RIVIER UNIVERSITY ONLINE ACADEMIC JOURNAL, Volume 19, Number 1, Spring 2024

AN EMPIRICAL APPROACH TO SORTING: COMPARING ALGORITHM PERFORMANCE USING DIFFERENT PROGRAMMING LANGUAGES AND OS ENVIRONMENTS

Ethan Boisvert
Rivier University
Keywords: Programming, Algorithms, Sorting, Comparison, Performance

Abstract

The focus of this research revolves around comparing the efficiency of sorting algorithms and determining the most and least efficient combinations of sorting method, programming language, and OS environment used. A set of ten commonly-used algorithms are run through a series of identical sorting tests, and the graphed results yield the most and least efficient algorithms in the collection. These two algorithms are then reprogrammed in three unique programming languages, allowing the advantages and disadvantages of programming and sorting using each language in question to be observed individually. The three variants of both algorithms (six programs total) are then run individually on three different operating systems. Through the information that is gathered and graphed from the results, it becomes clear which combinations of environment, language, and sort method demonstrate the fastest and slowest sorting speeds.

1. Introduction

As technology continues to develop and the handling of data storage and organization becomes more crucial, sorting algorithms become increasingly important in the world of data management. However, there are many options to consider when taking these algorithms into account; such as which sorting approach is the most efficient, or what kind of environment the algorithm itself should run within to achieve an outcome at the optimal speed. Although there are a wide variety of sorting algorithms to choose from, they all demonstrate vastly different results in terms of processing time and overall impact on system memory. In an environment where fast and efficient data organization is necessary, it is important to know which approach to sorting will yield the most desired results.

The most common sorting techniques used today can roughly be narrowed down to a set of ten unique algorithms, all of which sort data employing different methods. However, despite all ten algorithms producing identical outputs from a general standpoint, there is a lot more that takes place in the background that ultimately dictates whether or not a particular algorithm is a “reasonable” choice. Some algorithms may sort through a set of data in a fraction of a second, while others may take several minutes. For the sake of the coming research, all ten of these algorithms will be observed individually, and a deeper dive will be taken into what results are produced when the most and least efficient algorithms from the collection are exposed to different programming languages, as well as different OS environments.

1.1. An Overview of Sorting Algorithms

Sorting algorithms are fairly self-explanatory. A single sorting algorithm contains a small amount of code; instructions that facilitate the task of analyzing a set of data, rearranging the data in a specified pattern, and outputting the final result. There are many ways data can be sorted, typically specified by the user, and can include variations such as numerical or alphabetical sorting. A dataset can be presented most optimally in the form of an array or a linked list, which allows the algorithm to more quickly scan through the contents to find a specific element. The basis for all sorting algorithms comes down to locating specific elements that follow a sequence dictated by a chosen sorting method, however the means by which these elements are obtained by the code vary significantly between algorithms.

Some algorithms may divide the search space in half and search mere portions of the data for a specific element using a sort of "divide-and-conquer" style method, such as the merge sort. Other algorithms, like the bubble sort, may scan through the full dataset in search of an element, only stopping once said element has been located. Specific sorts like the heap sort take an entirely different approach, building a full binary tree out of the elements in the set [6]. These differing methods used to obtain desired elements directly impact each algorithm's efficiency, at least from a speed-focused standpoint. By observing each different algorithm in comparison to one another, it should become clear which of these methods yields the least and most time-consuming results.

1.1. Methods and Materials

In order to properly carry out each phase of this research, a set of programs and other resources will be employed. These resources include the following:

- A Dell G7 7588 laptop which will be employed during all phases of the research. The specifications of this machine are as follows:
    - Intel Core i7-8750H 2.20GHz CPU
    - 16 GB DDR4 RAM
    - Windows 10 64-bit operating system
- A 2023 MacBook Pro laptop which will be employed during the later phases of the research. The specifications of this machine are as follows:
    - Apple M2 Max CPU
    - 32 GB RAM
    - macOS Sonoma 14.2.1 operating system
- Oracle VM VirtualBox (a program for running virtual operating systems. This will allow for the eventual execution of algorithms on the Linux OS)
- jGRASP v2.0.6_11 (a small IDE for programming in Java and C++. This will allow for the initial programming of each algorithm that will be tested during the first phase).
- Microsoft Office
    - Microsoft Word (to allow for documentation of test results).
    - Microsoft Excel (to allow for the graphing and organization of collected data).
- Spyder 3.10 (a small IDE for programming in Python. This will allow for the creation of Python-coded sorting algorithms during the later phases of the research).

- CodeBlocks v20.03 (a small IDE for programming in C++. This will allow for the creation of C++-coded sorting algorithms during the later phases of the research).

2. The Sort Comparison

The ten sorting algorithms that will be observed during the first phase of the study are as follows:

- Bubble Sort
- Bucket Sort
- Counting Sort
- Heap Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Radix Sort
- Selection Sort
- Shell Sort

These selected algorithms are among the most commonly used sorting algorithms, and therefore are optimal test subjects for the first phase of the research [2]. During the initial comparison, each algorithm will be required to sort through a large dataset containing one million numerical values. These numerical values will be determined randomly, but their initial arrangement within the dataset will be identical during each test (using a seed value) to ensure no margin of error. Each algorithm will sort through this dataset and organize the values from least to greatest, and the sorting process will be timed to determine how many milliseconds were spent sorting by the algorithm. This test will be repeated five times each for all ten algorithms, resulting in a grand total of 50 individual tests. The average sort time for each algorithm will be computed from this data and will be graphed, allowing for proper visualization of the fastest and slowest algorithms in the collection. These two algorithms will then be singled out and focused on during the remaining phases of the research.

2.1. Universal Code

Because each algorithm will be run through the same test, there are small portions of code that will be used universally for each algorithm that remain consistent despite the sorting methods themselves varying. These portions of code are outlined below.

