Synergy: Cross-Disciplinary Journal of Digital Investigation
(ISSN 2995-4827) VOLUME 02 ISSUE 12, 2024

Towards a Realistic Analysis of Sorting and Searching Algorithms

Ibtihal Mohammed Khaleel
Al Farahidi University, Iraq

Abstract:

On the concepts of permutation and replication. We begin by explaining how these two concepts are embodied and how they can affect the performance of algorithms in terms of data distribution. Next, we draw on the current literature to study the complexity of different sorting algorithms, adjusting the element sizes used, ranging from 8 to 4096. We analyse how the performance of each algorithm is affected based on the size of the data and the organization methods used.

We also seek to provide data-driven insights into the efficiency of these algorithms in different operating conditions, enabling us to draw a realistic picture of their performance compared to the ideal models adopted in previous studies. The paper aims to contribute to a deeper understanding of choosing appropriate algorithms according to the size of the data and complex structures, which allows for improved data processing operations in practical applications. In conclusion, we highlight the importance of realistic analysis in developing future research in the field of sorting and searching algorithms and encourage further exploration of the factors affecting performance in different environments.

Keywords: searching algorithms, toward, sorting.

Introduction

Sorting and searching algorithms are fundamental to computer science, playing a vital role in improving data processing efficiency. These algorithms provide the necessary methods to organize data in a way that makes it easier to access and analyse. In this context, analysing sorting and searching algorithms is not just a computational process but requires a deep understanding of the fundamental properties of these algorithms, as well as the real-world conditions that can affect their performance.

The quest for a realistic analysis of these algorithms aims to go beyond the ideal models often adopted in the academic literature, where one must investigate how factors such as data size, structure, and value distribution, as well as memory and performance constraints, affect the effectiveness of algorithms. Through this realistic analysis, researchers and developers can identify the most suitable options for practical applications and achieve sustainable improvements in how data is processed.

Thus, we call for a broad discussion of current approaches to analysing sorting and searching algorithms and the search for new strategies that take into account the challenges they face in real-world contexts.

Literature Review

Sorting items in ascending (or descending) order is required when dealing with linear structures (arrays, lists, etc.) that include values of an ordered kind. This topic is addressed by dozens of algorithms that use a few fundamental ideas, to which we add variations.

Bubble-sort

The bubble sort algorithm is a simple and well-known algorithm in computer science. It is also considered to be one of the earliest sorting algorithms ever created. As a stable algorithm, it maintains the original order of the same key and is adaptable, i.e., it can use any existing order in the sequence. However, due to its high complexity, $O\{n^2\}$ in the average worst case is inefficient when sorting a large number of elements. The bubble sort consists of four main steps. First, all adjacent elements in the array are compared one by one, and if the first element is greater than the second, they are swapped. This process is repeated until all items are sorted Each iteration of the algorithm moves the largest element to the end of the array These steps are continued until a sorted array is obtained (see “Figure 2”). An improvement that can be made to Bubble Sort is the use of a boolean "flag" variable, which allows the algorithm to terminate early if no more swaps are needed.[1]

Insert-sort_

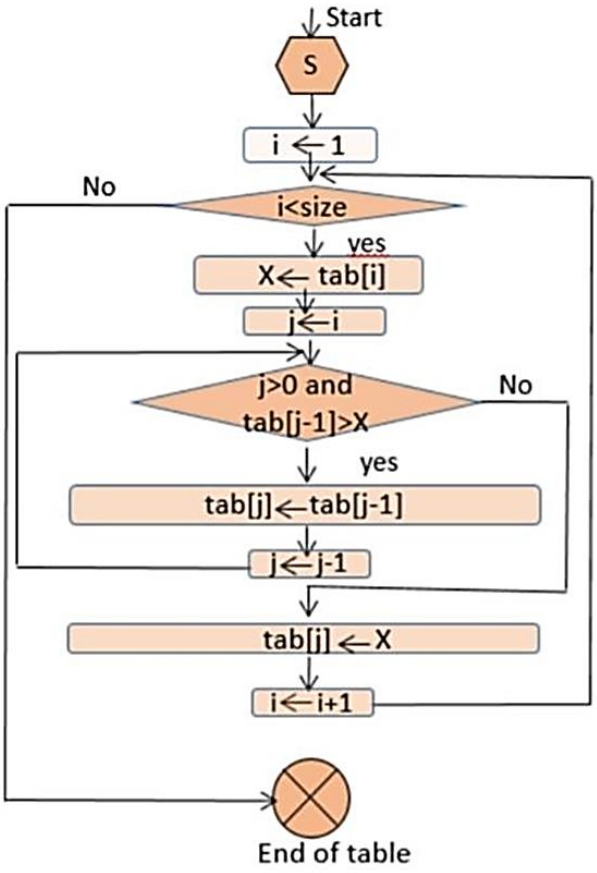
I Insertion sort is another significant algorithm, particularly useful for sorting a small number of elements, as demonstrated in Fig 3. It performs better than both bubble sort and selection sort. The implementation of Insertion sort is straightforward for arranging elements. However, it becomes less efficient when dealing with a large number of elements and is significantly slower than other algorithms such as Quick sort, Heap sort, and Merge sort, due to its high complexity of $O\ \{\ n^2\}$ on average and in the worst case.

Furthermore, this algorithm is stable, meaning it preserves the order of appearance of equal elements, and it is adaptive as well. In each iteration, the Insertion sort algorithm integrates a new element and compares it to the values of the elements in the list. If the value of this new element is less than that of the current element, the swap step is necessary. This process continues until the algorithm has processed up to n-1 elements.[2]

Selection-sort_

The selection sort algorithm is a straightforward algorithm that is easy to analyze. It is a comparison-based sorting method, making it simple to understand and particularly useful when handling a small number of elements. However, it is inefficient for sorting larger sets of data, as its complexity is $O\ \{\ n^2\}$ in all cases, where n represents the number of elements in the array that need to be sorted.

This algorithm is referred to ad



' Fig 2 (Bubble-sort) Algorithm

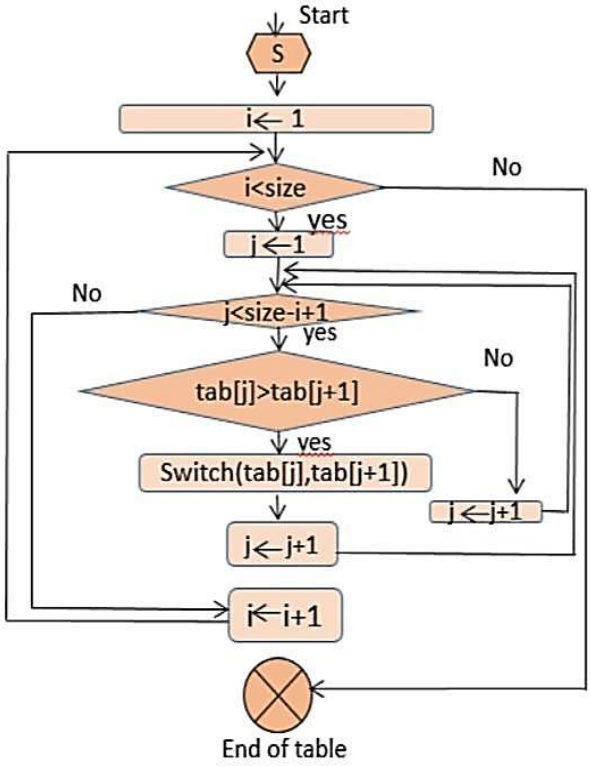


Fig 3

Insertion sort Algorithm_ Selection sort because it works by selecting a minimum number of elements in each sorting step, The principle is that to sort n values, you need to find the smallest element in the array and swap it with the element with index 0, then the smallest value in the remaining values and swap it with the element with index 1, and so on. Repeat these steps until you find the sorted array as shown in Fig 4. Selection sort is an in-place sort (the elements are sorted directly in the structure), but it is an unstable algorithm (the order of appearance of equal elements is not preserved). (3)

