Comparing Asymptotic Behavior of Sorting Algorithms under Differing Conditions

Michael Pacheco, Sean Davies

University of New Mexico Engineering – CS361

Introduction:

We are experimentally designing and testing three sorting algorithms and reporting their performance benchmarking for the $input\ size\ n = 2^{20} \leq n \leq 2^{30}$ of both numeric integers, $int,$ and Double-precision floating-points, $double$. Java is suitable for testing and execution and less suitable for analysis and data visualization, so Python is used for this to have a well-realized cohesive report. Lastly, the sorting algorithms were perfromed on an `i7-8750H 2.2GHz` Laptop, and the analysis and data visualization were performed on an `i7-6600u 2.8GHz` Laptop.

Preliminaries:

Overview

<u>$QuadHeap$</u>

Heap sort is a sorting algorithm that builds and extracts from a heap, then $heapifies$ the remaining values. The root node contains the maximum or minimum value of its subtree. Our algorithm, $QuadHeap$ is a modified version of $MaxHeap$ with four children at each parent node instead of the standard two.

For the heap construction, in the original array of the datatypes the initial tree is built starting at index $= n$, the children are placed from left to right and up until the root is finally added at index $= 0$. The children are compared to the root node and swapped if larger then $heapify$ is called recursively to maintain the heap. After the heap is built, we extract the root node, the maximum value, and swap it with the last element of the array then $heapify$ the remaining heap. We continue this until the array is sorted.

<u>$3 - Way\ Merge\ Sort$</u>

Merge sort is a recursive sorting algorithm that divides an array of numbers into two subarrays until there are only single element arrays left. Then the subarrays are recursively recombined in order until finally the array is sorted.

Our 3-way merge sort implements three subarrays instead of two. We divide the array into thirds until single element subarrays are left. The subarrays are contained in temporary arrays, and we recombine by comparing the smallest elements of each one into the original array until the entire list is sorted.

Overview (Continued)

$\underline{Randomized\ Quick\ Sort}$

Quick sort is a recursive algorithm that finds a pivot element in an array of numbers and then rearranges so that all values greater than it are after its original index and all values less than it are before it. We continue this recursively by creating two partitions and rearranging them until we have sorted the entire array. The Randomized Quick Sort will be different only in that it determines the pivot randomly instead of finding a median value pivot.

Expected Asymptotic Bounds

$\underline{3 - Way\ Merge\ Sort}$

First, there are $3T$ number of sorting calls, $\left(\frac{n}{3}\right)$ number of subproblems we are calling on,

and $f(n) = O(n)$ is the merge time so the recurrence relation is $T(n) = 3T\left(\frac{n}{3}\right) + O(n)$.

Now we solve it using the Master Theorem,

$$T(n) = aT\left(\frac{n}{b}\right) + O(n^c),\ \text{where}\ a = 3, b = 3, and\ c = 1$$

In our table,

$$\text{If, } c < \log_b a\,, then\ T(n) = \Theta\left(n^{\log_b a}\right)$$

$$\text{If, } c = \log_b a\,, then\ T(n) = \Theta\left(n^{\log_b a} \log n\right)$$

$$\text{If, } c > \log_b a\,, then\ T(n) = \Theta\left(f(n)\right)$$

The second case is matched for $c = \log_3 3$, so then $T(n) = \Theta\left(n^{\log_3 3} \cdot \log n\right) \rightarrow \Theta(n^1 \cdot \log n)$.

From this the expected asymptotic bound is $\Theta(n \cdot \log n)$.

$\underline{Randomized\ Quick\ Sort}$

In the $Randomized\ Quick\ Sort$ algorithm, there is one randomized pivot causing on average an equal splitting into two partitions. For the recurrence relation there will be $\frac{n}{2}$ partitions, with $2T$ recursive calls, and $\mathrm{O}(n)$ repositioning around the pivot, so $T(n) = 2T(n/2) + \mathrm{O}(n)$.