2.1.1. Generating the Dataset

Each algorithm in the initial test will be required to sort through a dataset of one million randomized numerical values. Using the code below (written in Java), a dataset of size 1,000,000 can quickly be generated with random values throughout. The code for this is outlined here, and will remain consistent in the code for every algorithm tested over the course of the research:

```
public static int[] generateRandomArray(int size)
{
    Random random = new Random(555);
    int[] arr = new int[size];
    for (int i = 0; i < size; i++)
    {
        arr[i] = random.nextInt(size);
    }
    return arr;
}
```

Fig. 2.1: The Dataset Generation Method

The "Random" function uses a seed (the value 555) to ensure that the random numerical values to be sorted still remain consistent throughout every test. This helps to reduce the small margin of error that may arise if the values were to be generated in a genuinely different order each time. The code of this method is modified slightly for the bucket sort, however, as the sort can only work with values between 0 and 1 and therefore requires float values to work properly rather than integers.

2.1.2. Timing the Algorithms

Each algorithm in the initial test will be required to time itself to accurately determine how many milliseconds of system time were dedicated to the sorting process. Using the lines of code below (written in Java), the start and end time of an algorithm's sorting can be captured, allowing each program to calculate how much time it took for the system to finish the process:

```
long startTime = System.nanoTime();

//[Insert sorting algorithm here]

long endTime = System.nanoTime();
long elapsedTime = endTime - startTime;
```

Fig. 2.2: The Time-Capturing Code

When implementing this code into each algorithm throughout the research, it is *crucial* that the code is positioned in such a way that ensures it *only* counts the time dedicated to sorting. Outputs to the console and other background processes should *not* be included in the timing period for more accurate results overall. No files will be written to or read from by any of the algorithms; all data is stored solely in memory during each test to ensure system read and write times are not a factor.

2.2. Phase I Graphed Data

The results of the sorting test for each algorithm are outlined in the table below. For simplicity, the values displayed in each data visualization in this section have been rounded to two decimals.

Algorithm Tested	Test 1 Sorting Time (in milliseconds)	Test 2 Sorting Time (in milliseconds)	Test 3 Sorting Time (in milliseconds)	Test 4 Sorting Time (in milliseconds)	Test 5 Sorting Time (in milliseconds)	Average Sort Time (in milliseconds)
Bubble Sort	1539264.49	1530443.21	1552594.82	1549414.88	1546713.48	1543686.18
Bucket Sort	344.43	339.40	336.10	345.12	332.80	339.57
Counting Sort	31.15	34.09	29.99	32.30	31.25	31.76
Heap Sort	157.83	155.19	155.01	156.13	156.83	156.20
Insertion Sort	95920.32	96052.04	96538.08	95771.60	95870.43	96030.49
Merge Sort	174.43	165.52	163.44	165.33	200.70	173.88
Quick Sort	93.84	92.76	86.53	93.18	93.59	91.98
Radix Sort	100.06	100.85	102.10	100.41	100.44	100.77
Selection Sort	521389.56	524674.72	510583.26	529385.96	530411.58	523289.02
Shell Sort	181.43	181.71	179.49	209.76	171.06	184.69

Fig. 2.3: Phase I Test Results

Taking the average sorting times for each algorithm calculated above, the graph below was created, demonstrating which algorithms took the shortest and longest average amount of milliseconds to complete the sorting test (in relation to one another). All tests were run in the Java programming language and on the Windows 10 64-bit operating system.

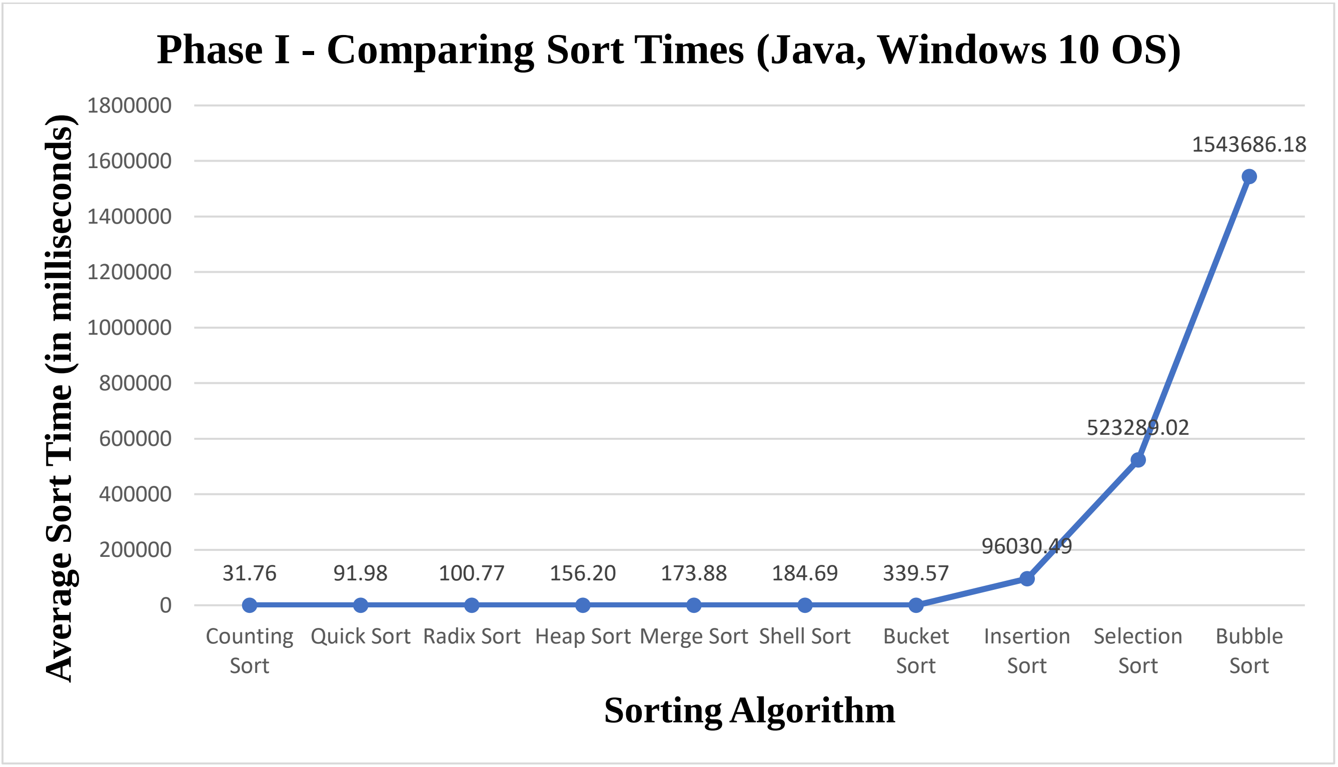

Fig. 2.4: Phase I Graphed Comparison (All Data)

As a result of the drastic difference in the sorting times taken by the insertion, selection, and bubble sorts compared to every other algorithm that was tested, a second comparison was graphed with the three algorithms in question omitted to better visualize the smaller differences in performance between the other algorithms.

