Performance Analysis of Sorting Algorithms: Comparing Optimization Techniques on Low-End vs. High-End Hardware

Revan Murton

Falmouth University - https://github.falmouth.ac.uk/GA-Undergrad-Student-Work-24-25/comp203-rm

Abstract

This project focused on the time efficiency of different sorting algorithms (Bubble, Insert, Shell, Radix, Quick). The sorting algorithms were coded in C++, then tested for the time taken to sort varying array sizes of integers, this was done on both a high-end (desktop) and low-end (laptop) device. The sorts were optimised to improve runtime efficiency, they were then compared across both devices to determine the relative change in performance. This poster aims to determine what optimisation strategies work best on sorts and what effect hardware has on sort times and if the complexity of a sort impacts the amount of change observed. The hypothesis is that common optimisation strategies for each sort will positively effect performance and that hardware was have a substantial effect of performance.

Introduction

There are coded functiond 5 sorts that were tested, Bubble, Insert, Shell, Radix, Quick. To test these sorts they sort from lowest to highest varying sizes of array, with the values of each element of the arrays being randomised from between 1-1000. To ensure the time taken to obtain results was kept to a reasonable amount of time the spaces between array sizes increased exponentially.

For the sake of simplicity the two devices that this will be tested on are "Desktop" (WIN-VVD6KV52UM5) and "Laptop" (LAPTOP-60KH6NOL).

Device Name	Processor	Installed RAM
WIN-VVD6KV52UM5	AMD Ryzen 5 7500F 6-Core Processor @ 3.70 GHz	16.0 GB (15.7 GB usable)
LAPTOP-60KH6NOL	Intel(R) Celeron(R) N4120 CPU @ 1.10 GHz	4.00 GB (3.82 GB usable)

Table 1:System Information for Desktop and Laptop

- **Bubble Sort:**
  Bubble Sort repeatedly loops through the array, comparing each element with the one ahead of it. If the current element is larger, they swap positions. Otherwise, the sequence remains unchanged, and the sort moves to the next element. This process continues until no swaps are needed.
- **Insert Sort:**
  Insertion Sort sorts elements one at a time by comparing each new element with those to its left. If a previous element is larger, it is shifted right, making space for the new element at its correct position.
- **Shell Sort:**
  Shell Sort is an optimized version of Insertion Sort. Instead of shifting elements one step at a time, it first compares elements that are far apart using a structured gap sequence that gradually decreases.
- **Quick Sort:**
  Quick sort picks an element in the sequence to act as a partition and moves every element that is bigger to the right and conversely every smaller element to the left. It then iterates through quick sort on either side of the partition.
- **Radix Sort:**
  Radix sort iterates through every element comparing them by the value of the last digit, for example first 1->9 then 10->90, then 100->1000.
- **"Standard" Sort:**
  The standard sort for C++ uses a version of "introsort", which is comprised of three different sorts:
  -Quick sort to split the sequence into smaller arrays -Heap sort if quick sort isn't as efficient as needed
  -Insert sort to efficiently sort the now much smaller arrays

Background Review

This paper [1] proposes a bidirectional bubble Sort with modified diminishing increments (Oyelami's sort) that is a way to optimise a sort as relatively slow as bubble sort to massively increase performance by using less than 10% of the swaps compared to bubble sort even when using smaller sample sizes (256). Although an older paper (1996) this paper [2] discusses the differing gap sequencing algorithms and how they can affect the worst, average and best case performance of various sorts. This paper [3] found that insert and bubble sort worked best at smaller sample sizes (>100), with insert sort being more efficient. Additionally the paper identified quick sort as performing better as the sample size increased.

Methods

To establish a baseline, unoptimized versions of each sort in C++ were implemented to compare against optimized variants and assess their effectiveness. The optimized versions were designed for maximum speed

Both [4] and [5] helped in the process of understanding not only the differences between each sort but how to implement them. With [5] being immensely useful in understanding linear/quadratic time complexity. To compare it, a lower-end laptop and higher-end desktop were used. The tests were run the exact same for each apart from for Rubix sort, this was due to an issue with getting it to run on the laptop. I ran each sorting algorithm 10 times and averaged the results to improve accuracy while keeping data collection time manageable. The resulting dataset was processed in Python using Seaborn to generate graphs illustrating performance trends.

Bubble Sort: Quadratic

- Implemented an early termination condition to prevent unnecessary sorting.
- Developed a comb sort variant, utilizing larger gaps to enhance efficiency by reducing excessive shifting.

Insert Sort: Quadratic

- Optimized with binary search to find the insertion point in O(log n) instead of O(n).

Shell Sort: Quadratic

- Used Knuth's gap sequence (3n + 1) instead of the standard n/2, which should theoretically reduce the required number of gaps.

