e-ISSN : 2614-8404
p-ISSN: 2776-3234

Comparative Analysis of Sorting Algorithms: TimSort Python and Classical Sorting Methods

Firmansyah Rekso Wibowo[1], Muhammad Faisal[2]
[1,2] Program Studi MagisterInformatika, Fakultas Sains dan Teknologi, Universitas Negeri Islam Maulana Malik Ibrahim
Email: [1]firmansyahrwibowo@gmail.com, [2]mfaisal@ti.uin-malang.ac.id

Abstract – The sorted() function within the Python programming language has emerged as the primary choice among developers for sorting operations. Consequently, this study offers a comparative analysis of various classical sorting algorithms and Python's built-in sorting mechanisms, with the objective of identifying the most time-efficient sorting algorithm. The analysis involves assessing the time complexity of each algorithm while handling data arrays ranging from 10 to 1,000,000 elements using Python. These arrays are populated with randomly generated numeric values falling within the range of 1 to 1000. The benchmark algorithms utilized encompass Heap Sort, Shell Sort, Quick Sort, and Merge Sort. A looping mechanism is applied to each algorithm, and their execution speeds are gauged utilizing the Python 'time.perf_counter()' library. The findings of this study collectively indicate that Python's standard algorithm, surpasses classic sorting algorithms, including Heapsort, Shellsort, Quicksort, and Mergesort, in terms of execution.

Keywords – Quicksort, Mergesort, Timsort, Heapsort, Shellsort

I. INTRODUCTION

The development of science and technology allows humans to create increasingly developed and complex works, even though computers can perform calculations faster than humans in general, computers cannot simply solve problems on their own without teaching humans through sequences or steps. The steps mentioned here can be called an algorithm. There are many definitions of what an algorithm actually is. According to Sismoro (2005), an algorithm is a set of instructions or steps written systematically and used to solve logical and mathematical problems/issues with the help of a computer [1]. According to Kani (2020), an algorithm is an effort with a series of operations arranged logistically and systematically to solve a problem to produce a certain output [2]. Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output [3]. Some of the understanding obtained by researchers shows that an algorithm is a systematic process for solving a problem, and a sorting algorithm is an example that can show that a problem can be solved with a systematic process.

A sorting algorithm in general is a process for rearranging a collection of objects or data using certain rules. In programming, sorting data is important because the time required for the sorting process must be taken into account. Sequencing is also used in compiling computer programs and has an important role in increasing the efficiency of processing data that needs to be repeated. The type and amount of data that needs to be sorted varies greatly. Additionally, determining the right algorithm for a particular situation can be a difficult task because there are various factors that influence its effectiveness. There are several methods that can be used to carry out the sorting process, including Quick Sort, Merge Sort, Bubble Sort, Insertion Sort, and many more. Sorting algorithms have their respective advantages and disadvantages which depend on the amount of data. Efficiency in an algorithm is very important, according to Anggraini Kusumaningrum (2010) a good algorithm is an efficient algorithm where the algorithm is said to be good because it is assessed from the aspect of short time requirements [4].

Time complexity is a measure of the computational effort required for an algorithm to complete its task, expressed as a function of the size of its input. It quantifies how the algorithm's execution time adapts to the size of the input data and characterizes the efficiency of the algorithm by analyzing the number of basic operations it performs. Along with current technological developments, sorting algorithms have also been applied to programming languages, in this case Python. Sorting in the Python programming language uses the sort() function or what is called Timsort. Timsort is a Merge Sort (hybrid) algorithm derived from Merge Sort and insertion sort, which is designed to handle sorting on many types of data so that it can work well (Tim Peters, 2002) [5]. Timsort was created by Tim Peters in 2002 for use in the Python programming language. The algorithm finds sub-sequences of data that are already running and uses them to sort the rest more efficiently. According to Auger Nicolas et al, this is done by combining processes until certain criteria are met. Timsort has been Python's standard sorting algorithm since version 2.3.

Time complexity is a measure of the computational effort an algorithm requires to complete its task, expressed as a function of the size of its input. It quantifies how the algorithm's execution time scales with the size of the input data and characterizes the efficiency of the algorithm by analyzing the number of basic operations it performs. Along with current technological developments, sorting algorithms have also been implemented in programming languages, in this case, Python. Sorting in the Python programming language uses the sorted() function or another is called Timsort. Timsort is a combined (hybrid) sorting algorithm derived from Merge Sort and insertion sort, which is designed for handling sorting on many types of data that works well (Tim Peters, 2002) [5]. Timsort was created by Tim Peters in 2002 for use in the Python programming language. The algorithm finds sub-sequences of the data already running and uses them to sort the rest more efficiently. According to Auger Nicolas et al, this is

e-ISSN : 2614-8404
p-ISSN: 2776-3234

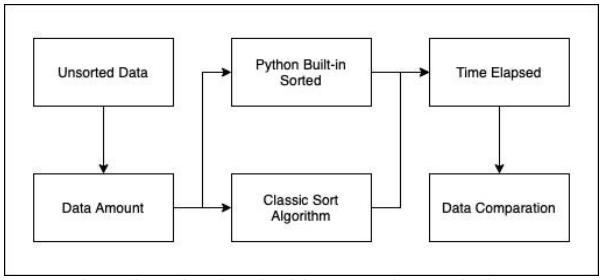

Fig 1. Flow of sorting algorithm comparison methods

done by combining runs until certain criteria are met. Timsort has been Python's standard sorting algorithm since version 2.3.