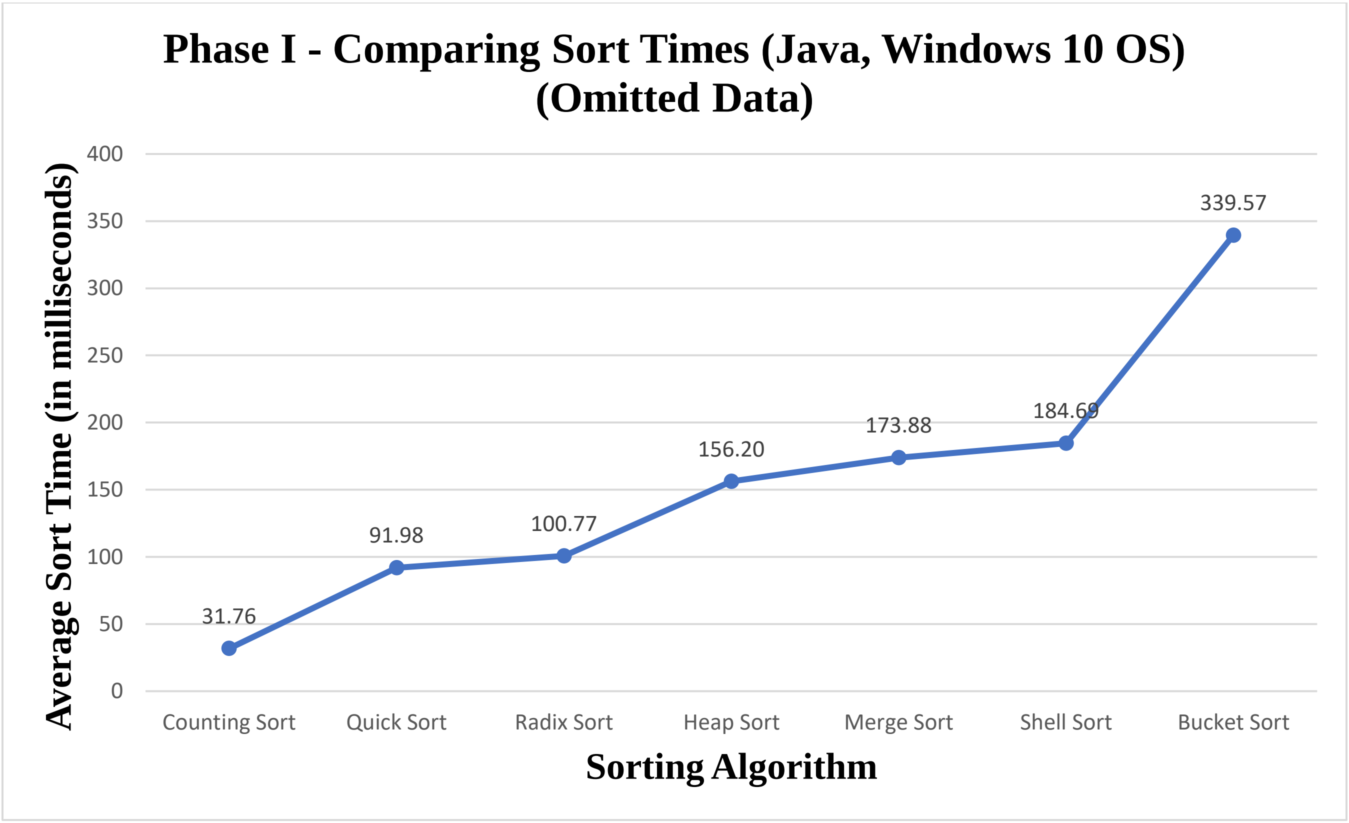

Fig. 2.5: Phase I Graphed Comparison (Omitted Data)

From this data, it can ultimately be concluded that, of the ten algorithms that were tested, the counting sort is the most efficient (taking a mere 31 milliseconds on average to complete sorting) while the bubble sort is by far the least efficient (taking over 1.5 million milliseconds, or roughly 25.7 *minutes*, on average to complete sorting). The insertion and selection sorts, while much more efficient than the bubble sort, took drastically longer to complete sorting than the other algorithms that were tested. While the quick, radix, heap, merge, and shell sorts all compared relatively similarly in terms of average sorting time, the bucket sort took nearly three times as long (at least when compared to the algorithms that timed faster than itself). The massive gap in sorting speed between the bucket and insertion sorts is so extreme that it divides the collection of algorithms remarkably clearly. With every algorithm tested and compared, the next phase of the research can begin. From this point forward, the counting and bubble sort algorithms will become the primary focus.

3. Narrowing Down the Research

After testing out ten different commonplace sorting algorithms, it has been observed at a basic level that the counting sort seems to be the most efficient in terms of demand on system time, while the bubble sort is the least efficient and by far the most time-consuming during runtime. In order to understand this drastic difference between the sorting speeds of these two algorithms, however, it is first necessary to understand the different approaches they take to sorting on an internal level.

