The Review and Experimentation on the Fast-Insertion-Sort Algorithm

Halomoan Geraldo - 2106720885

Raizaz Achmad Asyraf - 2106657815

Faculty of Computer Science
University of Indonesia
2023

Abstract - This paper will cover the analysis and evaluation of the maximum length of array to achieve the optimum time complexity when sorting with Fast-Insertion-Sort. The main goal of this study is to confirm and compare the Fast-Insertion-Sort with other sorting algorithms. Our method of analysis will be through the execution of the Fast-Insertion-Sort and other competitive sorting algorithms on an unsorted array and examining the time complexity of each of these sorting algorithms by the means of experimentation.

1. Introduction

There are plenty of sorting algorithms and an abundance of sorting problems today, and unlike back then, newer and possibly more efficient algorithms are available and are being implemented. Naturally, different problems require separate solutions, albeit some overlapping one another. Sorting problems are no exceptions to this philosophy, some algorithms are stable while others are not, the variation of requirements creates the need to alter algorithms to either improve its performance or its approach entirely.

When evaluating algorithms, some factors to consider are the time and space complexity of the algorithm and its stability. The time complexity of an algorithm expresses the relationship between the run time for the algorithm with a given input size. Time complexity is usually represented with the big O notation which indicates the upper bound of the run time of the algorithm. On the other hand, the space complexity of an algorithm focuses more on how much space an algorithm would take when it runs. Similar to the time complexity, the space complexity also uses the big O notation when represented.

In this paper, the center of attention would be on the time complexity of the fast insertion sort algorithm and on how it competes with other competing sorting algorithms. Some of the most used sorting algorithms used today include the Quicksort, Merge sort, Heap sort and Insertion sort.

Nowadays, algorithms with the use of recursion are more favored over the rest, the reason being that by implementing recursive calls, we are approaching the problem with the divide-and-conquer strategy. This strategy involves the division of problems into smaller subproblems of the same form and later combining solutions of each subproblem. This approach allows the algorithm to efficiently solve the problem in a quicker way compared to its iterative counterpart. In contrast, Iterative algorithms do not use the concept of recursion as it does not make any recursive calls.

$QUICKSORT(A, p, r)$
1. $if\ p < r$
2. $\quad then\ q = PARTITION(A, p, r)$
3. $\quad QUICKSORT(A,\ p,\ q - 1)$
4. $\quad QUICKSORT(A,\ q + 1,\ r)$

$PARTITION(A, p, r)$
1. $x = A[r]$
2. $i = p - 1$
3. $for\ j = p\ to\ r - 1$
4. $\quad if\ A[j] \leq x$
5. $\quad\quad i = i + 1$
6. $\quad\quad exchange\ A[i] \leftrightarrow A[j]$
7. $exchange\ A[i + 1] \leftrightarrow A[r]$
8. $return\ i + 1$

Fig. 1. Quick Sort with Lomuto Algorithm. It rearranges the subarray $A[p..r]$ in place.

An example of a recursive algorithm and one of the most used and efficient algorithms would be the Quicksort. The Quicksort works by using an element of a pivot. This pivot is then used to partition the array by iterating over the array from the leftmost element to the second last element and separating elements less and more than the pivot to the left and right side of the pivot respectively. The Quicksort then recursively calls itself to the"left" and "right" side of the array until the array is fully sorted. In contrast, an iterative approach to sorting an array would be the Insertion sort. It is an in-place algorithm that splits the array into two parts, a sorted part and the unsorted part. Insertion sort would iterate through each element in the unsorted part of the array and compare it with each element in the sorted part of the array. It then places the element in the sorted area in its right position.