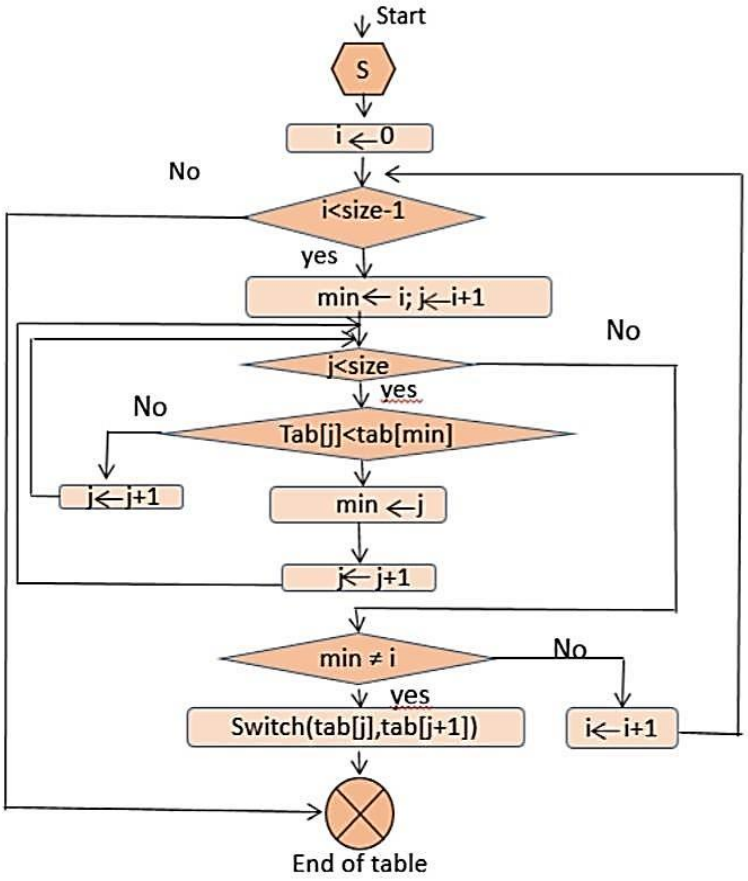
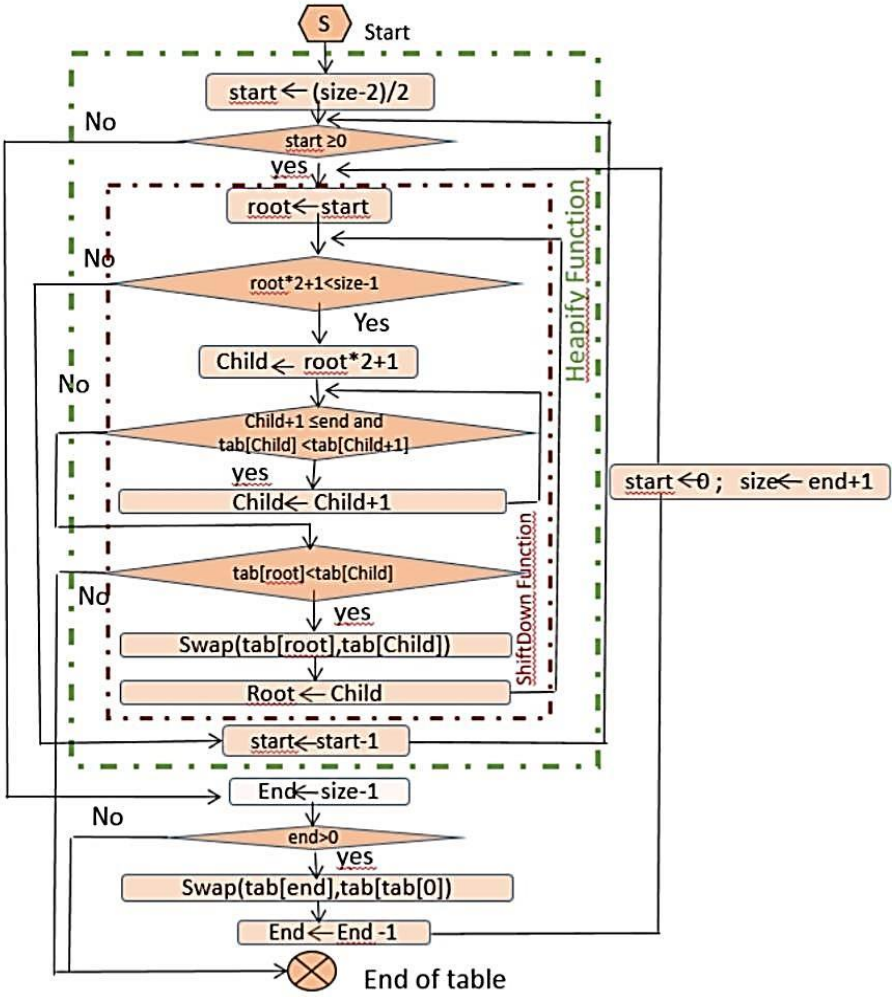


Fig 4: Selection-sort Algorithm

"Heapsort"_

"Heapsort" is a sorting technique that utilizes the efficient binary heap data structure. It resembles selection in sort in that it identifies the maximum element in the list and moves that element to the end, repeating the process for the remaining elements. "Heapsort" is an optimal comparison sort, achieving a time complexity of O"n log(n)" for any input order, where n is the length of the array. The algorithm is asymptotically optimal, meaning it has been demonstrated that no comparison sort algorithm can achieve a better complexity in the long run. Its performance is consistently O"n log(n)," making it highly efficient. [4]

"Heapsort" is relatively easy to implement as it can be done in-place and does not require recursion. From a complexity standpoint, it is one of the best sorting algorithms available, as it maintains a time complexity of "n log(n)". The algorithm consists of two primary steps, as illustrated in'Fig 5. The first step is to create a heap data structure (either Max-Heap or Min-Heap), with the root element being the largest or smallest, depending on whether a Max-Heap or Min-Heap is constructed. (5)



Heapsort" Algorithm Fig. 5"

Please repeat this step using the remaining elements to reselect the first heap element and place it at the end of the array until you obtain a sorted array. "Heapsort" is a high-speed sorting algorithm, but it is unstable since heap operations can alter the relative order of equal elements. As shown in Table 1, it has a time complexity of O"n log(n)" in both the best and worst cases. It is widely used for sorting a variety of data.

Shellsort is a general-purpose sorting algorithm that was invented by Donald Shell in 1959. Empirical results demonstrate that it competes well with the fastest sorting algorithms, particularly when the number of elements to be sorted (N) is not too large. Shellsort enhances the efficiency of insertion sort by quickly moving values to their correct positions. It is a non-stable, in-place sort. The algorithm makes several passes through the list, sorting groups of equal size using insertion sort during each pass. The main purpose of this algorithm is to calculate the value of 'h' that divides the list into smaller sublists of intervals equal to 'h'.[6]

