**Quicksort Sorting Algorithm Analysis**

Thomas C. Lindley

Department of Computer Science, University 1f Maryland

CMSC 451

December 8, 2021

Professor Romerl Elizes

INTRO

1. QuickSort (3 sentence/ bullet, pseudo code)

QuickSort takes a divide and conquer approach to sorting arrays. QuickSort divides and conquers an array it is sorting by choosing and element within the starting array as a pivot point, then partitions the starting array into two partitions. One of the partitions contains the elements to the left of the pivot point and the other contains the elements on the right side of the pivot point. The pivot point used can be chosen in a variety of ways, like choosing a random element, or choosing the median element as the pivot. (*“QuickSort*,” 2021)

1. Big O analysis of the two versions of algorithm

The Big O time complexity of both the recursive and the iterative version of this algorithm are the same, which is both having a worst case of n^2 and an average of n Log n. This happens here because they both rely on the same partition method. The only difference here is that the recursive method recursively calls itself to get through sorting the entire array, while the iterative method uses a stack to do the same sort.

Looking at the graph of execution times for both implementations we can see that the recursive implementation is consistently faster than the iterative implementation. However, the difference in execution is negligible, since we are seeing a max difference of ~0.06 milliseconds. We can also see from the graphs that both implementations are running withing the expected time complexity of worst case big-O (n^2) and average case of big-O (n log n). On a much larger scale this difference could be noticeable, especially since the iterative implementation seems to hit it worst case time complexity more often than the recursive method.

The iterative implementation expressing its worst-case execution time more often than the recursive could be caused by hardware. Both implementations eventually use a stack, the iterative version is straightforward and shown in the code. But, with recursion the JVM uses a stack frame to store the recursion in a stack. So, I believe there is some difference of how the JVM tells the hardware to handle the stacks. Maybe a stack that is created for recursive methods is given its own place on hardware, like its own thread, while a stack data structure is placed on a shared thread and stack area created by the JVM. (“*Java Virtual Machine”, 2021)*

1. Approach to avoiding JVM warm-up (10 sentences)

JVM warm-up is the process of caching classes that are not already loaded when a JVM process begins. When a new JVM process begins, an instance of ClassLoader loads only necessary classes are into system memory. This process loads the necessary classes in three phases, during these phases only essential classes, JAR files, and classes that are in the application’s class path are loaded. Items and classes not cached by ClassLoader are left to being loaded in on an as-needed basis. This results in a slower loading time of those classes left out by ClassLoader, which for our benchmarking can disrupt results or can lead to less responsive applications. (“*Java-JVM-Warmup*,” 2019)

I manually warmed up the JVM for my benchmarking application by creating thousands of instances of the class in the program responsible for running the sorting algorithms. This was done through a static method in the main java file, and that method is called within main, to ensure the sorting algorithm class is warmed up immediately after the start of the program.

I believe that for this benchmarking program, warming up the JVM resulted in a negligible difference. I could not tell if my warmup code was working when testing the program with or without it. This was because the difference was minute, maybe 100s of nanoseconds. Such a small-time difference is not perceivable in practice and difficult to notice in data tables.

However, there are applications where JVM warm-up is critical. These applications involve things like servers that must be low latency and that server large quantities of traffic. In a server that must be low latency that serves high traffic, not warming up the JVM can cause excessive latency and timeouts, that could leave customers and users unhappy with the server’s services. (Haudegand, 2018)

It is also important to keep in mind that warming up the JVM helps latency by getting necessary data for classes into the CPU’s cache. (Haudegand, 2018) I currently run a machine with the latest AMD consumer grade processor, but someone with a older processor with a slower or less cache could notice a more extreme difference between using and not using JVM warm up.

1. Critical Operations

I chose to put the critical operation count increaser for both the iterative and recursive implementation in each implementations respective main driver method (iterativeQuickSort() and recursiveQuickSort()). In the recursive implementation I put the critical operation count increase just before the call to partition. For iterative, the critical operation count increase is located within the while loop that iterates through the stack. It would be presumably more accurate to put the critical operation count increase in the partition method, but this would not make the comparison of critical operations different in a substantial way. I say there would be no significant difference, because both implementations must call the partition method. So, if there was a drastic difference in the number of critical operations for each implementation, we would also see it at the main driving methods level, because it would call partition a significant number of times more than the other implementation.

ANALYSIS

1. Graph critical operations:

Single Machine:

Multiple machine results:

Chart, line chart

Description automatically generated

1. Graph execution times

Single Machine:

Multiple Machine Results:

Chart, line chart

Description automatically generated

1. Performance comparison

There is a difference between both implementations of QuickSort’s execution times, which was surprising to me and made me think that there may be an issue with my data. However, after looking at it, everything seems right and I realize that the graphs make the execution time differences look drastic, but they are representing minute amounts of time on the y axis.