In several studies obtained from several references, like comes from Yolanda Rumapea (2017) found that the Quick Sort and Merge Sort algorithms each have advantages and disadvantages in computing time and number of steps [7]. Many factors influence this, one of which is a big factor. the size of the data input, the type of data input, and determining the pivot value (specifically in the Quick Sort algorithm).

Other studies from Oladipupo Esau Taiwo et al (2020) state that quick-sort is indeed faster, although merge-sort is stated to be better for organizing larger amounts of data/arrays [8]. In the same study, the author also stated that in terms of stability, Quick Sort is also more stable than merge-sort, also the performance of Merge Sort is indeed good, but the need to allocate memory used for sorting makes it less preferable when compared to the Quick Sort algorithm for application use where good cache locality allocation is the main thing.

In a study from S. Mansoor Sarwar et al (1993) comparing quick-sort, shell-sort and merge-sort, this study showed that shell-sort behaved better than merge-sort by $1000 < N < 150{,}000$ [9]. However, Merge Sort outperformsShell Sort for $N > 150{,}000$, then apart from these 2 algorithms, Quick Sort turns out to be better than Shell Sort and Merge Sort for all values of $N > 1000$.

Several studies tried to enhance sorting algorithms in order to efficiency. Like a study by Abu Sara et al (2020) [10], Enhanced Merge-Sort (EMS) has been carried out, experimental results show that EMS provides better sorting efficiency in terms of running speed than classic merge-sort. Another comparative study comes from Khalid Alkharabsheh discusses a comparison between the new suggested sorting algorithm (GCS) and selection sort, Insertion sort, merge sort, and quick sort. and bubble sort. It analyzes the performance of these algorithms for the same number of elements (10000, 20000, 30000) [11]. For small input, the performance for the six techniques is all nearest, but for the large input Quick sort is the fastest and the selection sort the slowest.

Opeyemi Adesina (2013) evaluated the performance of median, heap, and quick-sort techniques using CPU time and memory space as performance indexes [12]. The results obtained show that in the majority of the cases considered, the heap sort technique is faster and requires less space than median and quick sort algorithms in sorting data of any input data size.

Apart from that, there is research from Muhammad Ezar Al Rivan (2017) which tries to connect several classical sorting algorithms. The combination of the Quick-Insertion Sort algorithm has better performance compared to Quick Sort itself and Merge-Insertion Sort has better performance compared to classic merge sort and classic quick sort itself [13], Quick-Insertion Sort is 15% faster compared to Quick Sort with a limit of 16. Merge-Insertion Sort is 34.8% faster than Merge Sort with a limit of 16. After comparing several sorting algorithms from various studies above, we chose heap, shell, merge, and quick sort as a comparison method Python's built-in sorting

The aim of this research is to present a comparative study of several classical sorting algorithms and Python's built-in sorting methods with the aim of showing the time complexity of the most efficient sorting algorithms. In this case the researcher tries to prove the sorting process because each programming language creates a different sorting function. Functions such as Python build in sorted(), namely Timsort, are considered more frequently used, because classical sorting algorithms are rarely used to implement sorting.

II. RESEARCH METHODOLOGY

In this study, we aim to compare the efficiency of classical sorting algorithms with Python's built-in algorithm, known as timsort. The process flow carried out for this research is illustrated in Figure 1. The initial step involves determining the data size (denoted by 'n') to assess the real-time speed of each algorithm. To achieve this, researchers used loop functions to determine the desired data size for the execution of each algorithm examined. The data size used is randomly generated by the Python library, generating random numbers ranging from 1 to 1000. Next, these randomly generated numbers are sorted based on each selected algorithm.

Since each sorting algorithm is encapsulated in a function, the process of measuring its execution speed becomes easier. To perform these time calculations, the Python library 'time.perf_counter' will be used. Start the timing process by recording the start time before the sorting operation and conclude it by recording the end time. The difference between the final and initial values is calculated, providing the execution time for each algorithm. Next, we will describe the various algorithms used for benchmarking and comparative analysis in this research.