$INSERTIONSORT$
1. $for\ i \leftarrow 1\ to\ n\ do$
2. $\quad INSERT(A,\ i)$

$INSERT(A, i)$
1. $j \leftarrow i - 1$
2. $v \leftarrow A[i]$
3. $while\ (j \geq 0\ and\ A[j]\ >\ v)\ do$
4. $\quad A[j + 1] \leftarrow A[j]$
5. $\quad j \leftarrow j - 1$
6. $A[j + 1] \leftarrow v$

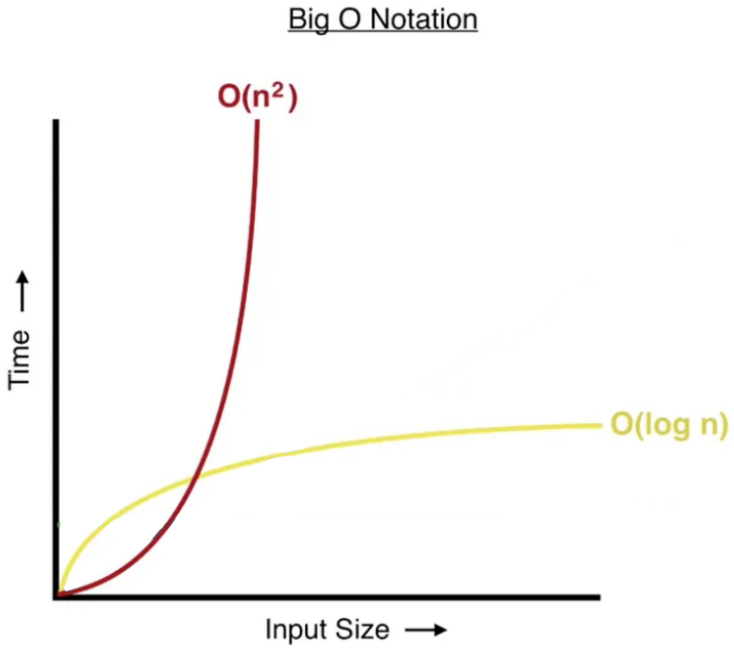
Fig. 2. The pseudocode of Insertion-Sort.

Recursive algorithms generally have a better worse case compared to its iterative counterpart. For example, the Insertion sort has a worst case time complexity of $O(n^2)$ and the worst case time complexity of the Merge sort, which is a recursive algorithm, is O(n log n).

2. Insertion sort

From the surface, it would be no doubt that recursive algorithms would perform better than iterative algorithms such as Insertion sort when looking at their worst case time

complexity. However, despite that, for some input arrays with length n, the run time of the Insertion sort can be faster than some recursive algorithms. While the worst case time complexity of the Insertion sort is $O(n^2)$, it does not mean that it would always be worse than O(n log n). Due to how the big O notation works, O(n log n) and $O(n^2)$ only signifies the upper bound or the worst-case time taken for the algorithm to run.



Since the time complexity shows how much the time taken grows as the input size grows, there should be some input sizes where the run time of the Insertion sort would outspeed the recursive algorithms. In addition to that, the simple and quick comparison of the insertion sort gives it an edge above the recursive approaches when input sizes are relatively small. The Insertion sort also has a smaller overhead in comparison to the "divide and conquer" recursive algorithm as it executes no additional tasks other than its main task, comparing and sorting the elements in the input array.

When the Insertion sort is called, assume that the smallest element of the array is being selected and is about to be inserted into the sorted subarray with length k. As the element being inserted is the smallest element, its position in the sorted subarray would be at the very left or at index 0. This means that all other elements in the subarray would have to move 1 index up. We will also assume that the number of lines taken to compare an element in the subarray and moving it to be the constant c. Therefore, it would take $c \cdot k$ lines to insert an element into the sorted subarray of k elements. If we are to call the *INSERT* function for the very first time, then $k = 1$ and $k = 2$ for the next and so on until the last element where k = n - 1. We will then obtain the total time of:

$$c \cdot 1 + c \cdot 2 + c \cdot 3 + \cdots c \cdot (n-1) = c \cdot (1 + 2 + 3 + \cdots + (n-1))$$