I believe the performance difference comes down to the way the JVM handles stacks versus the way it handles recursive functions, even though recursion uses a stack behind the scenes. I think this time difference has to do with memory allocation, memory deallocation, and CPU scheduling. I say this, because the only operations being done on the stack for the iterative version are creating it, push, and pop which all should have a negligible time complexity of O(1). I almost thought this could be because creating a stack may be inefficient, but then I realized if creating a stack is what causes the iterative implementation to be slower, then we should see a steady increase in avg time to sort increase as the array sizes get larger, which we do not.

I ran the benchmark and gathered results from three separate machines which include:

* AMD 5800x 8 core, 16 thread CPU – turbo 4.9 GHz, 32 GB RAM @ 3600 MT/s, Windows 10 latest update;
* Intel 10750h 6 core, 12 thread (laptop) CPU – turbo 5 GHz, 32GB RAM @ 3200 MT/s, Windows 10 latest update.
* AMD 3600 6 core, 12 thread CPU – turbo 4.3 GHZ, 15GB RAM @ 3200 MT/s, Manjaro KDE Plasma 21.1.6, kernel 5.15

What I find interesting from running the benchmarks on different machines and operating systems, is that all machines have almost identical execution times for the recursive implementation of the QuickSort algorithm. However, on the less powerful machines, intel 10750h running windows and AMD 3600 running Manjaro, the iterative versions execution times exceeds a 1 millisecond time difference from the recursive version.

There are two major differences between all three machines this test was run on. One of the major differences is that the machine that ran the benchmark the best has more cores and more threads. The other major difference is that the machine that ran the best is on a new processor that has the latest improvements in CPU technology, while the other two slower computers have older processors.

Two hypotheses can be derived from this information. One hypothesis is that the way the JVM handles the iterative version of this algorithm is sensitive to the number of cores and threads a CPU has. The other hypothesis is that the way the JVM handles the iterative version runs faster on newer processor technology.

1. Comparison critical operation and execution time

Both the implementations of QuickSort have close critical operations counts. I expected this, because both implementations are relatively similar. This tells us that neither implementation must use the partition method significantly more than the other, which should be the case as both methods have the same big-O time complexity.

The execution times of both implementations seems to differ drastically with the given data set sizes, however a difference of 0.06 milliseconds, at max, is really not that drastic. It seems that the iterative method, if it continues the pattern of seemingly random spikes in avg time, could be unpredictable with verry large data sets. In contrast, the recursive implementation is always at least 0.02 milliseconds faster than the iterative method and has an almost linear progression when looking at its avg time to number of elements in an array. This consistency means we can make reliable predictions of how long the recursive QuickSort sorting algorithm would take to sort an array of any size. However, with the iterative QuickSort, we could not rely on any predictions on how long it will take to sort an array of any size, because what if that array size is the size that causes it’s time to complete to relatively drastically increase and then what if that array size is what the program mostly ends up using.

A good point is made from the data represented in these graphs and it is that if we are trying to get the most optimal solutions in programming it is not always just comparing Big O and critical counts. There are other things at play within a system like memory management, CPU scheduling, etc. that can affect performance of algorithms that on paper should run just as well as other implementations of the same algorithm.

If we are trying to choose between using an iterative or recursive version of an algorithm, there is more to consider than just performance of the algorithms. Since recursive algorithms are more memory intensive, we must realize that given the same hardware and data, a recursive algorithm could reach a stack overflow condition much faster than an iterative algorithm. (Ahmed, 2011) In this situation, that means that we could like the fact that the recursive version of QuickSort is a bit faster, but not choose to use it because we know we will use large arrays that cause stack overflows with the recursive version.

1. Significance of coefficient of variance results and how it reflects the data sensitivity of selected algorithm (5 sentences)

The coefficient of variance shows us what is happening relative to a single point in our data. This single point, in our case, is representing what happened with all 50 data inputs that went into giving us the single point for average time or count, like at the point that represents time when the 50 arrays contained 100 elements. This is important because it tells us how close all the data is for time and critical count when arrays are of size 100. A large coefficient of variance for execution time means that within the 50 runs data that went into the average execution time, all the data was relatively far apart, in other words there was large variations from data entry to data entry.

The coefficient of variance critical count shows us that for both the iterative and recursive implementation, the data sensitivity is relatively low and as the number of elements in the arrays to be sorted increases the data sensitivity also decreases.