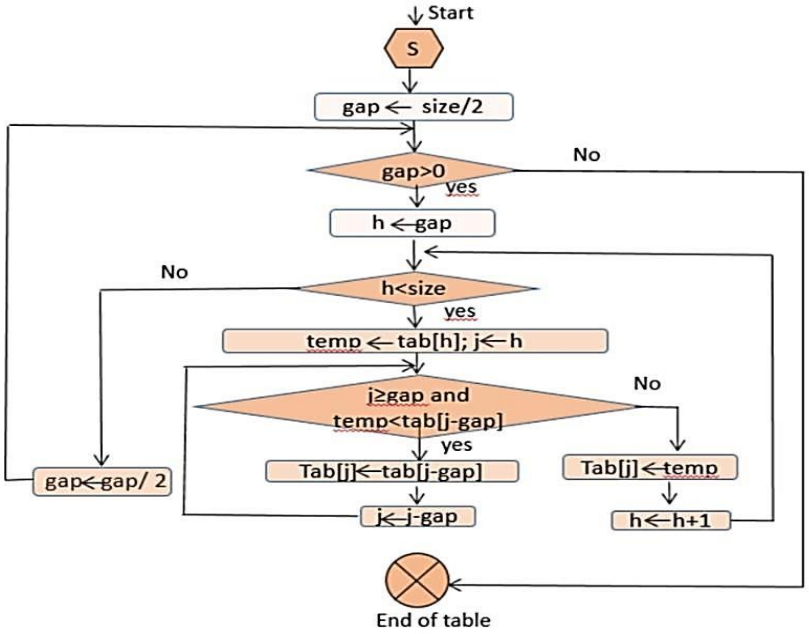


Fig. 6: Shellsort Algorithm

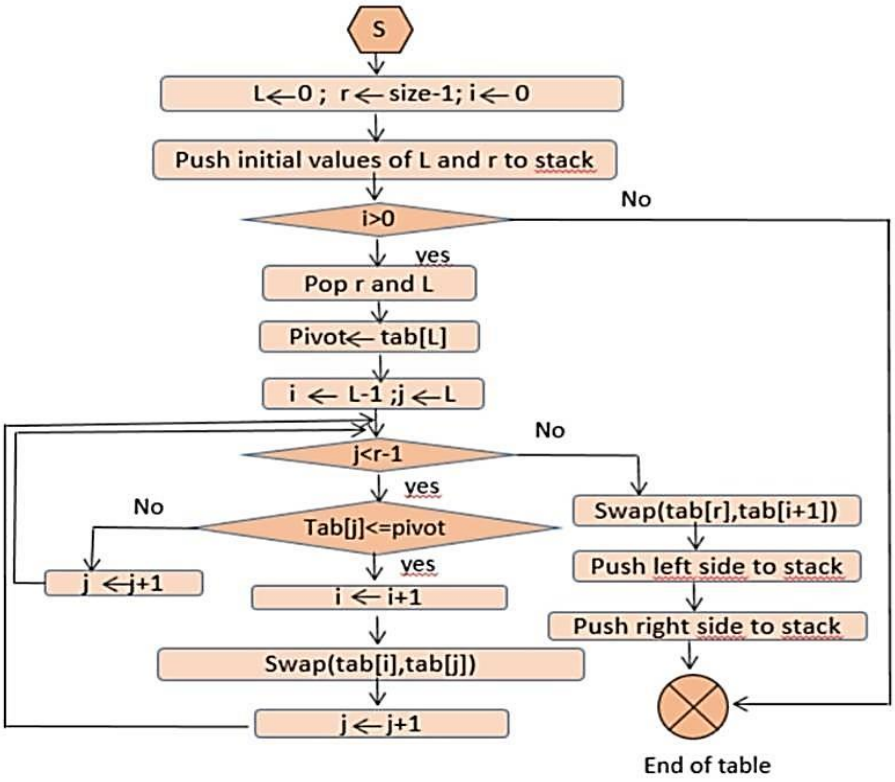
After that, it sorts each sublist that contains a large number of data using insertion sort. Finally, this step is repeated until a sorted list is obtained. The time complexity of the above implementation of shell sort is On log(n)2. It is not widely used in the literature. The steps required for the shell sort algorithm are shown in Fig A.6.

_Quick Sort

Quicksort is an algorithm based on the divide and conquer principle that employs a partitioning method. The process begins by splitting the array into two smaller subarrays: one with elements less than a chosen pivot and another with elements greater than the pivot. The steps of the algorithm are illustrated in Figure 7.[7]

Initially, a pivot element is selected from the array. During the partitioning phase, all elements smaller than the pivot are moved to its left, while those larger are moved to its right. This partitioning can be completed efficiently in linear time. Once the partitioning is done, the pivot is placed in its final position. The algorithm then recursively repeats these steps on the two resulting subarrays containing smaller and larger elements.

"Quicksort is recognised as one of the fastest sorting algorithms in practical use. Its average time complexity for sorting an array with n elements is O(n log(n)), which is considered optimal for comparison-based sorting methods. However, the algorithm's worst-case time complexity can reach $O(n^2)$. Despite this potential drawback, Quicksort is widely utilised due to its effectiveness in real-world applications. It is important to note that Quicksort is not a stable sorting algorithm, and its implementation can be complex. A major challenge within the Quicksort algorithm lies in the selection of an appropriate pivot element. To improve performance and mitigate issues arising from suboptimal pivot selections, Quicksort can opt to use the median as the pivot, helping to maintain the desirable time complexity of $O(n \log(n^2))$.[8]



sort" algorithm quick' Fig-

_"Mergesort"

Merge sort was first proposed by John von Neumann in 1945. It is an efficient and stable sorting algorithm that maintains the order of input in the output. Merge sort follows the well-known divide and conquer paradigm and is considered a stable, comparison-based sorting algorithm. Its time complexity for sorting n values is O"n log(n)," which is asymptotically optimal. However, merge sort is not an in-place algorithm because it requires additional space to store the additional array.

For a large number of data items (n), merge sort tends to be more efficient and faster than quick sort, but may be less efficient and slower for smaller data sets.

The main steps of the sorting algorithm are shown in Figure 8: [9]

1. The array is divided into two subarrays.

2. The two sub-arrays are recursively sorted independently.

3. The two sorted arrays are merged to produce the final result.

Merge sort operates with a worst-case time complexity of "n log(n)", requiring at most "n log(n)" operations.

_Tim-sort

"Timsort" is a sorting algorithm developed by Tim Peters specifically for use in the Python programming language, and it has since been implemented in other programming languages, including Java and C/C++. While it is a well-designed algorithm, its high-level principle is relatively straightforward: the sequence to be sorted is decomposed into monotonic sequences (either non-increasing or non-decreasing subsequences), which are then merged in pairs according to specific rules.

To understand and analyse "Timsort"'s merging strategy (i.e., the order in which merges are performed), we consider a class of sorting algorithms that utilise a stack to determine the merge

order. Other methods for ordering merges have been explored in the literature, such as the classical merge sort algorithm, Knuth's natural merge sort, and the optimal merge strategy proposed by Barbay and Navarro that incorporates ideas from Huffman coding. [10]

"Timsort" is a hybrid sorting algorithm that combines elements of merge sort and insertion sort. It is very stable and designed to work efficiently with real data. Its time complexity is O"n log(n)" in both the average and worst case. The algorithm's functionality is based on an optimality parameter (OP) that determines when to switch between the two sorting algorithms.

For sequential architectures (such as Intel i7 processors), this parameter is set to 64.

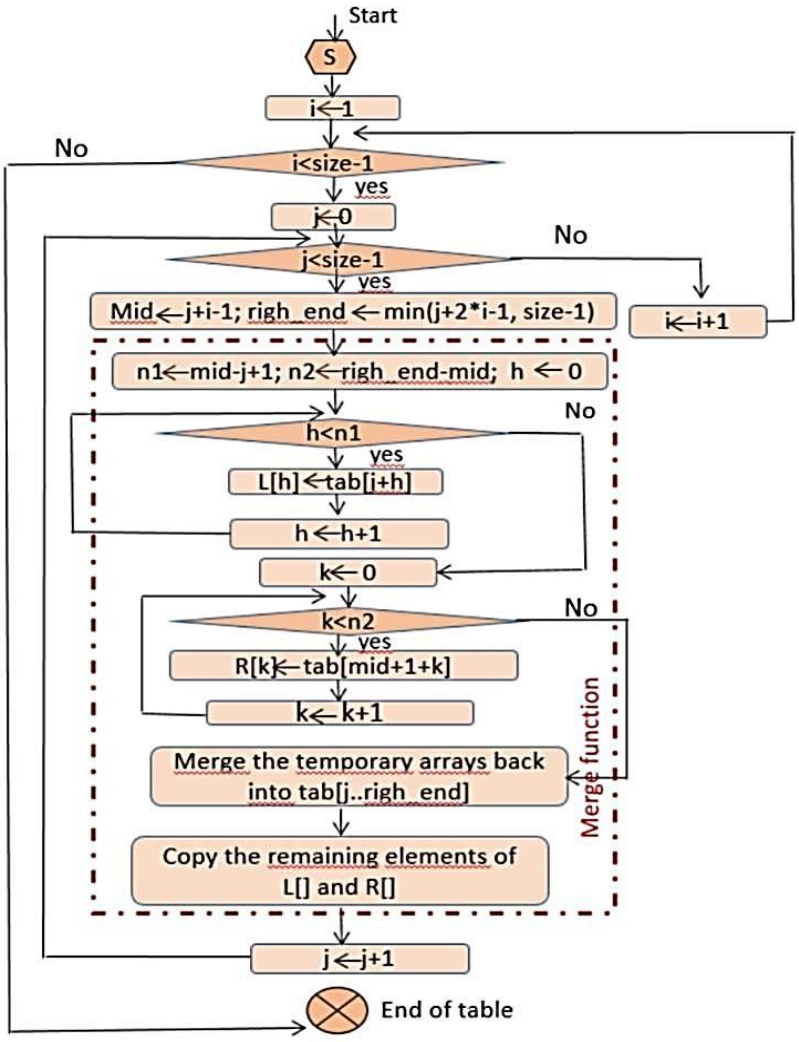


Fig. 8: "Mergesort" Algorithm

In parallel architectures, different values for this parameter can be used, resulting in nearly similar execution times. Thus, this thesis adopts a value of 64 for consistency with other authors and to facilitate cross-comparisons. Depending on the size of the data to be sorted, two different approaches are taken: if the array has fewer than 64 elements, Insertion sort is used; otherwise, Merge sort is employed in the sorting process, as illustrated in ' Fig 9. [11]