The master theorem can be used again to find the time complexity. We can quickly say that,

$$a = 2,\ b = 2,\ c = \log_2 2 = 1\ where\ c = \log_b a\,, then\ T(n) = \Theta\left(n^{\log_b a} \log n\right).$$

$$\text{Then, } T(n) = \Theta\left(n^{\log_2 2} \cdot \log n\right).$$

So, the expected asymptotic bound is $\Theta(n \cdot \log n)$.

Expected Asymptotic Bounds (Continued)

<u>$QuadHeap$</u>

For the level, i, there are 4^i leaves so that the total number of leaves is $1 + 4^0 + 4^1 + 4^2 \ldots. +4^i$. This is a geometric series that defines a relationship between the total number of leaves n, the number of leaves at each level 4^i, and critically the height of the heap i.

For geometric series formula, $n = \left(\frac{1(4^i-1)}{4-1}\right)$, we solve for i.

$$n = \frac{4^i - 1}{3} \rightarrow 3n + 1 = 4^i$$

$$\log_4(3n + 1) = \log_4 4^i \rightarrow \log_4(3n + 1) = i\ levels.$$

Or the height, $i = \log_4(3n + 1)$.

The height of heaps represents the expected worst case time complexity to perform important heap operations. Because there will be $n\ heapify$ operations after extracting in constant time, $O(1)$, the expected asymptotic bound is $O(n \cdot \log n)$

Performance Analysis & Discussion:

Methods of Analysis

<u>Asymptotic bounds:</u> For all of the algorithms the expected asympotic bound was $O(n \cdot \log n)$. Asymptotic analysis provides insight into the rate of change of time to complete output with an increase in the size of the input, so to simplify and understand the data we use logarithms scales. Logarithmic scales will allow the visualization and comparison of our expected time complexity and measured time complexity to be simplied to linear comparisons.

<u>Linear Regression & Theoretical Slope:</u> The linear slopes of the sorting algorithms were determined by linear regression of the experimental runtimes. Linear regression provides a more accurate slope reducing the error caused by outliers. For the slope of $n \cdot \log n$, a logarithmic version of the slope formula is used, $m = \frac{\log y_2 - \log y_1}{\log x_n - \log x_1}, where\ y_{1,2} = n \log n\,, of\ both\ x_1, x_n$. Since the expected runtimes were stable because they were determined theoretically from $f(n) = n \cdot \log n$, the slope would also be stable enough for a simple slope.

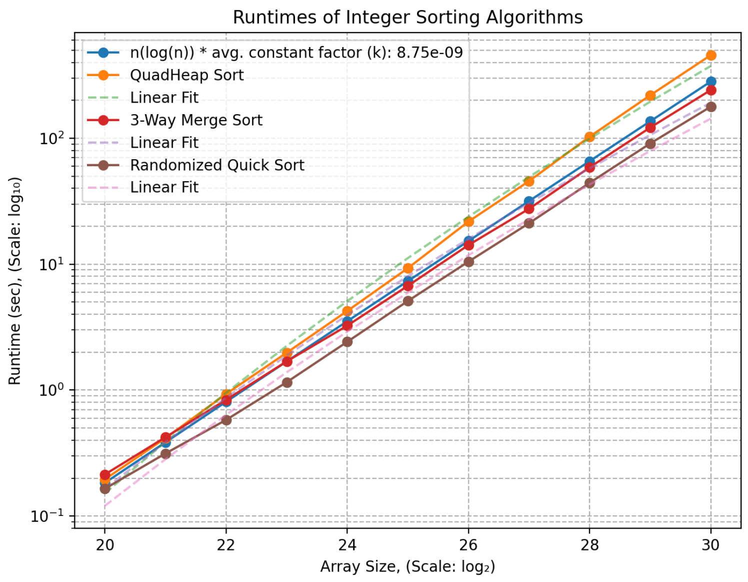
<u>Comparing Expected vs Experimental Runtimes:</u> After calculating the slopes of the sorting algorithms and the expected slope we can compare. Comparison is made by looking at both the relative and absolute distance of the algorithm outputs against the $n \cdot \log n$. Here the calcuations are made from the general relative and absolute errors, however here we are not trying to achieve the expected slope only to see if we accurately predicted it as a bound.