A. Heap Sort

Heap sort is a sorting technique that utilizes a binary tree structure to arrange elements in an array. This approach involves transforming the array into a binary tree, where the values contained in the individual array indices are then sorted. In the following section, we present a brief explanation of the heap sort method accompanied by a representative example, as illustrated in Figure 2. It is important to emphasize the heap construction of the provided array and its subsequent transformation into a maximal heap, as depicted in Figure 3. After this conversion process, the elements making up the array reach the configuration shown in Figure 4.

e-ISSN : 2614-8404
p-ISSN: 2776-3234

Fig 2. Unordered data initialization

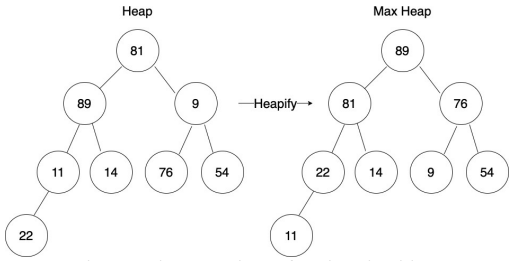

Fig 3. Study comparison of sorting algorithms

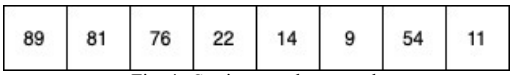

Fig. 4. Sorting max heap result

After swapping the array element 89 with 11, and converting the heap into max-heap, the elements of the array are Figure 6. The process is looping until the data is sorted properly.

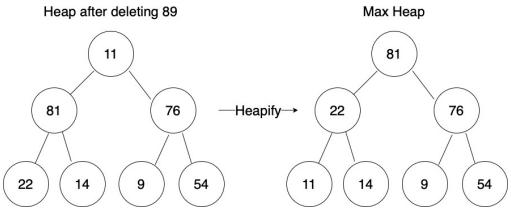

Figure. 5. Next step of erasing highest heap

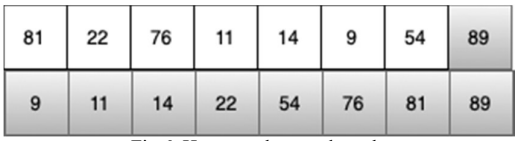

Fig 6. Heap sort loop and result

In heap sort there are 3 parts, namely Node, Edge, and Leaf where the node is each index in the array, the edge is the line that connects each node and the leaf is each node that does not have a child node (child node). Apart from that, there is also something called root, which is the initial node in a heap. Max heapify has complexity O(logN), build Maxheap has complexity O(N) and we run Max heapify **N−1** times in heap_sort function, therefore the complexity of heap_sort function is O(NlogN)

B. Shell Sort

This sorting technique, commonly referred to as the "diminishing increment method," is frequently denoted as the "Shell Sort Method." Its inception can be attributed to Donald L. Shell in 1959 [17], hence the nomenclature. This method orchestrates the sorting of data by scrutinizing each data element in relation to other elements situated at specific intervals, effecting exchanges where deemed necessary. The sorting process using the Shell method can be explained as follows:

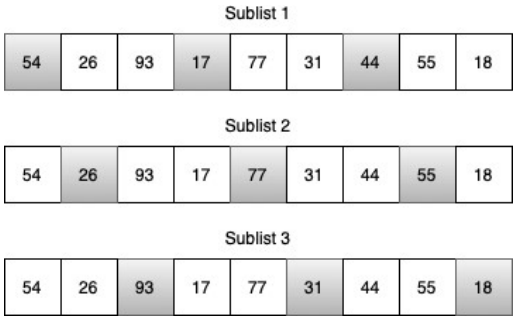

Fig 7. Initial sublist of Shell Sort

We can observe this in Figure. 7, where there are nine items in the list. By employing an increment of three, the list is divided into three sublists, each of which can be individually sorted using insertion sort.

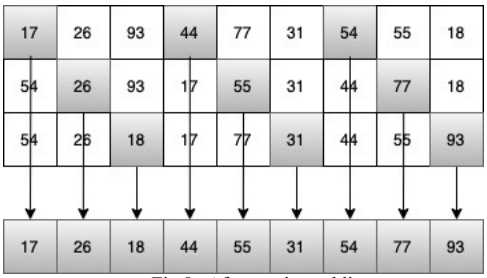

Fig 8. After sorting sublist

Once these individual sorts are finished, you'll notice the resulting list in Figure. 8. While it may not be entirely sorted, an intriguing transformation has occurred. Sorting the sublists has brought the items closer to their respective correct positions within the list.

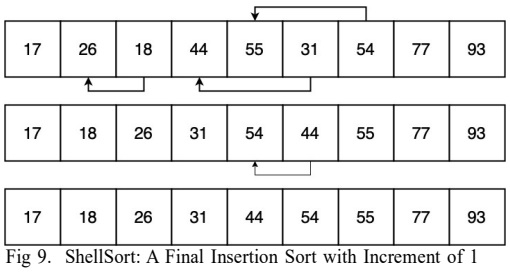

Fig 9. ShellSort: A Final Insertion Sort with Increment of 1

In Figure. 9, you can observe the last step of the insertion sort, which uses an increment of one, essentially representing a traditional insertion sort. It's worth noting that the previous sorting of sublists has effectively minimized the total number of required shifting operations to arrange the list in its correct order. In this particular instance, only four additional shifts are needed to finalize the sorting process.