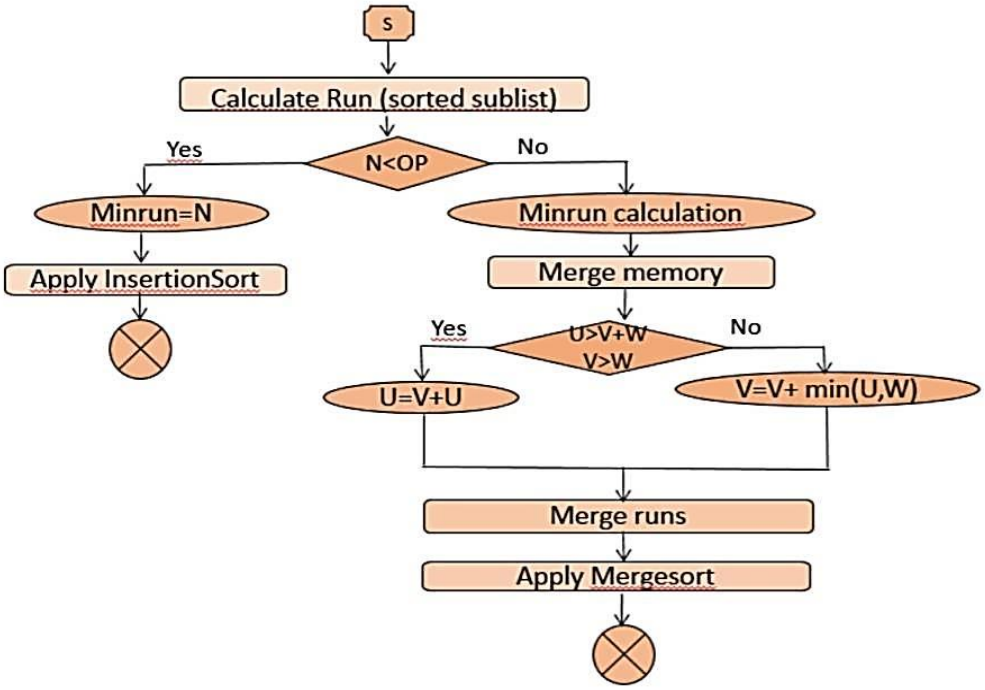


Fig 9

Methodology

we first explain the notion of permutation and replication. Subsequently, referring to the literature, we seek to study the complexity of sorting algorithms using different element sizes ranging from 8 to 4096, Among the methods used to address the problem of permutation generation, we use in this work the Lehmer method.

Lehmer's method is a specific method to encode every possible permutation of a set of n objects. It is an example of a numeric rotation permutation scheme and is the same method as the inversion table T(π) which is defined by a vector b1 b..bn, with bi being the number of elements greater than i and appearing to the left of i and 0<bi <n-i. [12]

For example, if the permutation is 52143, then the Lehmer code is 21210, which is the inversion table of this permutation since 5 is greater than 1, 2, 3, and 4, also 2 is greater than 1, and 4 is greater than 3. An inversion in a permutation is π pair of indices (i, j) such that i<j and πi< πj For example, if π = 52143, then (5,2), (5,1), (5,4), (5,3), (2,1) and (4,3) are their inversions. (L(π)) is the number of inversions (i, j) with i fixed and j variable. The identity permutation is the only permutation that does not include inversion. The same code for an inversion table is a Lehmer code, which is related to a unique sequence l1 l..ln, with l being the number of elements less than i and appearing to the right of i and 0<l1 <n-i For example, if π = 52143, then Lehmer code d = 41010.

To generate a permutation from the Lehmer code, we apply the following equation:

$\pi(1) = N1[l1+ 1]$with $N_1 = N - \{\pi(1), \pi(2), ...\pi(i - 1)\}$ (4)

With (i) is the permutation element, and i is the position of that element. With the same example:

$\pi(1) = N_1 [4+1]=5$

$\pi(2) = N2 [1+1] = 2$

$\pi(3) = N3 [0+1]=1$ =>Permutation = 1=>Permutation = 52143(5)

$\pi(4) = N4 [1+1] = 4$

$\pi(5) = N5 [0+1] = 3$

A Lehmer code is considered a factorial number, which allows the decimal number to be calculated using the following equation:

Deimal _number = Σni=0 di (n-i)! (6)

In order to find all permutations, we generate the factorial number from the decimal number in the first step and the permutations in the second step.[13]

Table 2: Execution time and standard deviation of sorting algorithms on the CPU

	BubbleSort (us)	InsertionSort (us)	SelectionSort (us)	HeapSort (us)	QuickSort (us)	ShellSort (us)	MergeSort (us)	TimSort (us)
8	1.57 (2.753)	1.074 (2.173)	1.243 (2.186)	2.296 (3.631)	2.316 (2.989)	1.514 (2.486)	2.639 (3.198)	1.818 (2.911)
16	5.725 (4.786)	2.781 (3.654)	3.730 (3.941)	5.040 (4.524)	4.725 (4.413)	4.471 (4.41)	4.253 (4.102)	2.870 (3.66)
32	24.375 (7.31)	9.011 (5.671)	15.293 (9.772)	12.371 (6.075)	13.197 (6.301)	12.184 (6.123)	14.641 (6.713)	13.084 (6.4)
64	96.871 (112.167)	28.516 (9.209)	41.079 (9.895)	48.304 (23.719)	41.555 (58.629)	42.830 (69.133)	32.052 (9.919)	34.702 (10.317)
128	432.990 (36.686)	98.884 (10.277)	164.989 (42.599)	83.592 (13.835)	93.241 (13.875)	122.11 (104.016)	52.753 (7.573)	86.434 (10.721)
256	1037.802 (182.061)	403.147 (302.001)	580.504 (83.059)	296.508 (557.587)	299.465 (431.402)	329.927 (74.621)	159.046 (19.687)	192.016 (75.775)
512	3779.371 (844.048)	1394.601 (757.201)	2127.602 (284.322)	468.886 (168.523)	1136.260 (757.716)	1145.126 (389.055)	275.380 (116.047)	507.135 (782.516)
1024	12290.912 (2519.38)	4548.036 (676.104)	7249.245 (1854.144)	1504.325 (1239.939)	3291.473 (437.65)	3528.034 (1362.46)	774.106 (409.297)	1140.902 (1164.588)
2048	47304.790 (4825.091)	24829.288 (1993.282)	27876.093 (2400.166)	2093.825 (210.328)	10978.576 (2207.375)	12180.592 (585.655)	1554.525 (85.225)	1911.241 (243.134)
4096	181455.365 (9895.405)	66851.367 (3179.333)	107568.834 (6339.409)	4281.869 (756.228)	41848.195 (3462.193)	46796.158 (1431.203)	2985.149 (159.427)	4098.597 (539.352)

Result & Discussion

For the execution time measurement, we set the number of replications (iterations) R to 1000 for each permutation. Subsequently, we calculated the average of the execution times obtained following each replication to obtain results similar to the real implementation. However, we calculate the execution time and the standard deviation of the sorting algorithms on the processor for each replication and each permutation/vector to verify the satisfaction of the real-time constraint.

The results of this study are detailed in Table. From the results provided in Table 2 and ' Fig 10, we find that the execution time of "Bubble-sort" and Selection-sort sorting algorithms is very high if the number of elements is greater than 64. .Otherwise, the insertion-sort algorithm is the best

Moreover,' Fig 11 presents a zoom on the five algorithms in terms of average execution time. We conclude that "Mergesort" is 1.9x, 1.37x, 1.38x, and 1.9 faster than "quick-sort," "headsort," "timsort," and Shellsort, respectively, when running it on the machine's CPU. Subsequently, we also calculate the performance of the algorithms with another performance criterion, which is the standard deviation on the execution time, to check if the deviations presented in Table 2 are statistically significant.