Methods of Analysis (Continued)

Constant factors: Regardless of the accuracy between the expected and experimental slopes the real outputs of both will vary by a constant factor caused by real world circumstances or theoretical simplifications such as the unit representation of the runtimes being in milliseconds or seconds. There exists a constant factor between each expected and experimental runtime of every algorithm so that $\frac{Experimental\ Runtimes}{Expected\ Runtimes} = k$. Simplification is needed since there are multiple expected and experimental runtimes. The simplification of the constant factor is computed as the average for each of the algorithms so that now, $\frac{(Experimental\ Runtimes)_{avg}}{(Expected\ Runtimes)_{avg}} = k_{avg}$. For visualization each algorithm's averaged constant factor is averaged again to allow the reader to compare the expected and experimental slopes.

Visualization: For the x-axis of the plots are changed since the input sizes are defined as $2^{20}\ to\ 2^{30}$ which simplifies to $20 - 30$ when displaying using $\log_2 n$, this will simplify and create clear relationships of scale. For the y-axis, we simply convert to $\log_{10} n$ scale since the functions were bounded logarithmically.

Results & Discussion



Graph 1: Time complexity of sorting algorithms of n integers

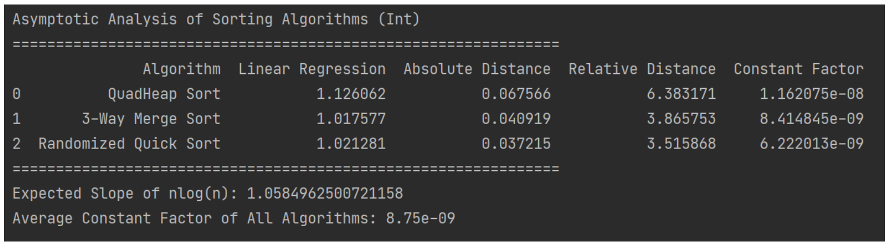
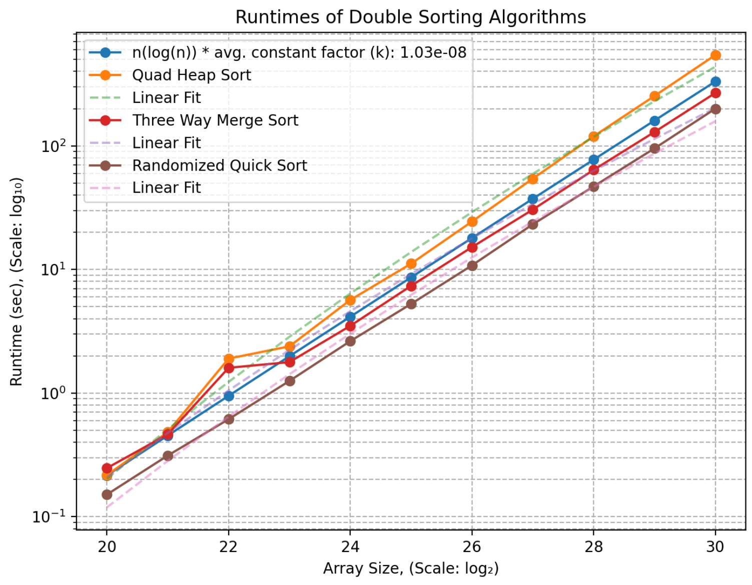
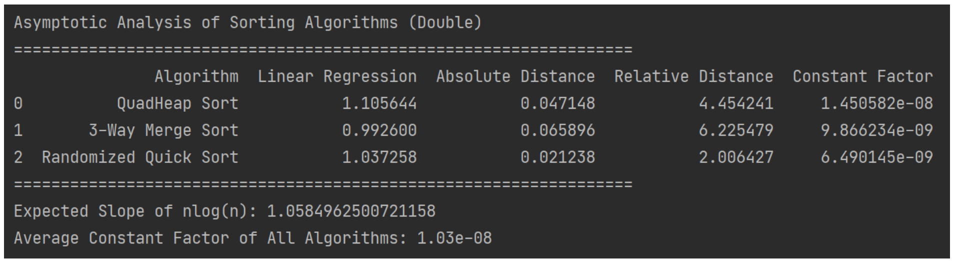

```
Asymptotic Analysis of Sorting Algorithms (Int)
==============================================================
               Algorithm  Linear Regression  Absolute Distance  Relative Distance  Constant Factor
0          QuadHeap Sort           1.126062           0.067566           6.383171     1.162075e-08
1       3-Way Merge Sort           1.017577           0.040919           3.865753     8.414845e-09
2  Randomized Quick Sort           1.021281           0.037215           3.515868     6.222013e-09
==============================================================
Expected Slope of nlog(n): 1.0584962500721158
Average Constant Factor of All Algorithms: 8.75e-09
```