C. Merge Sort

The Merge Sort algorithm uses the divide and conquer concept. The Merge Sort algorithm is an algorithm that performs sorting by dividing data into small parts. Then these small parts are divided into small sub-parts until one element is obtained. Sorting is done simultaneously with merging. One element is combined with another element by

e-ISSN : 2614-8404
p-ISSN: 2776-3234

directly sorting it. This combination of elements is then combined again with other combinations of elements. The time complexity of the Average Case and Worst Case is [5].

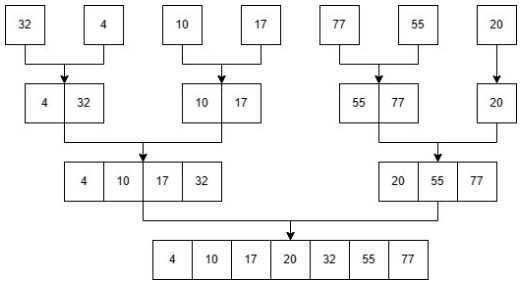

Fig 10. Merge Sort algorithm workflow

Briefly, Merge Sort can be explained as follows (Figure 10). Initially, the array will be divided into two almost equal parts. This is done by finding the midpoint of the array. This process repeats recursively until each subarray has only one element. This is the basic step (base case). Next, the two sorted subarrays are merged into one sorted subarray. When performing a merge, it compares the elements of the two subarrays and places them in thecorrect order. Next, the division and merge steps are repeated for each subarray until the entire array is sorted. The base case of recursion is when the subarray contains only one element or is empty. When that happens, the subarray is considered sorted and no longer needs to be sorted.

D. Quick Sort

Quick Sort method is also often called the Partition Exchange Sort method. This method was introduced by C.A.R. Hoare. To increase its effectiveness, in this method the distance between the two elements whose value will be exchanged is determined to be quite far. The Quick Sort sorting method can be implemented in non-recursive and recursive forms [7].

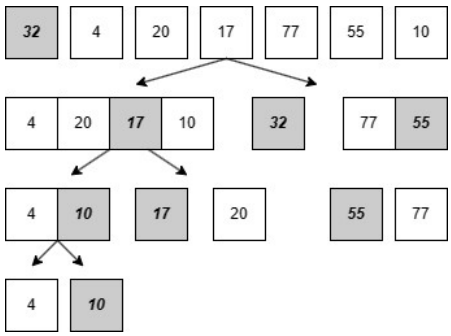

Fig 11. Quick Sort algorithm process flow

The sorting process is carried out by breaking the data set into two parts based on the selected pivot value. In principle, the selected pivot value will be placed in position at the end of each partition process. After the partition process is complete and the pivot is placed in the right position, the sorting process continues recursively to sort the data on the left pivot side and the right pivot side. In general, the Quick Sort sorting process can be explained in the following image, Figure 11.

E. Python Built-in Sort (Timsort)

Timsort is designed to take advantage of running sequential elements that already exist in most real-world data. This repeats the data collection elements into the process and simultaneously places the process in the stack. Whenever runs in the stack match the Merge Sort criteria, they will be merged. This goes on until all data has been passed, then all processes are merged two at a time and only one sorted process remains. The advantage of merging run sequences over merging fixed-sized sub-lists (as classical Merge Sort does) is that it reduces the total number of comparisons required to sort the entire list [19]. Each process has a minimum size, which is based on the input size and is determined at the beginning of the algorithm. If the process is smaller than this minimum process size, the insert type is used to add more elements to the process until the minimum process size is reached.

Timsort is a stable sorting algorithm (the order of elements with the same key is maintained) and attempts to perform a balanced merge (the merge combines its size) [5].

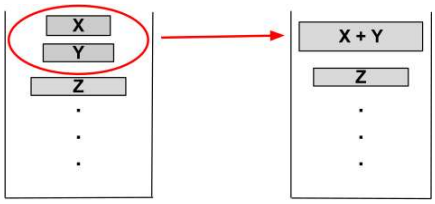

Fig 12. Magnetization as a function of applied

If | Z | | Y | + | X |, then X and Y are combined and replaced on the stack. In this way, the merge continues until all runs satisfy it. | Z | > | Y | + | X | and ii. | Y | > | X | [16].

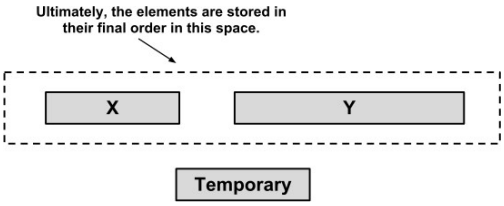

Fig 13. Python Built-in Flow

To merge, Timsort copies the elements of the smaller array (X in this illustration) to temporary memory, then sorts and fills the elements in final order into the combined space of X and Y Figure 13. Elements (indicated by blue arrows) are compared and smaller elements are moved to their final positions (indicated by red arrows) Fig. 14. All red elements are smaller than blue (here, 21). Thus, they can be moved in chunks to the final array of Figure 15.