3.0.1. The Counting Sort Algorithm

The counting sort takes a relatively unique approach to sorting, using key values associated with each element in the dataset in order to properly sort them. The algorithm begins by scanning through the dataset a single time and finding the largest value in the set. This value is then stored as the "max" value. A second "counting" array is created, its length is set to the "max" value increased by 1, and all elements in the list are set to the value 0. The algorithm then scans through the original array once more, and "counts" the total occurrences of each different value in the set by increasing the value of their corresponding indexes in the "counting" array by 1 upon locating each instance (for example, if the value "3" appeared six times in the original dataset, then index 3 of the "counting" array would be set to the value 6). A third array is then created as the "output" array, with its size set equal to the size of the original dataset. The "counting" array then fills this "output" array with the counted values from the original array in a specified order from beginning to end (ascending, for instance), adding each value as many times as the value stored by each corresponding index. The value of the original array is then set equal to that of the "output" array, effectively causing the original dataset to become definitively sorted [8]. Using the example provided previously, the value "3" would be added to the "output" array a total of six times in succession by the "counting" array (as index 3 of the "counting" array stores the value 6) before subsequent indexes are moved onto. By the end of the sorting process, the algorithm has only been required to scan through the original list a total of two times (once to determine the max value and once to count the occurrences of each unique value), which explains its remarkably fast completion time in comparison to other sorting algorithms. This low number of scans remains consistent no matter how large the dataset is, however larger sets may still result in individual scans taking longer to complete.

3.0.2. The Bubble Sort Algorithm

The bubble sort, on the other hand, takes a much more simplified approach to sorting. The algorithm begins to sort by first comparing the first element in the dataset to the second one. If the second element is lesser in value than the first, then the two elements are swapped. This process is repeated for each subsequent pair of elements until the end of the dataset is reached, thus shifting the lesser and greater elements in the dataset to the beginning and end of the list, respectively [1]. The outer loop then restarts, and the algorithm sorts out the new pairs of elements starting from the beginning of the list once again. This process repeats until every value in the dataset has been sorted in the specified order (in the case of this research, ascending). The bubble sort's name comes from the way values in the list tend to "bubble" from one end to the other during the sorting process; in ascending case, the lesser values would "float" up to the beginning of the list. Compared to the counting sort's more complicated method, it becomes clear as to why the bubble sort takes much longer to complete sorting. While the counting sort only needs to scan through a given array twice at most, the bubble sort may be required to scan through an array potentially hundreds of times (depending on the amount of data present) before the elements are swapped in an order that properly sorts the list. The larger the list, the more scans the bubble sort is required to take; hence why each increase to the set size causes the time taken for sorting to drastically increase.

3.1. Exploring the Potential

Previously, the ten algorithms that were selected for testing were all run within a Windows OS environment in the Java programming language. While the counting and bubble sort algorithms displayed considerably efficient and inefficient results respectively, it still leaves the question as to whether or not this performance was directly impacted by the choice of programming language. In order to properly observe a more accurate display of each algorithm's potential, it is necessary to analyze how the programming language chosen affects their overall performance (and consequently output speed). For the next test, the counting and bubble sort algorithms will be programmed in two additional programming languages (Python and C++) alongside the respective Java programs that were tested previously. All tests will once again be run in a Windows 10 64-bit OS environment, however the size of the dataset will be reduced to 100,000 going forward for the sake of simplicity.

3.2. Phase II Graphed Data

The results of the revised sorting test for both algorithms in each programming language are outlined in the table below. For simplicity, the values displayed in each data visualization in this section have been rounded to two decimals.

Algorithm Tested	**Test 1 Sorting Time (in milliseconds)**	**Test 2 Sorting Time (in milliseconds)**	**Test 3 Sorting Time (in milliseconds)**	**Test 4 Sorting Time (in milliseconds)**	**Test 5 Sorting Time (in milliseconds)**	**Average Sort Time (in milliseconds)**
Counting Sort (Java)	3.30	3.38	3.48	3.66	3.29	3.42
Counting Sort (Python)	57.86	59.85	55.82	55.90	56.85	57.26
Counting Sort (C++)	2.03	1.99	1.99	0.99	2.03	1.81
Bubble Sort (Java)	19794.89	20535.33	21765.97	20293.25	20329.35	20543.76
Bubble Sort (Python)	821506.07	893515.59	977784.25	906567.79	989357.89	917746.32
Bubble Sort (C++)	1132.53	609.95	882.25	563.34	809.69	799.55

Fig. 3.1: Phase II Test Results

Using the data outlined in the chart above, the differences the three chosen programming languages had on the two algorithms previously discussed can be properly graphed. The drastic difference between the output times of the two algorithms (even after normalization) forces the data to be displayed across two different graphs; one for each algorithm.

