksorts.

Quicksort has a best case big O runtime of O($n\log(n)$), and a worst case big O runtime of O(n^2) (where n = `arrSize`). The average runtime observed in this graph is somewhere in-between the best and worst cases, as when $n\log(n)$ was plotted along with the actual runtime data, it became indistinguishable from the x-axis; when the actual runtime data was plotted along with n^2, the runtime data then became indistinguishable from the x-axis.

Also noticeable on the above plot are the various ranges of zero slope. I think that these ranges are a result of best and worst case runtimes canceling each other out in the averaging process.

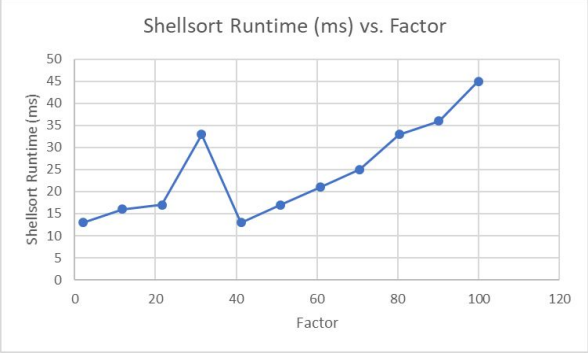
Shell Sort Multiplicative Factor Analysis

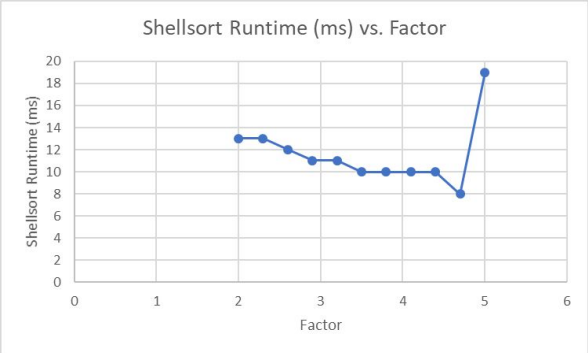
Timing Methodology

The methods I used to time shell sort are almost identical to the ones I used to time quicksort.

`timeAverageShellSort` computes the average of the times it took to shell sort randomly-generated arrays with the specified `minSize`, and `writeCSVFileWithFactorTestData` creates a CSV file that contains columns for `factor` and the corresponding average shell sort time, with `factor` varying from 0 to the specified `finalFactor` parameter.

Broad Trends Regarding `factor`





According to the topmost plot- there is a kind of “anti-Goldilocks” region of choices of `factor` that correspond to slow sort times. I think that this is because `factor` must lie in an extreme (it must be “small” or “large”) for shell sort to run quickly.

When `factor` is large, the gap sequence consists of a small number of smaller gap sizes. When `factor` is small, the gap sequence consists of a large number of larger gap sizes. Each number in a gap sequence corresponds to a series of gapwise insertion sorts; this means a large `factor` corresponds to *more* of these series of insertion sorts occuring on arrays that shrink more *slowly* and that a small `factor` corresponds to *less* of these series of insertion sorts occurring on arrays that get shrink more *quickly*. There’s a tradeoff- we can either get a fast shell sort by doing many series of insertion sorts on small arrays, or by doing few series of insertion sorts on large arrays. The point is, the middle ground doesn’t work.

Considering smaller `factor`s with the plot on the bottom of the last page, the best `factor` seems to be about 4.75.

Analysis of Tokuda's Sequence (`factor` = 2.25)

Using the `timeAverageShellSort` method by itself on arrays of length 1,000,000, I compared the "standard" shell sort factor of 2 with the factor corresponding to Tokuda's sequence (2.25). The following data was obtained:

Shell sort time with `gap` = 2: 173 ms
Shell sort time with `gap` = 2.25: 175 ms

These times seemed very similar, so I decided to see if the Tokuda gap is considered to be "special" because of a good best case runtime. So, I compared best case runtimes of shell sort with a `gap` = 2 and `gap` = 2.25:

Shell sort time with `gap` = 2: 3327ms
Shell sort time with `gap` = 2.25: 2317ms

Shell sort that uses the Tokuda sequence is about 30% faster than "regular" shell sort, so I think that the conclusion that the Tokuda sequence has a good best case runtime is valid.