e-ISSN : 2614-8404
p-ISSN: 2776-3234

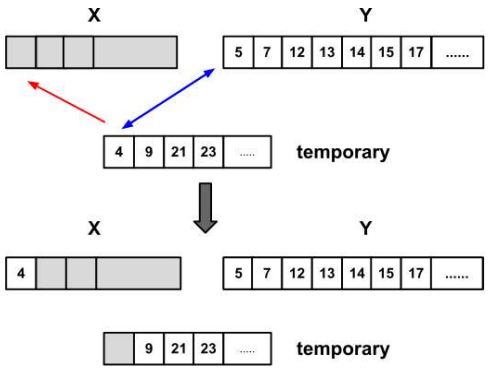

Fig 14. Element and final position

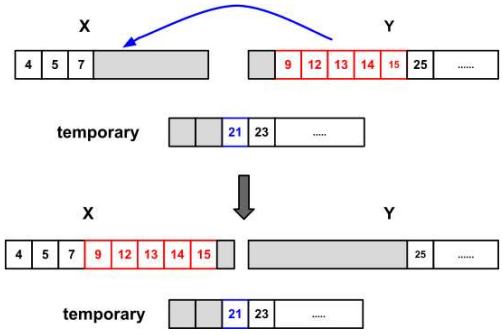

Fig 15. Element configuration

The Timsort algorithm looks for a sequence of minimum size, min runs, to perform the sorting. Because merging is most efficient when the number of runs is equal to, or slightly less than, a power of two, and notably less efficient when the number of runs is slightly more than a power of two, Timsort chooses minrun to try to ensure the former condition.

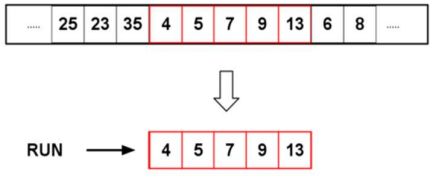

Fig 16. Timsort algorithm searches for minimum-size

III. RESULTS AND DISCUSSION

The average time complexity of the classic sorting algorithm heap, shell, merge, quick sort is O(n log(n)), which is the same as Python built-in (Timsort)t. Additionally, the best and worst case time complexity of merge sort is also O(n log(n)), which is the also same as quicksort and heap sort. As a result, the classical merge sort is generally unaffected by factors in the initial array.

However, classical merge sort uses O(n) space, since additional memory is required when merging. Quicksort also has this space complexity, while heap sort takes O(1) space since it is an in-place method with no other memory requirements.

A summary overall of the complexity time is shown in Table 1. The results of the comparison method produced first are data comparisons (n), namely 10 to 100 unsorted data. The comparison results are shown in Table 2.

Table 2. Time elapsed for each algorithm (10 - 100)

Heap (ms)	Shell (ms)	Merge (ms)	Quick (ms)	Timsort (ms)	Data Amount
0.04	0.01	0.02	0.02	0.01	10
0.03	0.02	0.04	0.03	0.01	20
0.04	0.03	0.05	0.05	0.02	30
0.06	0.04	0.06	0.06	0.02	40
0.08	0.06	0.07	0.08	0.02	50
0.16	0.05	0.08	0.09	0.03	60
0.11	0.07	0.1	0.12	0.04	70
0.13	0.08	0.12	0.13	0.04	80
0.14	0.09	0.14	0.15	0.05	90
0.18	0.11	0.17	0.2	0.05	100

.

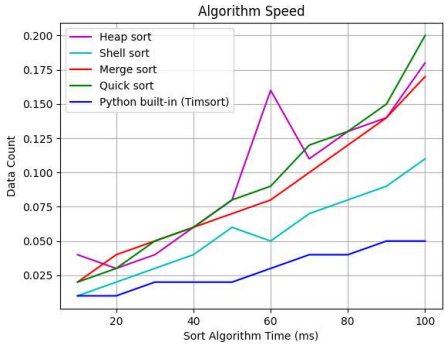

Figure 16. Time Complexity (10-100)

As the result from 10-100 sorting data Fig. 16, we found Quick sort and Python Sort appear to be the fastest sorting algorithms across all data sizes, consistently taking the least amount of time.

Table 3. Time elapsed for each algorithm (100 - 1.000).

Heap (ms)	Shell (ms)	Merge (ms)	Quick (ms)	Timsort (ms)	Data Amount
0.18	0.11	0.17	0.2	0.05	100
0.48	0.23	0.33	0.33	0.1	200
0.66	0.38	0.49	0.49	0.15	300
0.81	0.6	0.68	0.75	0.22	400
1.07	1.01	0.94	0.93	0.32	500
1.65	0.97	1.49	1.01	0.31	600
1.55	1.12	1.26	1.38	0.38	700
1.85	1.5	1.75	1.58	0.41	800
2.09	1.73	1.64	1.49	0.46	900
2.41	1.84	1.89	1.71	0.51	1000

Heap Sort, Shell Sort, and Merge Sort tend to take more time as the data size grows, and their execution times are relatively close to each other. The Python Sort (built-in) consistently outperforms the custom sorting algorithms in terms of speed (Table 2).