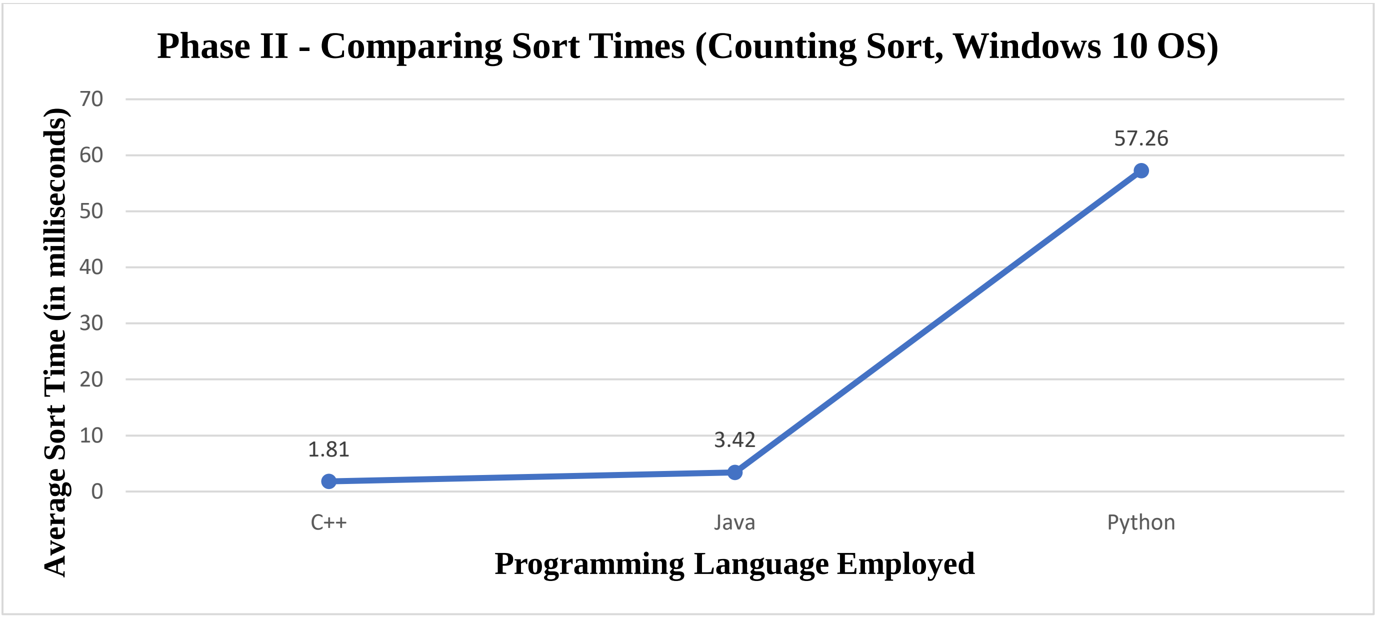

Fig. 3.2: Phase II Graphed Comparison (Counting Sort)

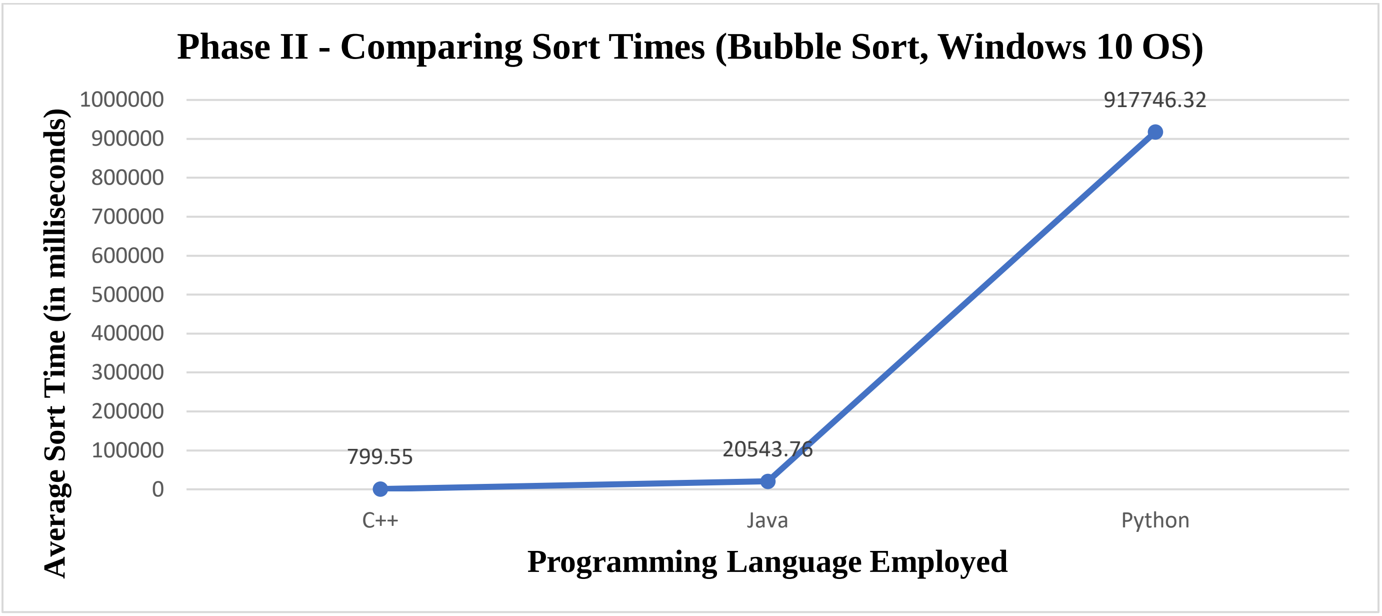

Fig. 3.3: Phase II Graphed Comparison (Bubble Sort)

The first and most obvious aspect of the data visualized above is the significant difference in output speed the Python programming language demonstrates compared to both Java and C++. This difference is most notably observed in the graph outlining the results of the counting sort algorithm. While both Java and C++ demonstrated relatively similar average sorting times, Python took an average of just over *nineteen times* as long as Java to finish the exact same process. This same result appears in the bubble sort's graphed data as well, where Python took just over fifteen full *minutes* on average to sort a dataset with the algorithm whereas Java was able to sort the same dataset in an average of just over twenty seconds. C++ ultimately demonstrated the most efficient sorting speed out of the three programming languages tested, completing a bubble sort of 100,000 values in under a second on average and sorting the same dataset with a counting sort algorithm in an average of just over a single millisecond. The results observed using the C++ programming language are therefore *monumentally* more efficient than *all* other tests that have been performed during the research thus far. To properly explain these results, however, each programming language and their respective operations must be investigated in-depth.

3.3. Seeking Out Explanations

When comparing programming languages in terms of process completion speed, it is paramount to understand the approaches they take toward interpreting and subsequently executing code. As previously observed, Python is by far the slowest programming language when regarding speed of program execution and completion. While writing programs using said language may be the most efficient from a programmer's perspective (due to its remarkably simplified syntax), Python is still an interpreted language. This means that Python executes programs one line at a time, whereas compiled languages such as Java and C++ simply compile the entire program prior to execution. This results in Python taking significantly longer to execute the code compared to the other aforementioned languages, as the running program is being interpreted and processed in real time [9].