Quick Sort: Linear/Quadratic

- Optimized by prioritizing the smaller partition first, reducing stack usage and processing time.

Radix Sort: Linear

- Two versions use bit wise operations to enhance speed and minimize memory usage.
- One variation replaces arrays with std::vector, allowing dynamic resizing and eliminating the need for manual memory management.
- Another version uses memsort, std::max and vectorisation to maximise the efficiency of the sorting.

Results

K is the number of bits, N is the number of elements in the array

Unoptimised Sort	Desktop(μs)	Laptop(μs)	Percentage Increase %
Bitwise Optimised Radix Sort O(nk)	642.62	2976.81	363.10%
Bitwise Optimised Vector Radix Sort O(nk)	634.03	3085.58	386.60%
Bubble Comb Sort Optimised $O(n^2)$	461.57	1872.22	305.49%
Bubble Sort $O(n^2)$	54924.83	288287.14	424.70%
Bubble Sort Optimised $O(n^2)$	54456.16	287580.31	427.90%
Insertion Sort $O(n^2)$	13182.32	79102.67	500.07%
Insertion Sort Optimised $O(n^2)$	14821.51	52560.19	254.56%
Quick Sort $O(n \log n)$	351.53	1623.92	361.70%
Quick Sort Optimised$O(n \log n)$	26.33	506.33	1822.20%
Radix Sort Optimised O(nk)	399.70	2145.50	436.70%
Shell Sort $O(n^2)$	194.33	1192.95	514.05%
Shell Sort Optimised $O(n^2)$	306.34	1416.05	361.91%
Std Sort $O(n \log n)$	826.39	4366.36	428.30%

Table 2:Ranking of Sorting Algorithms by Average Sorting Time

- Bubble, Insertion, and Optimized Insertion Sort all struggled as array sizes exceeded 100. Surprisingly, Comb Bubble Sort significantly reduced the sorting time compared to the standard and optimised Bubble Sorts on both devices($461.57(\mu s) - 54924.83(\mu s)$).
- The bitwise Radix variants performed slightly worse, likely because their worst-case scenarios depend heavily on N (number of bits). In this test, N is 32, making large values slower to process with bitwise operations, causing the observed discrepancy. The vectorised version of the bitwise sort performed better than it's counterpart, this is most likely because this removed the need to create new pointers and delete the array every time the algorithm was run. The bitwise sorts also suffered less loss of efficiency on the laptop when compared with the optimised radix sort (363.10% - 436.70%), from this data we can assume slower systems work better when sorts include bitwise operations over the usual operations.