Table & Analysis 1: Asymptotic Analysis of Integer Datatype Sorting Algorithms

Results & Discussion (Continued)



Graph 2: Time complexity of sorting algorithms of n doubles



```
Asymptotic Analysis of Sorting Algorithms (Double)
================================================================
             Algorithm  Linear Regression  Absolute Distance  Relative Distance  Constant Factor
0         QuadHeap Sort           1.105644           0.047148           4.454241     1.450582e-08
1      3-Way Merge Sort           0.992600           0.065896           6.225479     9.866234e-09
2  Randomized Quick Sort           1.037258           0.021238           2.006427     6.490145e-09
================================================================
Expected Slope of nlog(n): 1.0584962500721158
Average Constant Factor of All Algorithms: 1.03e-08
```

Table & Analysis 2: Asymptotic Analysis of Double-precision Floating Point Datatype Sorting Algorithms

Discussion: The asymptotic running times of the algorithms are close to their actual performance from $table$ 1, $Quad\ Heap\ Sort, 3 - way\ Merge\ Sort, \&\ Randomized\ Quick\ Sort$, relative distances are for Integers are $6.383\%, 3.87\%, \&\ 3.52\%$ away from the expected slope of $n \cdot \log n \approx 1.06$. And for Doubles are $4.45\%, 6.23\%, 2.01\%$ away, respectively, from the expected slope. All three programs are similarly complex in implementation and debugging since they're implementations of $divide - and - conquer$ strategies for sorting. From our experimental results we can see that the fastest algorithm is $Randomized\ Quick\ Sort$, with a slope for both Integers and Doubles of $1.021281, and\ 1.037258$ respectively.

Conclusion:

From the implementations of the three variant sorting algorithms the fundamentals of asymptotic predictions and analysis are supported by the results. Regardless of the complications that arise between theoretical and experimental time complexities, such as from constant factors, we are able to relatively predict the growth rate of the time to sort and the size of arrays to sort for these algorithms.

Bibliography:

Unknown, “Master Theorem,” programiz.com,

https://www.programiz.com/dsa/master-theorem

Various Authors, “Master theorem (analysis of algorithms),” Wikipedia,

https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)#See_also

Andreas Klappenecker, “Deterministic and Randomized Quicksort,” Texas A&M University,

https://people.engr.tamu.edu/andreas-klappenecker/csce411-s15/csce411-random3.pdf

Various Authors, “$K - ary$ Heap,” Geeksforgeeks.org,

https://www.geeksforgeeks.org/k-ary-heap/#

Various Authors, “Merge Sort – Data Structure and Algorithms Tutorials,” Geeksforgeeks.org,

https://www.geeksforgeeks.org/merge-sort/#

Various Authors, “Quicksort,” Wikipedia,

https://en.wikipedia.org/wiki/Quicksort

R. Horan & M. Lavelle, “Log-Log Plot,” The University of Plymouth,

https://www.reading.ac.uk/AcaDepts/sp/PPLATO/imp/interactive%20mathematics/loglog2.pdf

Various Authors, “Log-log Plot,” Wikipedia,
https://en.wikipedia.org/wiki/Log%E2%80%93log_plot#:~:text=where%20m%20%3D%20k%20is%20the,value%20corresponding%20to%20x%20%3D%201.

Contributions:

Michael Pacheco performed the runtime analysis and compiled the report. Sean Davies implemented the sorting algorithms.

Acknowledgements:

I would like to express gratitude to our overseeing professor, Sean Luan.