'Fig 12 gives the same conclusion for the standard deviation as for the running time. Next, we calculate the coefficient of variation, which is a relative measure of the dispersion of the data

around the mean. The coefficient of variation is calculated as the ratio of the standard deviation to the mean and is expressed as a percentage as shown in Equation 7.

$$CV = 100 * \sigma / \mu \quad (7$$

where σ represents the standard deviation of a given permutation over R replications and μ is the average of the times for a permutation over R replications This coefficient enables us to compare the degree of variation between different samples, even when their means differ On the one hand, when the standard deviation and the mean come from measurements repeated 1000 times

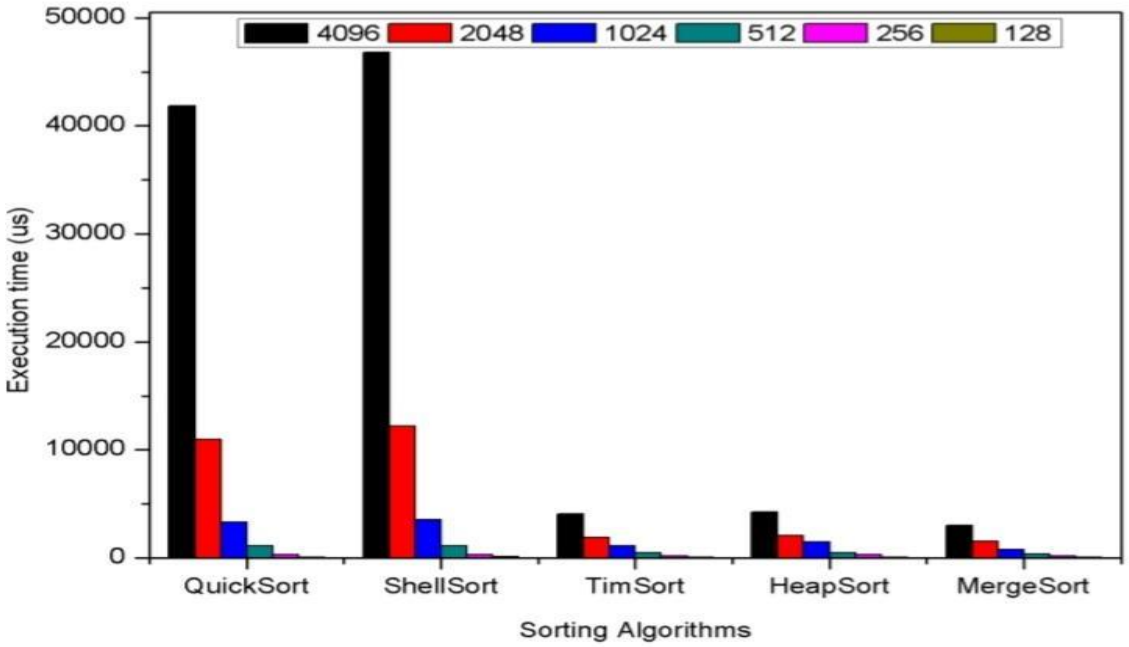


Fig 10. Execution time on the processor

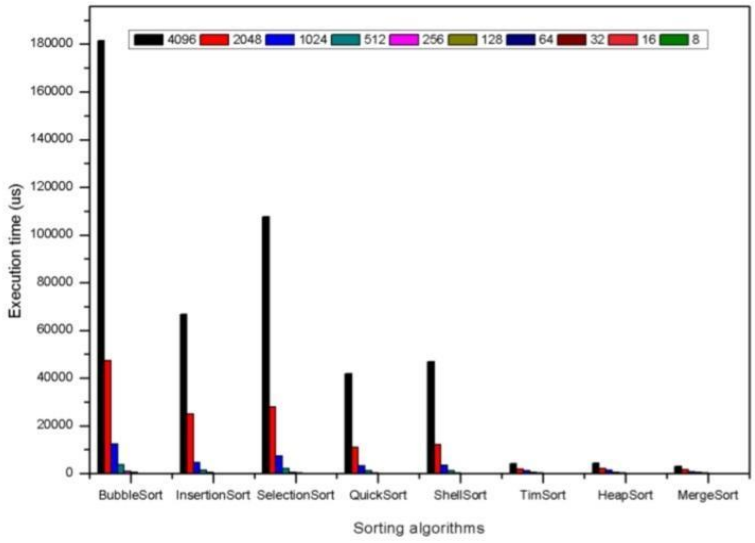


Fig 11-Execution times of "Heapsort", "quick-sort", "Mergesort", Shell-sort and "Timsort" ' algorithms on the processor ,on the same permutation, the coefficient of variation becomes an important measure of reliability On the other hand, relying solely on the standard deviation often does not provide a complete picture of the dispersion of values around the mean. In this context, a .higher coefficient of variation indicates greater variability in the data relative to the mean

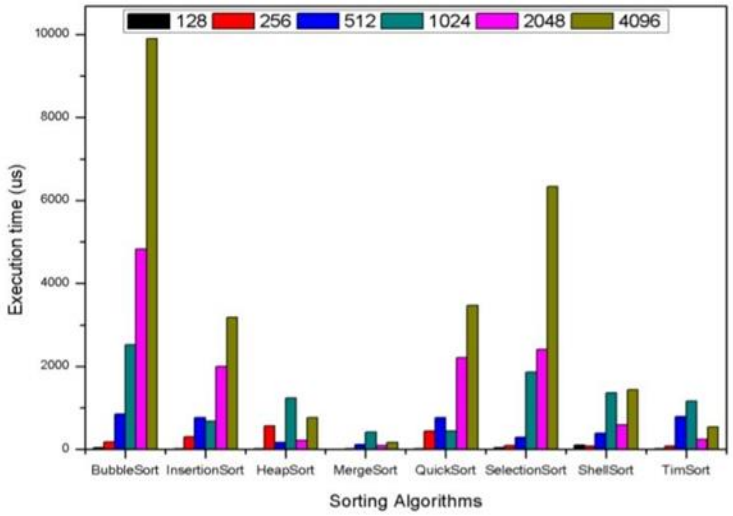


Fig 12-Standard deviation of execution times for sorting algorithms running on the CPU

However, a coefficient of variation of less than 5% is generally considered acceptable, i.e., the mean is considered representative if the population size is large enough (i.e. empirically >= 30) and the distribution of the population median follows Gauss's law. On the other hand, if the coefficient is greater than 5, it is useful to present the results graphically using a box plot that gives an idea of the spread (i.e. the distribution of valid durations from +/- 2 to 30) (see Appendix A). We can conclude that when the coefficient of variation is less than 5%, the stability of the sorting algorithm in terms of execution time is quite remarkable.

Table 3: Coefficient of variation of sorting algorithms on the processor

	BubbleSort (us)	InsertionSort (us)	SelectionSort (us)	HeapSort (us)	QuickSort (us)	ShellSort (us)	MergeSort (us)	TimSort (us)
8	175.35	202.35	175.86	158.144	129.05	164.2	121.18	160.12
16	83.59	131.39	105.65	89.76	93.36	98.635	96.44	127.52
32	29.98	62.93	63.89	49.1	47.74	50.25	45.85	48.91
64	115.79	32.29	24.08	49.1	141.08	161.41	30.94	29.73
128	8.47	10.39	25.82	16.55	14.88	85.18	14.35	12.4
256	17.54	74.91	14.3	188.05	144.05	22.61	12.36	39.46
512	22.33	54.29	13.36	35.94	66.68	33.97	42.14	154.3
1024	20.49	14.86	25.57	82.42	13.29	38.61	52.87	102.07
2048	10.2	8.027	8.61	10.04	20.1	4.8	5.48	12.72
4096	5.45	4.75	5.89	17.66	8.27	3.05	5.34	13.15

Note from Table 3 that the coefficient of variation for the algorithms of sorting is on average greater than 5. For this, we can represent the results in the form of Boxplots (Box and Whiskers) to visualise and compare the distributions of replications on the same scale.

This graphic representation can be a solution to approach the abstract concepts of statistics. To read and interpret a box and whiskers plot, it is necessary to know its construction. The box and whiskers use six values that summarise data: the minimum, the average, and the three quartiles Q1, Q2 (median), which is the data of the series that separates the lower 50% of the data, and Q3, which is the data of the series that separates the lower 75% of the data and the maximum. The quartiles Q1 which is the data of the series that separates the lower 25% of the data, Q2, and Q3 are the essential elements of this graph. To experimentally study the stability of the algorithms, first of all, it is necessary to verify that the percentage of the number of upper outliers is less than 5%.

In addition, it is necessary to verify that the boxplot is symmetrical with respect to the median and the mean. In this case, we can say that this algorithm follows the normal distribution Then, if the value of the median is very different from one distribution to another.

So, the sorting time varies, and the algorithm is not stable depending on the input data. Otherwise, it is stable which is particularly interesting for the targeted real-time applications We used the R language to draw the boxplots This language has a rich set of statistical libraries to compensate for this difference. We drew 80 graphs for each sorting algorithm and more precisely for each element size (8=>4096) in order to visualise and compare the permutations, which aim to change the order of the inputs to obtain several temporal variations in terms of the average of the execution times and the median of the boxplots. Each graph contains 47 permutations (47 Boxplots).