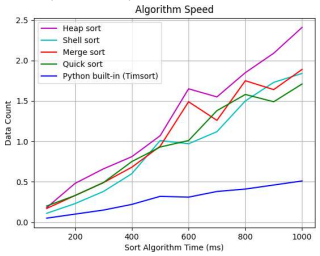

Fig 17. Time Complexity (100-1,000)

As we can see (Table 3), Quick Sort and Python Sort are consistently faster across all data sizes. Quick Sort, in particular, maintains its efficiency even as the data size grows. Heap Sort, Shell Sort, and Merge Sort exhibit longer execution times as the data size increases, with Heap Sort being the slowest among the custom sorting algorithms Fig. 17.

Table 4. Time elapsed for each algorithm (1.000 - 10.000).

Heap (ms)	Shell (ms)	Merge (ms)	Quick (ms)	Python built-in (ms)	Data Amount
2.41	1.84	1.89	1.71	0.51	1000
6.02	4.34	3.86	3.12	1.01	2000
8.51	6.53	6.13	4.3	1.54	3000
11.72	8.94	8.38	5.69	2	4000
14.8	12.79	10.91	7.09	2.55	5000
18.11	14.33	13.21	9.13	3.19	6000
21.9	16.94	15.38	9.35	3.7	7000
25.36	21.11	17.81	10.75	4.28	8000
29.41	24.11	20.24	11.58	5.15	9000
32.02	27.53	22.47	13.34	5.25	10000

Heap Sort and Shell Sort show the highest percentage increases in execution time, indicating that they become significantly slower as the data size grows (Table 4 & Fig. 18).

Quick Sort exhibits a lower percentage increase compared to the custom sorting algorithms, making it more efficient for larger data sets. Python Sort remains a robust choice, with its execution time increasing by less than 1000% over the data size range, suggesting its consistent efficiency.

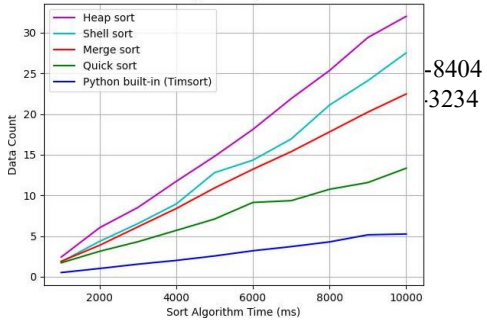

Fig 18. Time Complexity (1,000-10,000) Table 5.

Time elapsed for each algorithm (10.000 - 100.000).

Heap (ms)	Shell (ms)	Merge (ms)	Quick (ms)	Python built-in (ms)	Data Amount
32.02	27.53	22.47	13.34	5.25	10000
71.64	68.98	52.12	29.7	12.4	20000
123.65	104.99	79.05	39.74	17.68	30000
159.55	146.2	109.02	53.99	22.99	40000
203.19	190.69	145.88	67.79	28.91	50000
245.93	244.8	168.6	93.34	35.06	60000
307.37	296.3	196	90.99	40.58	70000
356.79	329.69	235.7	108.01	46.77	80000
408.63	404.4	314.56	132.6	51.35	90000
435.74	494.3	371.57	197.26	57.13	100000

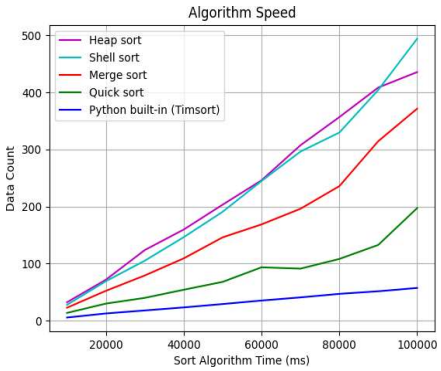

Fig 19. Time Complexity (10,000-100,000)

As data size reaches 100,000 elements, the execution times vary significantly between sorting algorithms, with Quick Sort and Python Sort maintaining their efficiency, while the other algorithms experience more substantial increases in execution time. Heap Sort, Shell Sort, and Merge Sort become increasingly slower as the data size grows. Heap Sort is notably slower for larger datasets.

Table 6. Time elapsed for each algorithm (100.000 - 1.000.000).

Heap (ms)	Shell (ms)	Merge (ms)	Quick (ms)	Timsort (ms)	Data Amount
419.13	494.3	371.57	197.26	55.16	100000
919.21	1399.76	593.99	284.13	125.2	200000
1477.77	1580.54	918.64	441.91	170.65	300000
2165.92	2286.17	1298.11	810.55	684.66	400000

e-ISSN : 2614-8404
p-ISSN: 2776-3234

2875.81	3181.02	1621.98	813.99	289.56	500000
3176.95	3522.2	1982.79	1023.11	355.8	600000
3917.22	4109.59	2332.34	1266.21	408.73	700000
4420.99	5341.38	2689.28	1477.53	469.31	800000
5263.96	5474.57	3061.3	1680.99	541.9	900000
5700.81	6999	3450.7	2011.64	596.34	1000000