$$c \cdot (n - 1 + 1)((n-1)/2) = cn^2/2 - cn/2$$

3. Fast-Insertion-Sort

The idea of the Fast-Insertion-Sort is to modify the Insertion sort to be recursive while maintaining the quick sorting of insertion sort for shorter inputs. After reading the research done by Simone Faro et al. [1], we noticed that there were two kinds of Fast-Insertion Sorts both with similar but different approaches. The first type is the nested Fast Insertion sort and it works by first checking whether the length of the array is greater than 2^{h-1}, if not then it updates the value of h to $\log_2(n)$. In this case, h is the depth of the algorithm, meaning that the depth of the nested Fast-Insertion-Sort will always be less than $\log_2(n) + 1$. Once this condition is fulfilled, a new variable k is assigned with the value of $n^{(h-1)/h}$. With the newly assigned variable k, the algorithm then iterates through the input array in blocks of size k. For each of the blocks, a recursive call is done with Fast-Insertion-Sort$^{(h-1)}$ and the Insert-Block function is called to arrange the sorted elements of each block back into the correct positions in the main Array.

INSERT-BLOCK$(A,\ i,\ k,\ T)$

1. $for\ j \leftarrow 0\ to\ k\ do$
2. $\quad SWAP(T[j],\ A[i\ +\ j])$
3. $l \leftarrow k\ -\ 1$
4. $j \leftarrow i\ -\ 1$
5. $while\ l \geq 0\ do$
6. $\quad while\ j \geq 0\ and\ A[j]\ >\ T[l]$
7. $\quad\quad SWAP(A[j\ +\ l\ +\ 1],\ A[j])$
8. $\quad\quad j \leftarrow j\ -\ 1$
9. $\quad SWAP(A[j\ +\ l\ +\ 1],\ T[l])$
10. $l \leftarrow l\ -\ 1$

Fig. 3. The pseudocode of INSERT-BLOCK method for Fast-Insertion-Sort

FAST-INSERTION-SORT-NESTED$(A,\ n,\ h)$

1. $if\ n \leq 2^{h-1}\ then\ h \leftarrow \lceil(lg(n))\rceil$
2. $k \leftarrow n^{(h-1)/h}$
3. $T \leftarrow array[k]$
4. $for\ i \leftarrow 0\ to\ n\ (step\ k)\ do$
5. $\quad b \leftarrow min(k,\ n\ -\ 1)$
6. $\quad$ *FAST-INSERTION-SORT-NESTED*$(A\ +\ i,\ b,\ h\ -\ 1)$
7. $\quad$ INSERT-BLOCK$(A,\ i,\ b,\ T)$

Fig. 4. The pseudocode of nested Fast-Insertion-Sort.

The other type of Fast-Insertion-Sort takes a simpler approach by just checking whether n is less than or equal to 2. If True, then a simple Insertion Sort is called to the array to sort it. By using this approach, the base case of this recursive algorithm is the simple Insertion Sort and as we know, the Insertion sort is the fastest sorting algorithm for sorting

very short arrays. In addition to that, since the length of the array will always be 2 or 1, the Insertion sort will only have to compare 2 or 1 element therefore it will run at O(1) time complexity. The reason for this is that the Insertion sort algorithm will only have to iterate through the array once and in that iteration, it will only have two outcomes. One where the second element is smaller than the first or when the second element is greater than or equal to the first. For the first case, after the iteration, the algorithm will insert the second element to the first position. This additional operation is very fast and is negligible to the overall runtime, with this reasoning we can assume that the Insertion sort will run at O(1).

FAST-INSERTION-SORT-RECURSIVE$(A,\ n)$
1. $if\ n \leq 2\ then$
2. $\quad return$ *INSERTION-SORT*$(A,\ n)$
3. $h \leftarrow \lceil (log_c n)) \rceil$
4. $k \leftarrow n^{(h-1/h)}$
5. $T \leftarrow array[k]$
6. $for\ i \leftarrow 0\ to\ n\ (step\ k)\ do$
7. $\quad b \leftarrow min(k,\ n\ -\ 1)$
8. $\quad$ *FAST-INSERTION-SORT-RECURSIVE*$(A\ +\ i,\ b)$
9. $\quad$ *INSERT-BLOCK*$(A,\ i,\ b,\ T)$

Fig. 5. The pseudocode of the recursive Fast-Insertion-Sort.