This, however, still leaves the question as to why C++ was observed to be notably faster than Java, despite both programming languages being compiled. This similarity is reflected in the noticeably smaller gap between the two languages in the graphed data, however C++ still demonstrated a significantly higher completion speed when executing a slower algorithm (in this case, the bubble sort algorithm). While programs written in C++ are machine-independent and therefore can be run on almost any system regardless of additional installed components, Java requires a Java virtual machine (JVM) to execute Java code. Because these JVMs are expected to cause minor slowdowns in execution due to class initialization resulting in performance overheads, Java takes slightly longer to finish executing programs compared to C++ [11]. C++ code does not require any sort of virtual machine to execute, negating the possibility of performance overheads like this while using the language.

In summary, Python's execution slowdowns are largely caused by its interpreted nature and the fact that code is compiled by the language in real time rather than pre-execution. C++ typically does not require additional components to execute code on a given machine while Java does, causing a slight gap in performance speed between both languages despite their shared attribute of being compiled rather than interpreted languages. C++, requiring minimal additional preparation from the programmer and executing programs with remarkable speed, appears to be the ideal language to use when seeking out faster methods of sorting larger sums of data.

4. Alternating the Environment

With the data collected up until this point, it has become clear that the choice of programming language can heavily impact the efficiency of a sorting algorithm, and therefore by extension any program written in a specified language. However, there remains one more additional major factor that could also affect a program's performance just as heavily. All tests performed thus far have been executed on the Windows 10 64-bit operating system. Despite the fact that Windows is one of the three most commonly-used operating systems today [3], only 72% of computers worldwide run off of the system [7]. The remaining 28% of computers largely run off of the Mac OS and Linux operating systems, and thus could potentially yield different results when executing sorting algorithms in comparison to the results that have already been observed through Windows. In order to truly understand in what ways the choice of both programming language and operating system can affect the performance of an algorithm, it is necessary to expand the previously-executed tests to cover these aforementioned additional operating systems.

During the final phase of the research, the two algorithms previously observed will once again be tested in the Java, C++, and Python programming languages, sorting a dataset of 100,000 numeric values. However, each algorithm will also be tested individually on the following machines (described in more detail in the Methods and Materials section), each one running off of a different operating system:

- A Dell G7 7588 laptop running the Windows 10 64-bit operating system.
- The same Dell laptop running the Linux Ubuntu 22.04.4 operating system on Oracle's VirtualBox.
- A 2023 MacBook Pro laptop running the macOS Sonoma 14.2.1 operating system.

4.1. Phase III Graphed Data

The results of the finalized sorting test for both algorithms are outlined in the table and graphs below.

Algorithm Tested	Language Used	Operating System	Test 1 Sorting Time (ms)	Test 2 Sorting Time (ms)	Test 3 Sorting Time (ms)	Test 4 Sorting Time (ms)	Test 5 Sorting Time (ms)	Average Sort Time (ms)
Counting Sort	Java	Windows	4.82	5.48	3.84	4.46	3.35	4.39
		macOS	2.11	2.12	2.83	2.23	2.57	2.37
		Linux	4.28	3.62	3.68	3.26	3.86	3.74
	Python	Windows	64.83	61.88	67.82	53.87	54.90	60.66
		macOS	24.30	24.36	24.40	24.26	23.28	24.12
		Linux	52.97	48.89	54.36	52.91	53.56	52.54
	C++	Windows	1.99	1.99	1.99	1.03	0.99	1.60
		macOS	3.27	3.07	3.23	2.72	2.98	3.05
		Linux	4.62	2.03	2.50	2.36	4.63	3.23
Bubble Sort	Java	Windows	19593.37	19616.88	20121.45	20337.40	19885.05	19910.83
		macOS	12065.74	12431.59	12375.87	12483.75	12364.60	12344.31
		Linux	15078.17	14959.64	14831.08	14912.06	14761.59	14908.51
	Python	Windows	973706.60	926772.92	919395.76	980338.94	938489.72	947740.79
		macOS	382700.09	393093.04	401573.08	381948.74	390578.09	389978.61
		Linux	708069.51	694200.07	650416.52	695688.71	831016.39	715878.24
	C++	Windows	528.43	537.55	559.55	518.39	530.99	534.98
		macOS	20592.60	20585.50	20599.10	20637.80	20521.80	20587.36
		Linux	42775.60	42736.20	42303.20	43359.90	43969.20	43028.82