We cannot put all the graphs in this chapter (See Appendix A). As previously stated, the "Mergesort" and insertionsort algorithms are faster in terms of execution time and standard deviation if N>64 and N<=64. For this, we present in this part the different Boxplots of the "Mergesort" algorithm for the data numbers 128, 256, 512, 1024, 2048, 4096.

'Fig 14 and Table 5 show the five values that summarise the data: Q0 (Min), Q1, Q2 (Median), Q3 and Q4 (Max) for the 47 permutations. The values are the comparison criteria between the different distributions.

On the one hand, we observe in ' Fig 14 outliers that provide information about possible measurement errors (e.g. transmission error, operating system noise error, I/O buffering) These values are defined by Chandola as "data patterns that do not conform to a well-defined notion of normal behavior." This definition is very general and is based on how observations or patterns differ from normal behavior. In other words, anomalies represent data that fall outside a specific normal range. ' Fig 13 shows an example of five anomalies labeled V, W, X, Y, and Z that are clearly isolated from the normal data range.

Table 4. Outlier percentages for the "Mergesort" algorithm

Sizes/Number of outliers		total outliers	of higher outliers %
	128	9,5	5
	256	4	3,4
Insert sort	512	3,91	2,7
	1024	3,93	3,9
	2048	2,77	2,7
	4096	8,18	8,18

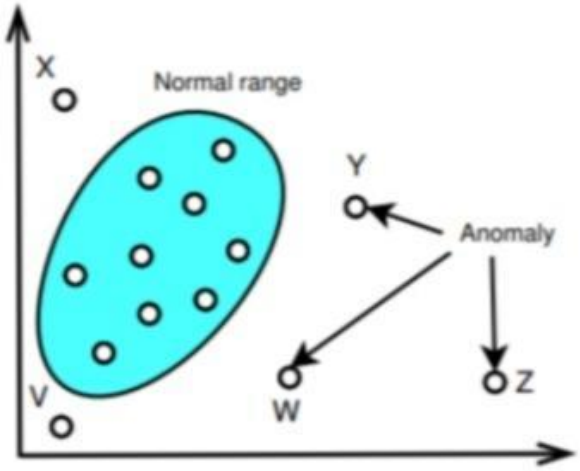


Fig 13: Outlier numbers

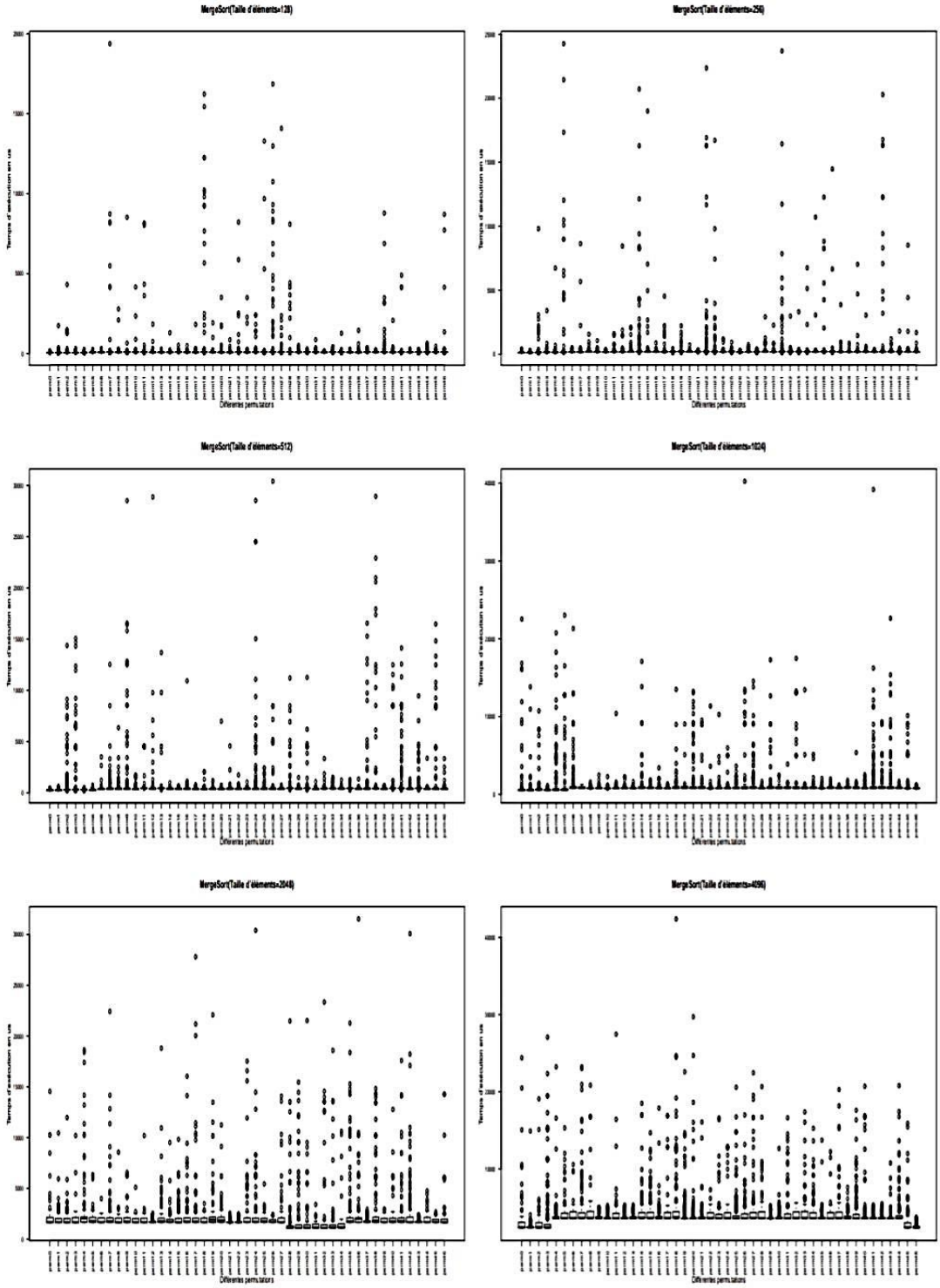
Subsequently, ' Fig 14 shows the asymmetry of the whiskers We observe that the number of outliers is significant. For this, we calculated the percentage of outliers (especially on the upper whisker) for each element size.

Table 4 illustrates the percentages of outliers for each size of the permutations note that the percentage for N = 128/256/512/1024/2048 is varied between 2% and 4% Therefore, we can conclude that these distributions follow the normal law On the other hand, the number of outliers is significant for N = 4096 for large data size.

Then, ' Fig 15 presents the temporal variations for N>64 of the median and the mean of the permutations found when we launch 1000 iterations, we note that the values of the median are almost close in most cases; on the other hand the value of the mean varies We can deduce that the mean is a measure that is sensitive to outliers at the data level calculated from the following formula:

Outliers _Percentage = 100* N_Out/N_R*N_P (8)

With N Out is the number of upper outliers, N_R is the number of replication used in this work (set to 1000) and N_P is the number of permutations, which is set to 47 From Equation 8, we calculate the value of the percentage for all sorting algorithms. We find that this value increases with the increase of the number N.