The next step of this algorithm is similar to the nested Fast-Insertion-Sort where a *k* is obtained. For this algorithm, a *h* is first assigned with value $\log_c(n)$ where c is the partitioning parameter and $c \geq 2$. To clarify the partitioning parameter, the constant *c* will represent the base of the logarithm used to calculate the recursion level and the block size. For example, if *c* is set to 2, that means the binary logarithm to find the recursion level of the algorithm and its block size to iterate through the array. The algorithm then iterates through the array in blocks of *k* to recursively call the Fast-Insertion-Sort again on each block and uses the Insert-Block to insert each sorted block into the proper and sorted positions.

The main concept of the Fast-Insertion-Sort algorithm is that if $c^{h-1} < n \leq c^h$, then the input array will be partitioned into at most *c* blocks of size *k*. This allows us to know how many blocks the input array will be split into and also the size of each block. Furthermore, this gives us the ability to play around with the partitioning parameter to change how many elements per block should the algorithm partition. An example of this would be to decrease the block size *k* by increasing the partitioning parameter *c*. The possibility of this lets the user slightly change the algorithm that may possibly increase the efficiency and the overall runtime of the Fast-Insertion-Sort algorithm.

The Insert-Block function works by first swapping the elements between the block in the input array *A* with index *i* with its counterpart in the temporary array *T*. This moves the block from array *A* to *T*. It then initializes two pointers: *l* is set to the last index of the block in T, and *j* is set to *i* - 1, pointing to the index before the block in A. In other words, it performs swap and comparison operations strategically and inserts the sorted block back into the main array *A* while maintaining the order of elements.

After understanding the recursive version approach of the Fast-Insertion-Sort, we can analyze the algorithm deeper and calculate the time complexity. For input array size *n* greater than 2, the algorithm first determines the recursion level *h* from the logarithm of base *c* and also calculates the block size *k* where both of these steps have a time complexity of O(1). The creation of temporary arrays then happens where this step takes O(k) time. The partitioning of the array then takes place and this iteration is done n/k times. Each of the blocks which was produced from the iteration then recursively calls the Fast-Insertion-Sort algorithm on itself. Next, the Insert-Block function is called and since the swapping process is done for each element in the block, this step has a time complexity of O(k). Finally the sorted blocks are then merged again.

From this analysis, we get the following result for the the recursion:

$$T(n) = T(\frac{n}{k}) + O(n) + O(k)$$

Solving the recursion, we get that the Fast-Insertion-Sort algorithm has a time complexity of O(n log n) for its worst case and its average case.

4. Re-implementation Difficulties

The pseudocode was the foundation of the algorithms that are used for testing. There was a significant flaw in the pseudocode. It was pointed out in the technical report of the paper by Thomas[4]. To sum it up, there was a typo in the pseudocode that misrepresented the last index of an array, making the code not work as it should.

There was also an improvement that can be made in the pseudocode that was present in the source code. The condition to initiate the insertion sort should be $n \leq k \; or \; k \leq 5 \; or \; h \leq 1$, instead of just $n \leq 2$. This is because with the condition stated, the array is short enough that an insertion sort is faster than to call recursively again. Without this improvement, the algorithm runs slowly. The improvement was added in the testing code.

Note that we tried to re-implement the code with minimal input from the source code, as it is already optimized to run on C++ and we want the test code to be as similar to the pseudocode as possible. Also, Thomas' technical report helped in making the testing code. It is easier to re-implement when there exists an implementation of a code that is written in the simple programming language Python.

5. Experimentation

In the experimentation part, we compare the Fast-Insertion sort to other competitive sorting algorithms. We chose Quicksort, Mergesort, and Timsort mainly because these sorting algorithms use divide and conquer methods much like Fast-Insertion sort. Also similar to Fast-Insertion sort, Timsort uses insertion sort to sort the smaller subsets of the array.

