605.202: Introduction to Data Structures

Christine Herlihy

Lab #4: Analysis Paper

Due Date: 12/8/15

Dated Turned In: 12/8/15

Lab #4 Analysis

I. General Commentary

This project included four primary activities: (1) writing a function that could generate and read-in each test case file, in accordance with project specifications; (2) writing each of the specified search algorithms (i.e. iterative quicksort, along with various permutations; iterative heap sort); (3) comparing the run time of each algorithm on each of the input files, to assess efficiency/performance of each given the order and distribution of the underlying data; and (4) aggregating the results (including plots) in a clean and easy-to-read format.

I thought that having to prompt the user for each input file would be tedious, and could be more efficiently carried out by a simple generating function call from within the main method. Thus, I wrote a method to generate test case txt files based on a size and type parameter (i.e., file size of 50 integers in ascending order with no duplicates). I included each of the file sizes and orders required by the project (i.e. 50, 100, 1000, 2000, and 5000 integers; randomly ordered data, reverse-ordered data, and ascending data). I also generated file sizes of {7,000; 10,000; 13,000}, and randomly ordered data (of all file sizes) containing between 15-20% duplicates. I wrote a shuffle function to generate the random files, and used this shuffle function (along with code to splice and replace pieces of the input array) to generate the files with duplicates.

I designed separate classes for each type of algorithm (i.e., quicksort and heap sort), and also designed LinkedStack and StackNode classes for implementing the iterative version of quicksort. My main program driver read in each generated input case txt file and—for each algorithm—made a copy of the input array and sorted the array using each algorithm, printing the sorted array and running time to an output file. I used this running time data to compare the efficiency of each algorithm for the various file sizes and types. To improve the granularity of my results, I calculated runtime in seconds and nanoseconds.

II. Justification for Design Decisions

I opted to implement my primary quicksort method (in which the first item of the partition is selected as the pivot) iteratively. To do so, I created my own StackNode and LinkedStack classes, and was thus able to pop off and push on array index values as I iterated through the original input array, subset this array into sub-arrays, and sorted these sub-arrays (and their component pieces, up until partition sizes of 1 or 2). At this point, the sub-array was trivially sorted (in the case of partition size=1), or could be easily sorted by use of the swap function I wrote to interchange two array elements if element a > element b, and array[index of a] < array[index of b] (in the case of partition size=2). I used an array-based representation of a max heap to iteratively implement the heap sort algorithm. I made my primary implementations iterative rather than recursive so that I could compare the performance of each without having to consider the overheard of recursion; however, this overhead does factor into the analysis when each of the variations of quicksort (which use recursion down to their respective specified stopping cases) are compared to the iterative implementation.

III. Lessons Learned

Working on this project helped me to better understand the different ways in which the quicksort algorithm can be implemented and made more efficient. It was also interesting to me to see how recursion could be made more efficient when used in combination with other sorting algorithms (i.e. insertion sort for smaller array values when implementing quicksort recursively). In attempting to run my sorting algorithms on file sizes > 5,000 from the command line, I also encountered stack overflow issues associated with the JVM itself, due to the overhead costs of using the recursive implementations of quicksort on such large arrays (particularly on the test cases that were already in ascending order). I was able to solve these problem for arrays of size 10,000 and 15,000 by allowing the JVM to use more memory from the Terminal window, although an input array of size 20,000 still proved problematic. I

had not previously encountered this sort of problem, and it reinforced to me the importance of writing code that is efficient not only from a time perspective, but from a space perspective as well. There are undoubtedly other ways I could have optimized my code to reduce its memory usage, and I would hope to improve on this aspect of my design in future iterations.

IV. Alterations to Consider in Future Iterations

Additionally, in future iterations of this project, it could be interesting to explore alternate means of selecting the pivot for quicksort—for example, instead of selecting the first item in the array, or the median of three elements, you could randomize the pivot selection process. Another possible modification could be to do a check on a small subset of the data to see if it is ordered, and, if it is, to shuffle the entire input array to ensure optimal performance (and avoid the worst-case performance associated with an in-order input array) when using quicksort. I would have liked to integrate a graphical component within my Java code, but was not familiar enough with the data visualization options in Java to do this efficiently; I would hope to integrate improved visualizations (both of the sorting process, as well as the results) in future iterations.

V. Issues of Efficiency

With respect to the comparative performance of the two sorts (i.e. quicksort versus heap sort), the most efficient choice of algorithm will depend on the file size, as well as on the order (or lack thereof), of the input file. In general terms, quicksort using the first item as the partition performs best on randomly ordered data (best and average cost of $O(n \log n)$) and worst on pre-sorted (i.e., in-order) data (worst case cost of $O(n^2)$). Heap sort, on the other hand, offers $O(n \log n)$ performance regardless of the order of the input data set, and because it is implemented iteratively here, avoids the high overhead of recursion with larger arrays.

The results of each algorithm for each input file allow me to highlight a few key takeaways. (1) The performance of the recursive quicksort functions (i.e. with stopping cases k=50 and k=100) deteriorated as the file sizes increased due to the overhead costs associated with recursion. Insertion sort for smaller array sizes helped to mitigate these effects, but the other algorithms still outperformed these recursive implementations. (2) Among the quicksort algorithms, quicksort with median of three as the pivot outperformed quicksort with first item as the pivot. (3) The quicksort algorithms as a group performed best on the random and duplicate input files, and worst on the ascending input files. Heap sort performed equally well across file types, and performance deteriorated only marginally as file size increased, due to the lack of recursive function calls. (4) Across the quicksort algorithm types and file sizes, performance was best on the duplicates input files, because fewer comparisons are required, since an item that is of equal value as the partition can go on either side of it.

In conclusion, holding size constant, the order of the data has the greatest impact on time efficiency vis-à-vis algorithm selection—if we know that the data is in order, for example, any form of quicksort would be a poor choice, and a heap sort would be preferred. If the data is randomly ordered, then we may be equally well off with quicksort or heapsort. With respect to space efficiency, beyond a particular size threshold (which will be a function of the memory constraints of the JVM), file size has the most immediate and negative impact on efficiency, and recursive sorting algorithms will be particularly ill-advised and inefficient.