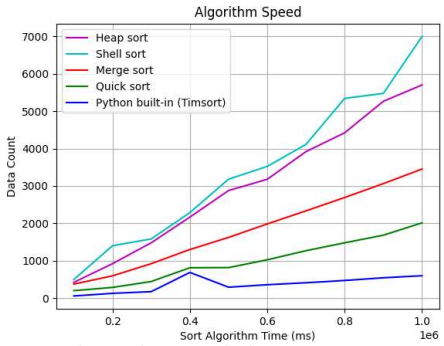

Fig 20. Time Complexity (10,000-1,000,000)

Same as described above, for the largest dataset with 1,000,000 elements (Table 6), the differences in execution times among the sorting algorithms are pronounced, with Quick Sort and Python Sort being significantly more efficient than the others (Fig. 20).

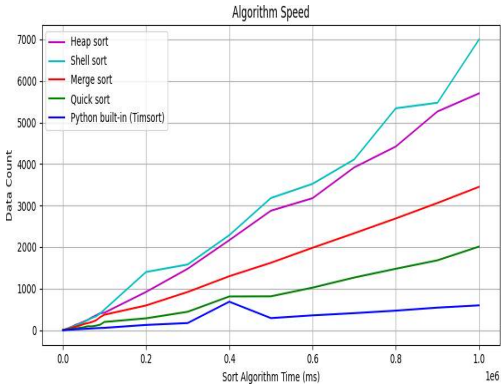

Fig 21. Overal Time Complexity (10-1,000,000

Python Sort (Python's built-in sorting function) is the second most efficient sorting algorithm (Fig. 21), closely following Quick Sort. Heap Sort, Shell Sort, and Merge Sort tend to become slower as data size increases, with Heap Sort being the slowest among these three custom sorting algorithms. For the largest dataset with 1,000,000 elements, the differences in execution times among the sorting algorithms are pronounced, with Quick Sort and Python Sort being significantly more efficient than the others. As data size reaches 1,000,000 elements, the execution times vary widely among the sorting algorithms, reflecting the importance of choosing the right sorting algorithm for specific use cases.

For data sizes of 100,000 or more, the differences in execution times among sorting algorithms become even more pronounced, with Python Sort consistently demonstrating its efficiency. Heap Sort exhibits the longest execution times for large datasets, making it less practical for very large datasets

IV. CONCLUSION

The dataset consists of execution times (in milliseconds) for various sorting algorithms on different data sizes ('data_count'). This research includes five classical sorting algorithms: Heap Sort, Shell Sort, Merge Sort, and Quick Sort compared with Python Built-in Sort (Timsort), with data sizes ranging from 10 to 1,000,000 elements. As the data size increases, the execution time for all sorting algorithms generally increases, following the expected trend.

Python Sort (Python's built-in sorting function) consistently shows the fastest execution times across all data sizes, maintaining its efficiency and scalability. Quick Sort is the second most efficient sorting algorithm. Heap Sort, Shell Sort, and Merge Sort tend to get slower as data size increases, with Heap Sort being the slowest of these three specific sorting algorithms.

For the largest data sets with 1,000,000 elements, the difference in execution time between the sorting algorithms is apparent, with Quick Sort and Python Sort being much more efficient than the others. When data sizes reach 1,000,000 elements, execution times vary greatly between sorting algorithms, highlighting the importance of choosing the right sorting algorithm for a particular use case.

Other classical algorithms are still reliable for use in small data sets (1000 elements and below) but in very large data sets (100,000 elements and above), Quick Sort and Python Sort are the most efficient sorting algorithms. When choosing a sorting algorithm, we need to consider factors such as worst-case time complexity, memory usage, and specific application requirements.

Data set analysis highlights the variation in performance of different sorting algorithms as the data size increases. Quick Sort and Python Sort consistently stand out as efficient options for sorting small and very large data sets, making them the preferred choice for most practical applications. However, the choice should be aligned with your application's specific needs, taking into account factors other than execution time, such as memory usage.

REFERENCES

[1] H. Cormen Thomas, E. Leiserson Charles, L. Rivest Ronald, Stein Clifford. Introduction to Algorithms: Third Edition. MIT Press, 2009. Massachusetts Institute-of-Technology-Cambridge. https://dahlan.unimal.ac.id/files/ebooks/2009%20Introduction%20to%20Algorithms%20Third%20Ed.pdf.

[2] Kurnia, Firdilla (2022, December 28). Algoritma: Pengertian, Ciri-ciri, Jenis, Serta Fungsi dan Manfaatnya.Dailysocial. https://dailysocial.id/post/algoritma-adalah.

[3] Laitinen, S. (2010). *Better Games Through Usability Evaluation and Testing*. http://www.gamasutra.com/view/feature/2333/better_games_through_u...