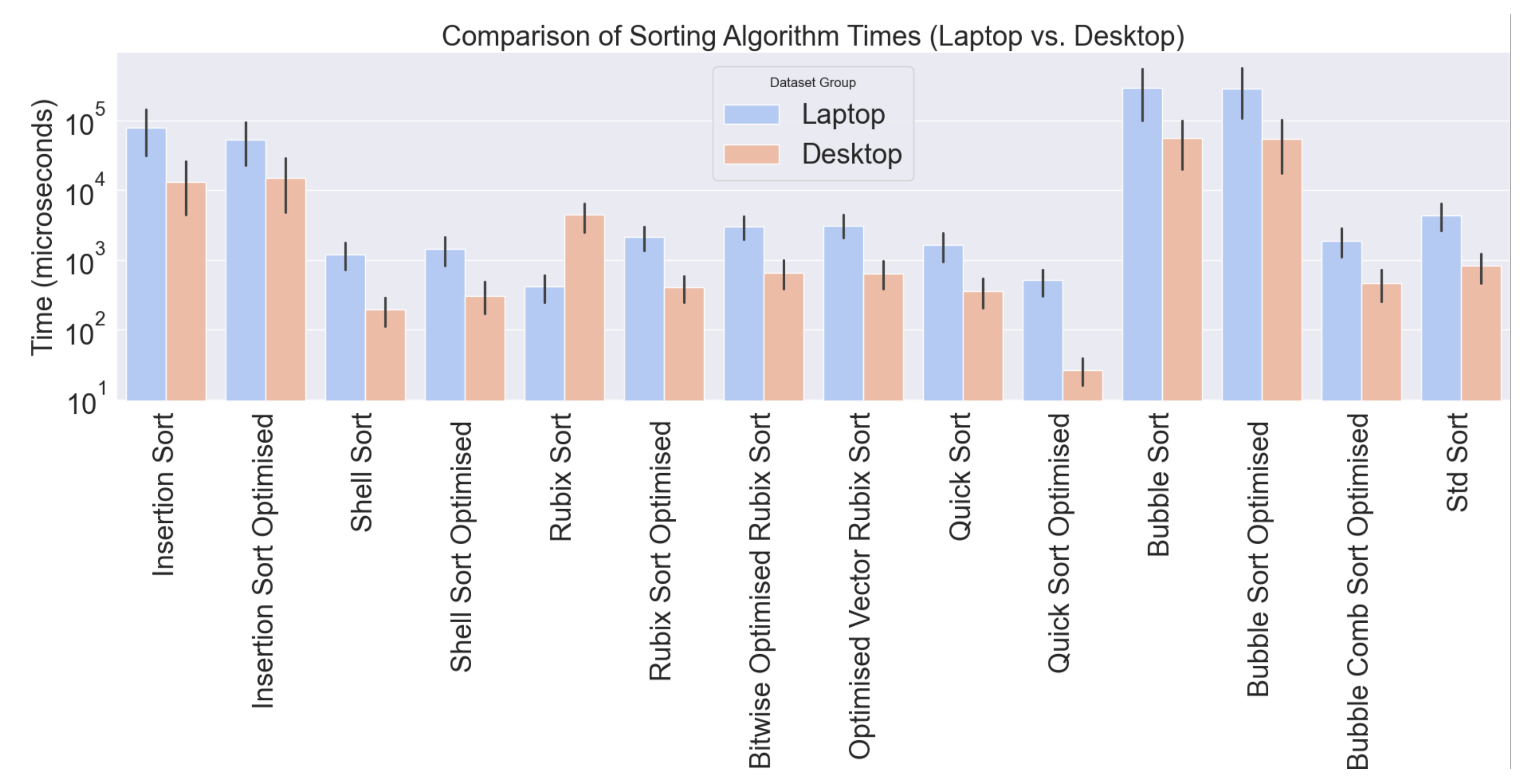

Figure 1: Line graph of time taken by various sort algorithms vs array sizes.

- The optimized version of shell sort was significantly slower. This was due to a flawed implementation of Knuth's gap sequence. Complex gap calculations work best when precomputed, but dynamic calculation was necessary due to varying array sizes, increasing sorting time.
- Unoptimized Quick Sort performed well, while the optimized variant was the best overall, consistently achieving the shortest sorting time. Surprisingly it had the largest percentage increase in time taken of any sort (1822.20%), this is most likely due to the more complex mechanisms and the nested function calls.
- Optimized Insertion Sort performed slightly worse than its unoptimized counterpart, despite using binary search, which should theoretically improve efficiency. This discrepancy likely arises because, while binary search reduces the time required to find an element's position $(O(logn) vs. O(n))$, the overall worst-case complexity remains $O(nš)$.

Conclusion

The majority of the sorts saw significant improvements in terms of performance when optimised. So common optimisation techniques when leveraged correctly can lead to significant increases in performance, such as optimised quick sort taking less than 1/10th of the time when compared to the unoptimised variant. The difference between both devices in terms of performance was relatively consistent with the optimised versions of most sorts having a lower percentage increase relative to the unoptimised sorts, suprisingly quick sort had a very large increase when tested on the lower-end device (laptop). This implies that the use of more memory intensive code such as iterative function calls impacts the device with less processing power more. Although the optimizations implemented improve performance, there remains scope for further enhancement in certain areas, particularly in refining the optimisation of specific algorithmic operations. Additionally testing the memory used by each sort would provide more evidence to support the claim that memory intensive code causes a decrease in performance on computers with worse hardware.

References

[1] '(PDF) Improving the performance of bubble sort using a modified diminishing increment sorting', ResearchGate. Accessed: May 06, 2025. [Online]. Available: https://www.researchgate.net/publication/228949866 _Improving_the_performance_of_bubble_sort_using_a_modified_diminishing_increment_sorting

[2] 'R. Sedgewick, 'Analysis of Shellsort and related algorithms', in Algorithms — ESA '96, vol. 1136, J. Diaz and M. Serna, Eds., in Lecture Notes in Computer Science, vol. 1136. , Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–11. doi: 10.1007/3-540-61680-2_42.

[3] '(PDF) Sorting Algorithms – A Comparative Study'. Accessed: May 06, 2025. [Online]. Available: https://www.researchgate.net/publication/315662067_Sorting_Algorithms_-_A_Comparative_Study

[4] [1] 'Sort Visualizer'. Accessed: May 07, 2025. [Online]. Available: https://www.sortvisualizer.com/

[5] [1] S. K. Gill, V. P. Singh, P. Sharma, and D. Kumar, 'A Comparative Study of Various Sorting Algorithms', 2018, Social Science Research Network, Rochester, NY: 3329410. Accessed: May 07, 2025. [Online]. Available: https://papers.ssrn.com/abstract=3329410