The coefficient of variance time shows us that for the recursive implementation, the data sensitivity is extremely high at small array sizes, but drastically decreases and stabilizes at larger array sizes. However, we see unpredictability with the iterative implementation, because the coefficient of variance spikes at array sizes of 500, but besides that remains stable.

1. Comparison of results vs big 0 analysis

Table

Description automatically generated

Just looking at the graphs for execution time it is impossible to tell what time complexity the two versions of the algorithms are running in. However, if we use Excels log function, like above, we can see what time complexity the two versions are running in. As seen in the two charts above, both algorithms are running between n log n and n^2-time complexity. Both versions are trending towards n^2-time complexity, but it seems it would take a very large array to reach n^2-time complexity.

Comparing benchmark results of our implementations of an algorithm is important, because it can show us that there is something wrong in our code. One sign that there is something wrong with out implementations would be that our results time complexity is worse than what the known big-O analysis of the algorithm said it should be. This would be something like our algorithm running in n^3 time, when the big-o analysis says it should run in n^2. Furthermore, if big-O analysis tells us that QuickSort should run in worst case of n^2 and are algorithm is always running in n^2-time complexity there may be an issue, because QuickSort should experience worst case when the array is already sorted. If it is the case that an algorithm that we implemented is not running as fast as the known big-O of that algorithm says it should, we should compare our implementations to others and see if there is discrepancies between the two.

CONCLUSION

As shown in the comparisons between the recursive and iterative implementations of the QuickSort algorithm, they both perform as expected and relatively like each other. The recursive implementation may seem to be much faster on the graph, but when we look at the context of the graphs, we see that there is at max only a 0.06 millisecond difference between it the iterative version. This could be due to the way each version is handled on the hardware level or by the JVM. Since both are similar in performance, when choosing which algorithm to use, it comes down to other issues with both versions of the QuickSort algorithm. Recursive versions of many algorithms can be simpler and clearer to users, while iterative algorithms can be longer and vaguer. However, I think the most important difference between the two versions of Quicksort, is the fact that the recursive version is more likely to cause a stack overflow condition when sorting large arrays.

We also saw differences in the two versions of QuickSort, in that the iterative version expressed the worst time complexity of n^2, while the recursive version remained in n log n time complexity. However, as mentioned before, even when the iterative version ran in the time complexity of n^2, it was still only 0.06 milliseconds slower than the recursive version.

Warming up the JVM could be critical in some applications or on certain hardware, but in my case JVM warmup did not seem to do much to the overall benchmarking of the algorithms. It would be best to analyze whether JVM warm up is needed on a per case bases and take other factors into consideration when deciding if to implement JVM warmup in code. Implementing JVM warmup may not make a difference in testing on a small-scale single machine but could have a drastic effect if used in production.

Evaluating the coefficient of variance for the execution times and critical operation counts brought to light what was happening in the data that went into creating the graphs. The coefficient of variance showed us the certain points within execution times and critical counts where the data varied greatly.

After running the benchmark on multiple systems, it became apparent that if we are developing a program for a certain machine, we should benchmark that program on the same hardware for the most accurate idea of how it will perform. This is highlighted in my data by the fact that my older systems ran the iterative version of the QuickSort algorithm differently than my newer system. Here the differences are small, around a millisecond, but the differences could be drastic on certain hardware configurations. For example, say I have a high-end setup for my work computer, running the top-of-the-line threadripper CPU with 64 cores and 128 threads, but I am developing an application for a server that has a 10-year-old CPU. The performance of algorithms and the program could be acceptable on my computer, but then when running on the old server, produce unacceptable results and then must be reworked.

REFERENCES

*Quicksort*. (2021, August 10). GeeksforGeeks. Retrieved December 9, 2021, from https://www.geeksforgeeks.org/quick-sort/.

*Java Virtual Machine (JVM) stack area*. GeeksforGeeks. (2021, June 20). Retrieved December 10, 2021, from https://www.geeksforgeeks.org/java-virtual-machine-jvm-stack-area/.

Haudegand, B. (2018, November 8). *JVM and cache warm-up strategies for high traffic services*. Medium. Retrieved December 10, 2021, from https://medium.com/teads-engineering/jvm-and-cache-warm-up-strategy-for-high-traffic-services-4b5016f8b565.

*Java-JVM-Warmup*. Get docs. (2019, November 19). Retrieved December 9, 2021, from https://en.getdocs.org/java-jvm-warmup/.

Ahmed. (2011, January 20). *Recursion vs iteration (looping) : Speed & memory comparison*. IT. Retrieved December 10, 2021, from https://eslamghanem.wordpress.com/2011/01/20/recursion-vs-iteration-looping-speed-memory-comparison/.