'Fig 14-Boxplots of the "Mergesort" algorithm with N>64

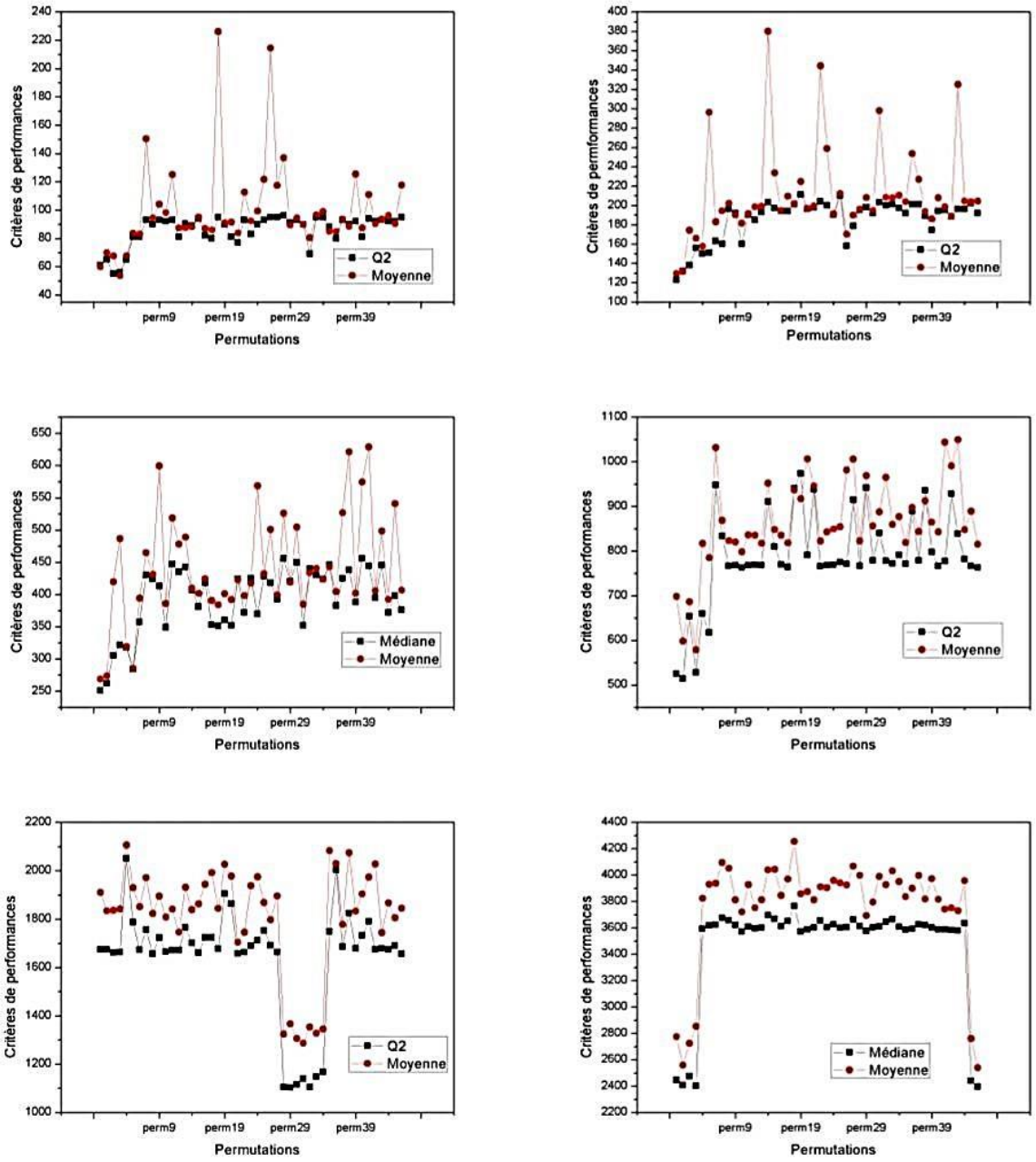
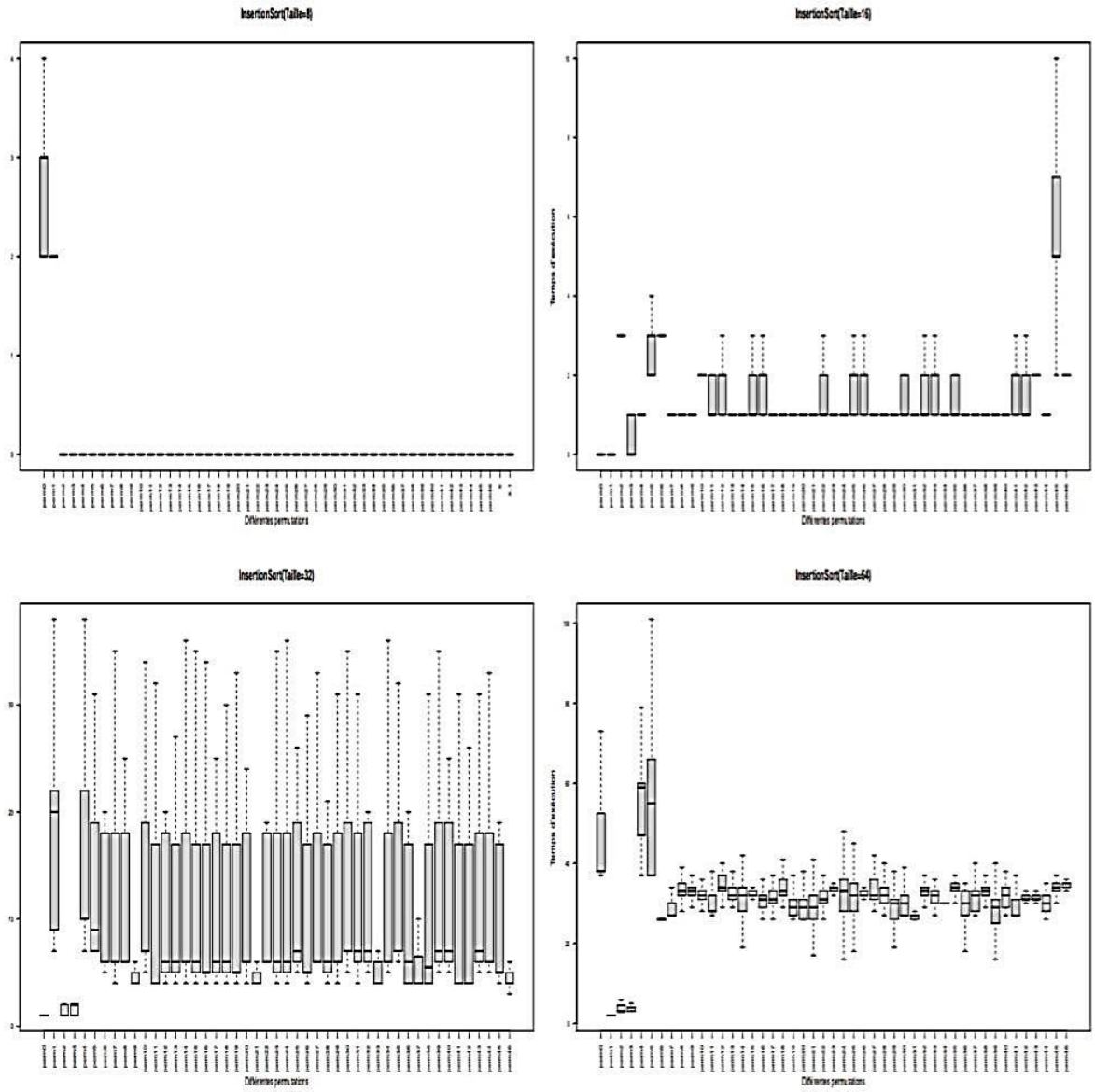


Fig: 15 Median and Mean of "Mergesort" algorithm with N>64

Subsequently, we statically study the Insertion sort algorithm by taking in this case N=8, 16, 32, and 64. Then, we obtain the results displayed in 'innings. 16 and Table 7. Table 7 shows that the values for Q0, Q1, Q2, Q3, and Q4 tend towards 0 for the element size equal to 8, i.e. if IQR-Q3-Q1-0 then the standard deviation is close to 0, so the durations are almost identical in this case Moreover, we choose another performance criterion, which is the value of the mean We observe that the box plots are not symmetrical for the median We now move on to calculate the percentage of outliers.



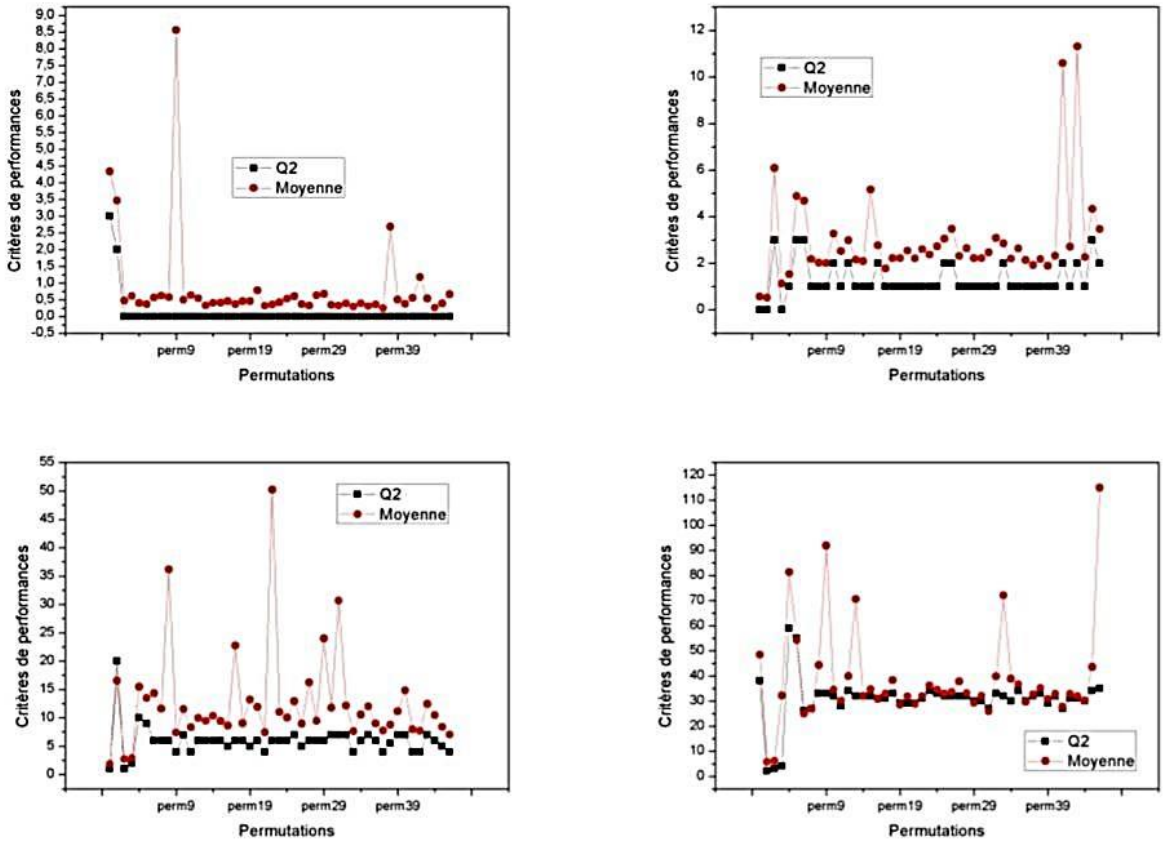
'Fig 16 Boxplots of the Insertion sort algorithm with N<=64