For the Fast-Insertion sort, there are pseudocodes for the nested version and the recursive version. As mentioned in section 4, there is an optimized implementation of the latter version provided in Faro et al.'s, source code in C++ format. However, we chose not to benchmark the source code as it is already optimized and differs from the pseudocode given. Hence, we only implemented the algorithms from the pseudocode. Below are the details of the algorithms used for benchmarking:

- **Fast-Insertion Sort Nested (FIS-N)**, the variant of the nested algorithm supposed to have a worst case of $O(n^{1+1/h})$ and average case of $O(n\ log\ n)$ with $h = 5$.
- **Fast-Insertion Sort Recursive (FIS-R)**, the variant of the recursive algorithm supposed to have a worst case of $O(n^{1+\varepsilon})$ and average case of $O(n\ log\ n)$ but claims to be faster than the nested variant, with $c = 5$.
- **Quicksort (QS)**, the Lomuto partition version that have an average case of $O(n\ log\ n)$ and worst case of $O(n^2)$. For this benchmark, the pivot is always the last index..
- **Mergesort (MS)**, a divide and conquer recursive sorting algorithm that is supposed to have an average case of $O(n\ log\ n)$.
- **Timsort (TS)**, a better version of Merge Sort that uses insertion sort to the subsets of the array, supposed to have a average case of $O(n\ log\ n)$.

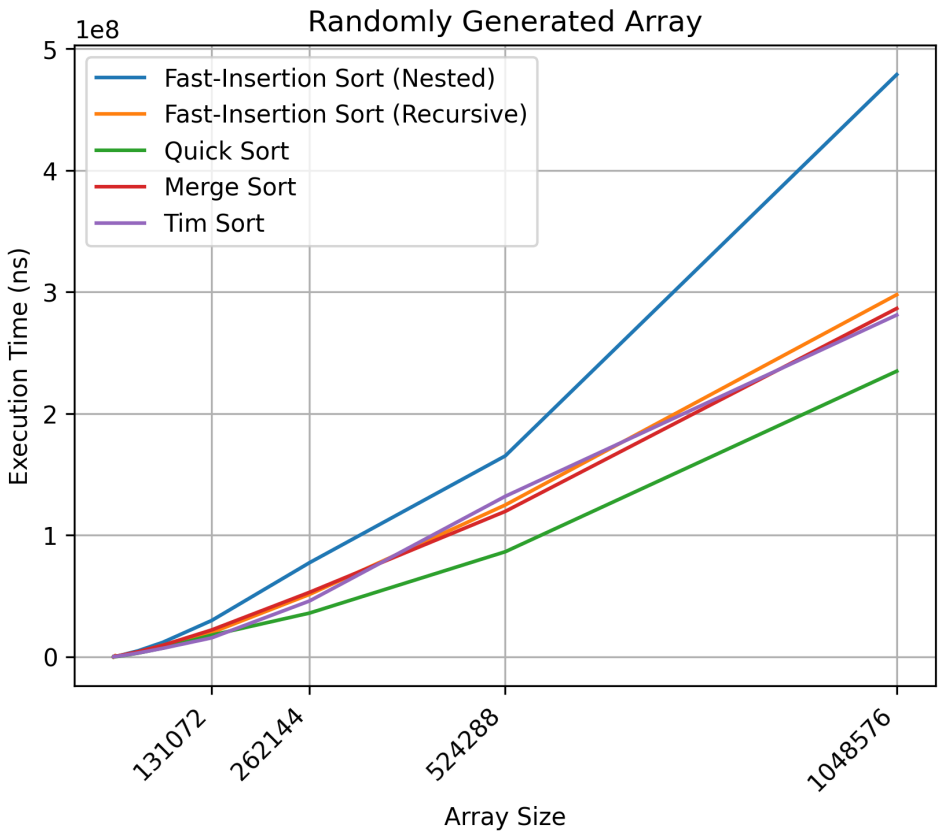
We implement the algorithms above by using the Java programming language that ran on Java (JDK) version 17. All tests have been performed in a Laptop with an AMD Ryzen 9 5900HX processor with only one thread used and running Windows 10. The running time have been recorded with Java's built in $System.nanoTime()$ method before and after the algorithms to signify the difference of time.