[4] Anggraini Kusumaningrum, 2020. Perbandingan Kecepatan Antara Selection Sort, Insertion Sort,

e-ISSN : 2614-8404
p-ISSN: 2776-3234

Dan Bubble Sort. Teknomatika: Jurnal Informatika Dan Komputer, 3(1), 63-70. Retrieved from https://ejournal.unjaya.ac.id/index.php/teknomatika/article/view/363.

[5] Nicolas Auger, Vincent Jugé, Cyril Nicaud, Carine Pivoteau, 2018. On the Worst-Case Complexity of TimSort. France: 26th Annual European Symposium on-Algorithms-(ESA 2018).-

DOI: https://doi.org/10.48550/arXiv.1805.08612.

[6] Canaan, C. et al. "Popular sorting algorithms - TI Journals." World Applied Programming (2012): n. Pag. https://www.semanticscholar.org/paper/Popular-sorting-algorithms-TI-Journals-Canaan-Garai/432ba0382141cacef751199288147d4daaeb3cd7

[7] Rumapea, Yolanda Y. P. *Analisis Perbandingan Metode Algoritma Quick Sort dan Merge Sort dalam Pengurutan Data terhadap Jumlah Langkah dan Waktu.* Methodika, vol. 3, no. 2, 2017, pp. 5-9, https://media.neliti.com/media/publications/345425-analisis-perbandingan-metode-algoritma-q-e2e2d79a.pdf.

[8] Oladipupo Esau Taiwo, Abikoye Oluwakemi Christianah, Akande Noah Oluwatobi, Kayode Anthonia Aderonke, Adeniyi Jide kehinde, *Comparative Study Of Two Divide And Conquer Sorting Algorithms: Quicksort And Mergesort*, Procedia Computer Science, Volume 171, 2020, Pages 2532-2540,-ISSN-1877-0509, https://doi.org/10.1016/j.procs.2020.04.274.

[9] S. Mansoor Sarwar, Mansour H.A. Jaragh, Mike Wind, An empirical study of the run-time behavior of quicksort, Shellsort and mergesort for medium to large size data, Computer Languages, Volume 20, Issue 2, 1994,-Pages-127-134,-ISSN-0096-0551, https://doi.org/10.1016/0096-0551(94)90019-1.

[10] Abu Sara, M. R., Klaib, M. F. J., & Hasan, M. (2020). *Ems: An Enhanced Merge Sort Algorithm By Early Checking Of Already Sorted Parts.* International Journal of Software Engineering and Computer Systems,-5(2),-15–25.-Retrieved-from https://journal.ump.edu.my/ijsecs/article/view/3525.

[11] Alkharabsheh, Khalid & Alturani, Ibrahim & Alturani, Abdallah & Zanoon, Dr.Nabeel. (2013). *Review on Sorting Algorithms A Comparative Study*. International Journal of Computer Science and Security (IJCSS). 7. https://www.researchgate.net/publication/259911982_Review_on_Sorting_Algorithms_A_Comparative_Study.

[12] Adesina, Opeyemi. (2013). A Comparative Study of Sorting Algorithms. African Journal of Computing & ICT.-6.-199-206. https://www.researchgate.net/publication/288825600_A_Comparative_Study_of_Sorting_Algorithms.

[13] Al Rivan, Muhammad Ezar. "*Perbandingan Kecepatan Gabungan Algoritma Quick Sort Dan Merge Sort Dengan Insertion Sort, Bubble Sort Dan Selection Sort.*" Jurnal Teknik Informatika dan Sistem Informasi, vol. 3, no. 2, 2017, doi: 10.28932/jutisi.v3i2.629.

[14] You Yang, Ping Yu and Yan Gan, "*Experimental study on the five sort algorithms*" 2011 Second International Conference on Mechanic Automation and Control Engineering, Inner Mongolia, China, 2011, pp. 1314-1317, doi: 10.1109/MACE.2011.5987184.

[15] Krishna, Sai. (2017). *Comparative Analysis of Bucket and-Radix-Sorting*.
DOI: 10.13140/RG.2.2.13755.00807

[16] Kazim, A. *A Comparative Study of Well Known Sorting Algorithms.* International Journal of Advanced Research in Computers. Science. 2016; 8(1):277-280. doi : https://doi.org/10.26483/ijarcs.v8i1.2903.

[17] Mitra, Avik & Kothari, Jash & Ganguly, Annesa. (2019). *Analysis Of Shellsort Algorithms.* 10. 48-51. 10.26483/ijarcs.v10i3.643.

[18] MacIver, David R. (11 January 2010). "*Understanding Timsort, Part 1: Adaptive Merge Sort*". Retrieved 28 October-2023. https://github.com/python/cpython/blob/main/Objects/listsort.txt.

[19] Peters, Tim. "listsort.txt". CPython git repository. Retrieved-28-October-2023. https://svn.python.org/projects/python/trunk/Objects/listsort.txt.

[20] "listsort.txt". Python source code. 18 May 2022. Archived from the original on 28 January 2016. https://hmn.wiki/id/Timsort.

[21] Tim Peters. Timsort description, accessed 28 October 2023.-URL: http://svn.Python.org/projects/Python/trunk/Objects/listsort.txt.