Table 7: Percentage of outliers for Insertion sort

Sizes/Number of outliers		total outliers	of higher outliers %
	8	4,8	4,5
	16	12,3	11,3
Insert sort	32	3,5	3,5
	64	16,13	9,3

Table 7 shows that the percentage of outliers is significant for N = 16 and N = 64, which means that these distributions do not follow the normal law.

' Fig 17 shows the



'Fig 17: The Median and Mean of the Insertion sort algorithm with N<=64

results obtained for the median and the mean of the Insertion sort algorithm if N<64 We note that the value of the median remains almost constant with R = 1000 replications Conversely, the average execution times vary due to hardware fluctuations. To facilitate a comparison of the experimental measurements of the time complexity of the sorting algorithms, we calculated the average execution times for each algorithm based on the array size. This approach provides insight into how the execution time changes in relation to N To this end, we also calculated the dispersion of the execution times, and the results are presented in ' Figs 18 and 19.

_Statistical dispersion

Statistical dispersion arises when repeated measurements of the same quantity are taken. Specifically, if we measure the same phenomenon multiple times using a sufficiently accurate device, we may obtain varying results. This variation can be attributed to external disturbances or, in cases of highly precise measurements, to the inherent randomness of the phenomenon itself. The error resulting from statistical dispersion can be estimated using the following formula:

D = k * Standard deviation (9)

With k representing a constant value. In physics, k is often defined as 3, which corresponds to a confidence interval of 99.73%; that is, 99.73% of the values xi are within the confidence interval and 0.27% will be outside this interval. For each algorithm, we measure both the observed margin of error due to statistical dispersion and the calculated margin of error due to statistical dispersion.

Observed margin of error The observed margin of error is calculated as follows:

- We have a data set to sort. We calculate the upper limit (with formula 2) as well as the lower limit (with formula 3) using the following formulas:

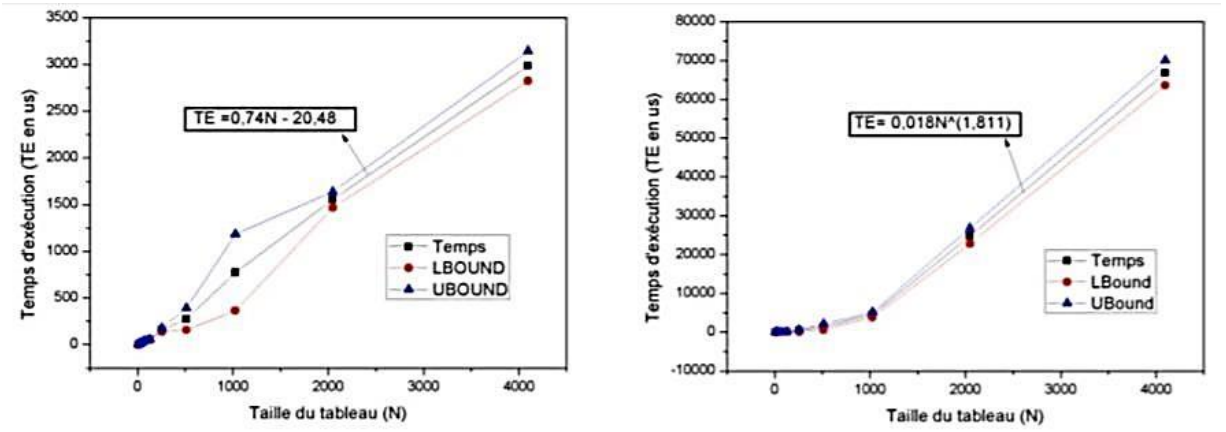
LBound = Time - Standard Deviation (10)

U Bound = Time + standard deviation (11)

Time refers to the mean of the time series created from the execution time values, while standard deviation represents the standard deviation of the same series. After calculating these two metrics, we verify whether all samples fall within the specified confidence level

Calculated margin of error: The margin of error calculated due to statistical dispersion is calculated using the formula.

In the following, we discuss the time variation results for each sorting algorithm as shown in Figs 18 and 19.



' Fig 18 "Mergesort" ' Fig 19 Insertion sort

' Fig 18 shows the time variation of the "Mergesort," LBound, and UBound algorithms as a function of N. We observe that the time curve is almost linear. On the other hand, Figs'. 19 and 17 show that the function of the Insertion sort algorithm is quadratic in terms of N, and the value of the median remains constant. In this case and from the Boxplots, we can give an idea about the stability of the "Mergesort" and Insertionsort algorithms.

Conclusion

We present an overview of sorting algorithms ("Bubble Sort," Insertion Sort, Selection Sort, "Heap Sort," "Quick Sort," Shell Sort, "Merge Sort" and "Timsort"). We present a preliminary study on CPUs, and the results show that the "Mergesort" algorithm is more efficient and stable in terms of execution time and standard deviation when the number of elements is greater than 64. Otherwise, Insertion Sort is chosen as the best algorithm. This is due to the complexity of the sorting algorithms. We present the results using box plots to visualise and compare different permutations.

Reference

1. CLÉMENT, J., NGUYEN THỊ, T., AND VALLÉE, B. Realistic analysis of the Quickse- lect algorithm. To be appear in ToCS (2021,p48
2. CLÉMENT, J., THI, T. H. N., AND VALLÉE, B. Towards a realistic analysis of some popular sorting algorithms. Combinatorics, Probability & Computing 2021,p83
3. FILL, J. A., AND NAKAMA, T. Analysis of the Expected Number of Bit Comparisons Required by Quickselect. Algorithmica (2020),p173
4. FLAJOLET, P. The ubiquitous digital tree. In STACS 2006, 23rd Annual Symposium on Theoretical Aspects of Computer Science, Marseille, France, February ,2020,p21

5. FLAJOLET, P., ROUX, M., AND VALLÉE, B. Digital Trees and Memoryless Sources: from Arithmetics to Analysis. Proceedings of AofA 10, DMTCS, 2021,p190
6. FLAJOLET, P., AND SEDGEWICK, R. Analytic Combinatorics. Cambridge University Press, 2022,p21
7. KNUTH, D. E. The Art of Computer Programming, Volume I: Fundamental Algo- rithms, 3rd Edition. Addison-Wesley,2021,p111
8. KNUTH, D. E. The Art of Computer Programming, Volume III: Sorting and Search- ing, 2nd Edition. Addison-Wesley,2020,p35
9. NÖRLUND, N. E. Lectures on linear equations with finite differences. In Collection of monographs on the theory of functions. Gauthier-Villars, Paris,2020,p50
10. Roux, M. Information theory, Dirichlet series and analysis of algorithms. PhD thesis, University of Caen Basse Normandie, 2011,p48
11. ROUX, M., AND VALLÉE, B. Information theory: Sources, Dirichlet series, and realistic analyses of data structures. In WORDS (2011), pp. 61
12. SEDGEWICK, R. Quicksort. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 2021,p197
13. SEIDEL, R. Data-Specific Analysis of String Sorting. In Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA (2010),p67