Similar to the source paper, we tested the algorithms with an array of size $n = 2^i$, where $2 \leq i \leq 20$. The arrays themselves are randomly generated with a uniform distribution of 2^{32} using Java's built in random library, and reversely sorted array. For each value of n, we tested the algorithm for 100 times then mean over it to get the data. Every element of the array is generated before every iteration of the sorting.

All of the sorting algorithms that are implemented in the benchmark follows the pseudocode found online, then code to Java. Because of this, the algorithms tested may underperform, as it is not optimized. Much the case with both of the Fast-Insertion-Sort algorithms. The algorithms that are tested in Faro et al.'s paper, are optimized, making the pseudocode and data shown quite misleading.

Size	FIS-N	FIS-R	QS	MS	TS
4	2841	1352	1189	2855	2562
8	1959	3712	2105	2228	1203
16	3630	3842	1185	3949	1028
32	8070	2888	2490	7749	4827

Table 1. Benchmark data of sorting randomly generated array of small size (2^i, $where\ 2 \leq i \leq 5$)

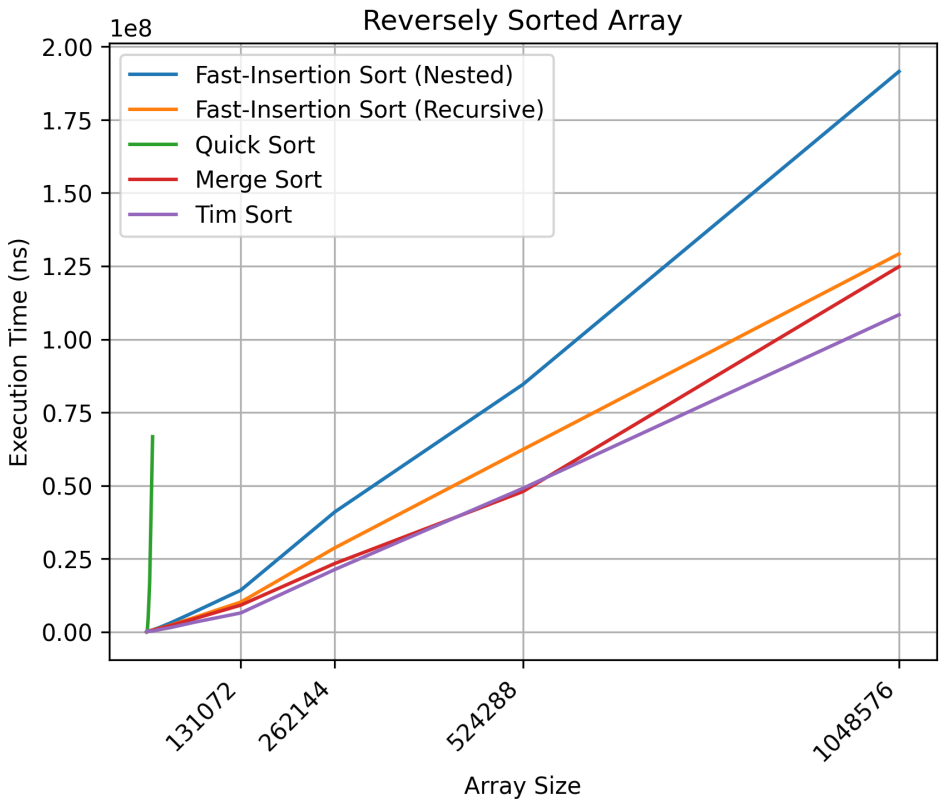


Graph 1. Line graph of the result from the benchmark of the sorting randomly generated arrays.