Fig. 4.1: Phase III Test Results

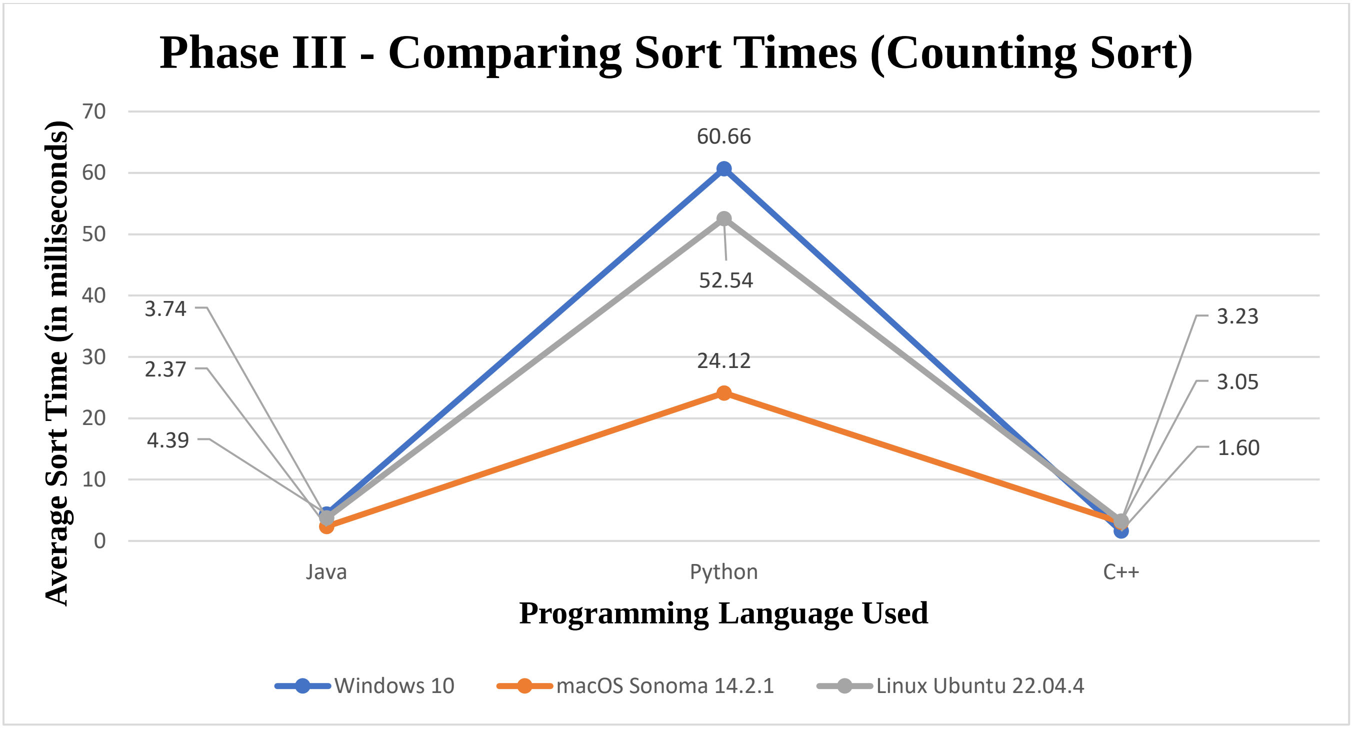

Fig. 4.2: Phase III Graphed Comparison (Counting Sort)

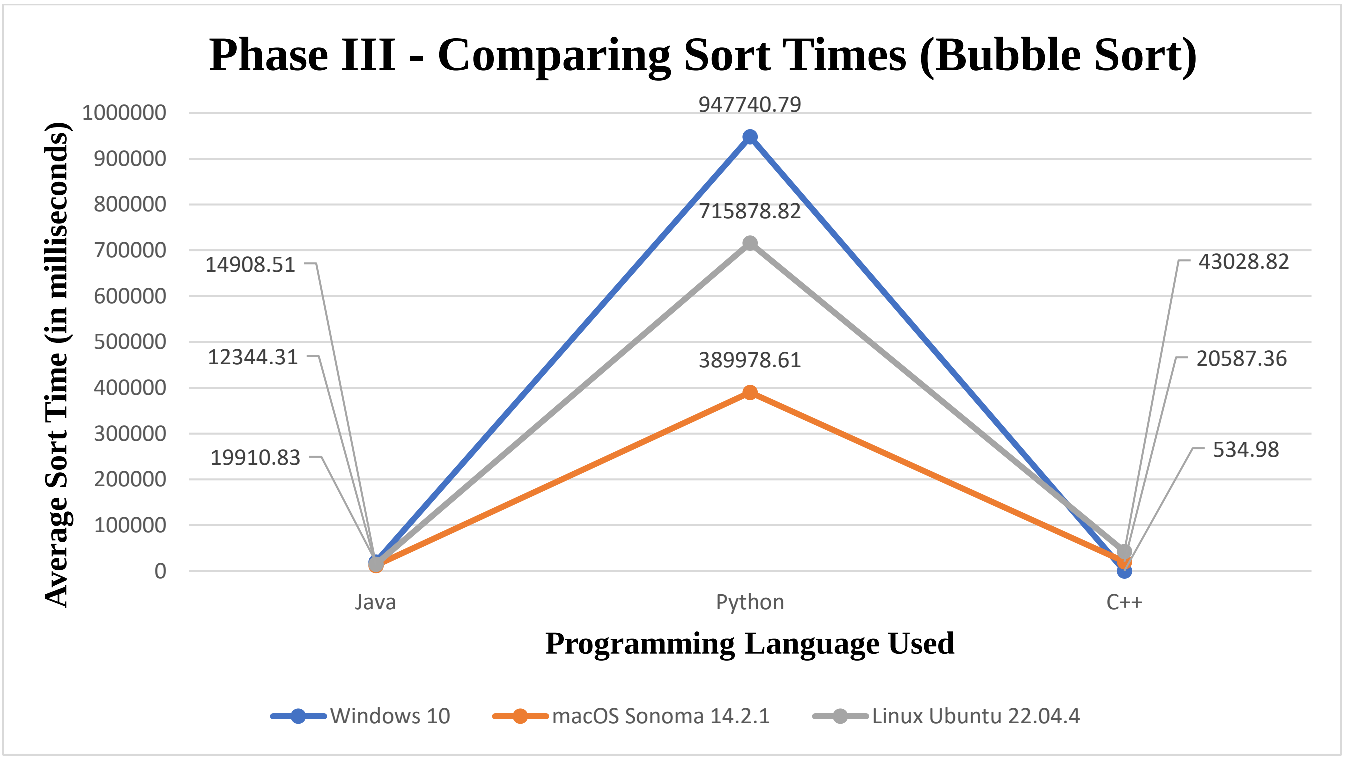

Fig. 4.3: Phase III Graphed Comparison (Bubble Sort)

5. Final Results and Discussion

The graphs above effectively visualize the efficiency of each individual operating system that was tested, as well as how performance varied between each programming language and sorting algorithm due to the difference in OS environment. The results of these tests reflect those visualized during the previous phases of the research; Python is still noticeably slower than the other two languages tested and the bubble sort algorithm seemed to have more variance overall with how different languages were able to handle running it. However, perhaps the most interesting and notable aspect of the two graphs above is how each different operating system introduced during this phase seemed to have strikingly different levels of speed variation between different programming languages.

macOS, while its efficiency results were mostly average from a basic standpoint, had remarkably tighter upper and lower bounds in terms of time taken to complete sorting. While both Windows and Linux operating systems seemed to have much more variance in speed between Python and the other two languages that were tested, macOS was able to maintain a much smaller amount of variation between the execution speeds of all three. This is fairly easy to observe by viewing the first graph outlining the results of the counting sort algorithm. Both Windows and Linux are represented by lines that take on an almost triangular shape as a result of the speed differences across the three languages, while the line representing macOS is nearly flat in comparison. As evidenced by the subsequent graph, however, these levels of variation seem to gradually increase the longer an algorithm takes to execute (although not by much). This is likely due to the way macOS compiles and processes different programming languages in general. While Python is a fairly standard programming language used on macOS, other languages like Java and C++ are typically disregarded by Apple developers in favor of languages like Objective-C [5]. For this reason, it is necessary to install very specific software to execute programs written in languages like Java and C++ on macOS. With these additional constraints, it may take a system running macOS longer to execute programs written in these languages, consequently causing their execution speeds to more evenly match those of more supported (yet slower) languages like Python.

