Lab 04

**Adam Rich**

Section EN.605.202.87

Spring 2018

May 8, 2018

# Summary

For a quick overview of the results, this shows the "nanoseconds per number" for the five methods prescribed by the lab, using the provided input files.



qa-100-1 = iterative quick sort using array, smallest partition = 100, first element method

qa-2-1 = iterative quick sort using array, smallest partition = 2, first element method

qa-2-2 = iterative quick sort using array, smallest partition = 2, median-of-three

qa-50-1 = iterative quick sort using array, smallest partition = 50, first element method

For ascending or reversed data, the quick sort with first element method performs really badly. This makes sense because the worst case for the quick sort is O(N^2) when the pivot is either a min or max in the partition. However, the quick sort median-of-three method works well here.

Heap sort has more overhead than the quicksort, it appears. And, this is because of the bubbling up and down required to keep the max heap part of the array well-formed. However, since the maximum number of operations per index is a function of log N as N gets bigger this overhead goes away.

**So what would I choose?**

For very large datasets where it is unknown whether the data is going to be pre-sorted or random, I would choose a heap sort. For smaller data sets, the quick sort with the median-of-three method works well, although I suspect it would perform even better with a higher threshold for breaking over to the insertion sort.

# Method

**Iterative or Recursive?**

I wrote QuickSort first and did it recursively. I gave up the mental exercise of how to write Heap Sort recursively and instead did some Googling to get some help on an iterative QuickSort. I was hoping to find a QuickSort implementation that did not use a stack or recursion (using a stack just felt like cheating). However, I was unable to find one. Well, there is this, but the authors have not yet responded to my request for the full-text:

<https://www.researchgate.net/publication/220975876_Quicksort_Without_a_Stack>

All of the non-recursive examples of QuickSort that I saw (about a dozen in total) worked the same way, essentially. The one that I studied the most, and based the "array for stack" part of the implementation on can be found here:

<http://alienryderflex.com/quicksort/>

**QuickSort Implementation**

I ended up with three different implementations of QuickSort:

* Recursive
* Iterative, using the Stack class from Lab 1
* Iterative, using an array acting as a stack

Each of these also has the four versions, as required, for a total of 12 QuickSort outputs per input file.

**HeapSort Implementation**

My heap sort follows the example shown in the lectures. It does a forward pass and a backward pass in place in the array. All iterative . . . I could not get my head around how this would be done recursively.

**Input and Output Files**

I did not create my own input files, but just used the ones provided. (Not enough time!) I did clean up the formatting a bit to have one number on each line, but my code still works with any n0n-numeric delimiters.

There are 9 or 13 output files per input file. The reason there are 9 for some is that recursion achieved "stack overflow" on n = 5000, 10000, and 20000 for pre-sorted files in either ascending or descending order.

The following suffixes were used to keep track of the different output files for each input file:

* heap
* qa-100-1
* qa-2-1
* qa-2-2
* qa-50-1
* qr-100-1
* qr-2-1
* qr-2-2
* qr-50-1
* qs-100-1
* qs-2-1
* qs-2-2
* qs-50-1

qa = iterative quick sort using array

qs = iterative quick sort using stack

qr = recursive quick sort

The first number is the code gives the "insertionAt" value – for partitions with lengths below that value insertion sort kicks in. The second number in the code is the method for selecting the pivot value. 1 = first number in array, 2 = median of first, mid, and last elements in array.

I didn't actually write 12 different QuickSort functions, but instead just have the three and got the four "versions" prescribed in the lab using parameters.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Method ID | Lab Description | initPivotMethod | insertionAt | suffix for output file |
| 1 | Select the first item of the partition as the pivot. Treat a partition of size one and two as a stopping case. | 1 = "first" | 2, i.e. for partitions of length 3 or more, the partition will be processed using the "partition exchange sort" methods, for 2 or less switch to insertion sort | 2-1 |
| 2 | Select the first item of the partition as the pivot. Process down to a stopping case of a partition of size k = 100. Use an insertion sort to finish. | 1 = "first" | 100 | 100-1 |
| 3 | Select the first item of the partition as the pivot. Process down to a stopping case of a partition of size k = 50. Use an insertion sort to finish. | 1 = "first" | 50 | 50-1 |
| 4 | Select the median-of-three as the pivot. Treat a partition of size one and two as a stopping case. | 2 = "median of three" | 2 | 2-2 |
| 5 | HeapSort | N/A | N/A | heap |

**Pivot Selection Methods**

When "median of three" is the pivot selection method the QuickSort finds the median of the values at the two ends of the partition and the midpoint and swaps that with the value in the first position of the partition. So, the sort can then continue as if the pivot is always in the first position.

# More Analysis

**QuickSort Implementations**



For fun, and because I wrote the three different quick sort implementations, I can compare them.

The only difference between qs (iterative using stack) and qa (iterative using array) is that qs creates a Stack data structure from a class definition and calls its push and pop methods to keep track of the partitions to sort. qa just uses an internal array for the purpose. You can see very clearly that qs has overhead in creating the stack. However, I am surprised at the differences for qs and qa for ascending and reverse data for n = 5000+. qa is supposed to be faster because it is not utilizing the system stack to do function calls...

qr has great performance until it doesn't work! This observation with those in the last paragraph makes me wonder what kind of optimization the system stack is using because it appears to be possibly giving gains to both qs and qr, over simply assigning numbers to an array, as is done in qa. All three are essentially variations on the same theme. Unlike the Towers of Hanoi lab where the recursive and iterative solutions were dramatically different, these are essentially the same, just using different "stacks" to keep track of where to go next.



This table is an attempt to rank the methods for n = 1000+ using results for random data only. I am still surprised at how well the recursive one appears to be performing versus the others, adding in an early insertion sort and more intelligent pivot selection helps even more.



This last table gives a quick comparison of the performance of "early out" insertion sort at 100 and 50. The 100 is always slower, it appears.

**Take Aways**

This lab has made it very clear that there is no one "best sort". I suspect that depending on the language, the application, the size of the data, the efficiency of relied upon code, the presorting or not of the data, and many other factors will all work to determine the efficiency of a sorting method and implementation. Experimenting like this has been a good learning experience, and insightful.

# Other Notes

**Helper Classes**

*ArrayStringParser*

Takes a string that contains numbers delimited by anything other than numbers and splits it into an array.

*IntegerSorter*

This is the class where all the sorting logic sits. On init, it gets a pointer to the array and makes two internal copies of it. The first, "original", is never touched. The second is the copy that gets sorted in place. In between each sort it gets reset to the original state.

*StackInteger*

The stack class that I wrote for Lab 1.

*Util*

A static class of functions that just make sense to have sectioned off somewhere, like "stringToFile", "fileToString", "arrayToString", etc.

**Attributions**

I wrote all my code for the sorts myself, i.e. no copy and paste, or copy and re-write, but I did use the idea of an array acting as a stack from this website:

<http://alienryderflex.com/quicksort/>