The results gathered from Windows and Linux, on the other hand, were noticeably similar to one another. This may have been a consequence of using the same machine to run both operating systems, however the difference in execution speed variance between the two is still fairly noticeable through viewing the graphs. Both systems produced nearly identical results when running a counting sort algorithm in both the Java and C++ programming languages. However, this similarity quickly vanished once a more inefficient algorithm was tested, hence why the results produced by the bubble sort algorithm display a much more noticeable difference between the two operating systems in the second graph. As expected, the only language that seemed to display varying results throughout the testing of both algorithms was Python. This can likely once again be pinpointed to the fact that Python is an interpreted language, and its execution speed is largely more dependent on how fast the interpreter can process each line in the program than how fast the system can execute it once it is compiled. This results in much less predictability when it comes to output speed, and of course more variance. This, however, still leaves the question as to why Linux had lesser variance in sorting time compared to Windows. This can be attributed to the very simple fact that Linux is a lot more lightweight on a machine's processor than Windows. While Windows typically has several programs running in the background that cannot even be controlled by the user, Linux has very few and thus has a lot more available memory to work with [10]. Windows may have been losing resources to background processes and could not dedicate the optimal amount to the sorting algorithms during each test, while Linux was likely able to provide the necessary resources much more reliably.

6. Conclusions

From a very basic point of view, the data collected yields the following results:

- The **most** efficient sorting algorithm documented was **a counting sort algorithm, programmed in the C++ programming language and run on the Windows 10 64-bit operating system**. The average total time taken for this algorithm to completely sort a dataset of 100,000 random numeric values in ascending order was a mere 1.60 milliseconds.
- The **least** efficient sorting algorithm documented was **a bubble sort algorithm, programmed in the Python programming language and run on the Windows 10 64-bit operating system**. The average total time taken for this algorithm to completely sort a dataset of 100,000 random numeric values in ascending order was a staggering 947,740.79 milliseconds.

This means that, for systems that are required to sort through large sums of data as quickly as possible, a combination of C++ and the Windows operating system using a counting sort method would likely perform the best. Using the same operating system, it would not be effective whatsoever to instead employ the Python programming language and a bubble sort method. While Python may have a very programmer-friendly syntax, C++ requires minimal additional software to compile and run and therefore outclasses Python when system time is of the essence. This is especially true for systems like Windows that are structured using C as a base programming language [4], which the C++ syntax naturally stems from.

However, there are still many factors that could easily change this prospect. While operating systems, algorithm choices, and programming languages are definitely some of the most major factors that determine the overall efficiency of a system's sorting capabilities, there are also the impacts created by background processes, machine hardware, and many other minor attributes of the computer running the algorithm. For instance, the Linux OS has far fewer processes running underneath the surface than Windows, and with proper optimization could likely execute an even faster C++ counting sort algorithm. Ultimately, it all comes down to resource management and how the system in question chooses to handle it. There are many other operating systems and programming languages that have yet to be tested, as well as many other sorting algorithms to be analyzed in-depth. There is still a lot of room for optimization, and many combinations that could possibly yield even faster results. Through a standard analysis, it is clear that the C++ programming language, the Windows OS, and the counting sort algorithm are a quick and easy combination for guaranteed fast sorting. However, it is ultimately up to the programmer to determine whether or not this combination can be rivaled by another.

References

[1] Alake, R. (2023, November 16). *Bubble sort time complexity and algorithm explained*. Built In. https://builtin.com/data-science/bubble-sort-time-complexity

[2] del Alba, L. (2023, April 13). *10 best sorting algorithms explained, with examples*. SitePoint. https://www.sitepoint.com/best-sorting-algorithms/

[3] GCFGlobal. (2023). *Computer basics: Understanding operating systems*. GCFGlobal.org. https://edu.gcfglobal.org/en/computerbasics/understanding-operating-systems/1/

[4] Munoz, D. (2015, July 21). *After all these years, the world is still powered by C Programming: Toptal®*. Toptal Engineering Blog. https://www.toptal.com/c/after-all-these-years-the-world-is-still-powered-by-c-programming

[5] Possamai, M. (2023, March 5). *What programming languages are used at Apple?*. Medium. https://medium.com/codex/what-programming-languages-are-used-at-apple-a36261c715da

[6] Sharma, R. (2020, November 23). *Heap sort in data structures: Usability and performance*. upGrad blog. https://www.upgrad.com/blog/heap-sort-in-data-structures/

[7] Sherif, A. (2024, March 5). *Desktop Operating System Market Share 2013-2024*. Statista. https://www.statista.com/statistics/218089/global-market-share-of-windows-7/

[8] Simplilearn. (2023, February 16). *Counting sort algorithm: Overview, time complexity & more: Simplilearn*. Simplilearn.com. https://www.simplilearn.com/tutorials/data-structure-tutorial/counting-sort-algorithm

[9] SnapLogic. (2023, December 14). *Python vs. Java: Head-to-head performance comparison*. https://www.snaplogic.com/glossary/python-vs-java-performance

[10] Sruthy. (2024, March 7). *Linux vs windows difference: Which is the best operating system?*. Software Testing Help - FREE IT Courses and Business Software/Service Reviews. https://www.softwaretestinghelp.com/linux-vs-windows/

[11] Tran, T. (2023, August 21). *Java vs. C++: Which is better for your project?*. Java Vs. C++: Which Is Better for Your Project? https://www.orientsoftware.com/blog/java-vs-c-plus-plus/