The Nested Fast-Insertion sort algorithm is competitive for size when $i \leq 13$. Though, that is not true for the recursive version. Even on a high size array, the recursive version still performs similar with other sorting algorithms that has $O(n \log n)$ time complexity. Though it does not outperform Quicksort in any size case. Quicksort still has the best performance for an $O(n \log n)$ algorithm in the scope of the benchmark.

Size	FIS-N	FIS-R	QS	MS	TS
4	3176	1142	2420	3880	2833
8	2580	2199	2874	2830	1761
16	3463	3403	2971	4604	2216
32	6525	5135	7408	13892	5263
64	16373	9141	38556	9369	8941

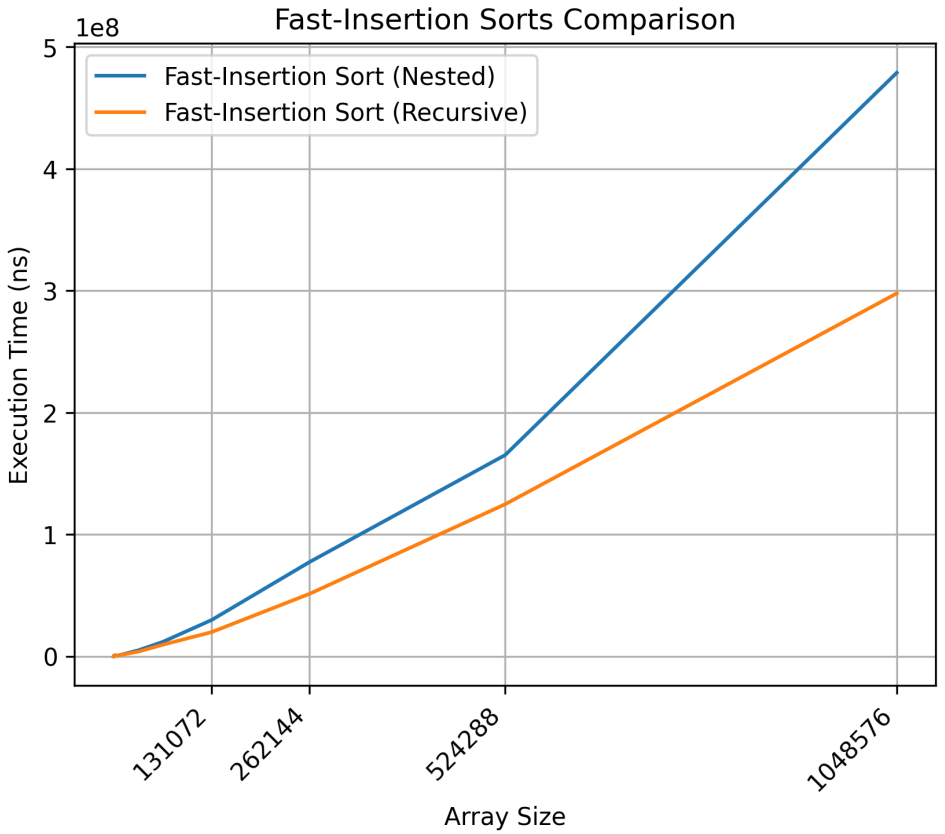
Table 2. Benchmark data of sorting reverse order array of small size (2^i, $where\ 2 \leq i \leq 6$)



Graph 2. Line graph of the result from the benchmark of the sorting reversely ordered arrays.

The sorting algorithms apart from Quicksort run well on reverse order arrays. The rankings of the algorithms are similar to the randomly generated arrays case, but at a faster scale. Nested Fast-Insertion sort is still the slowest algorithm and recursive Fast-Insertion sort, Mergesort, and Timsort are competitive to each other. Though Timsort is still better for most cases than the aforementioned algorithms.

Quicksort on the other hand, runs terrible in reverse order arrays, as expected. It has a time complexity of $O(n^2)$ when run on this condition. It runs so poorly that the algorithm stops the benchmark after the iteration $i = 14$, because the Java compiler raised a StackOverflow exception. Making both of the Fast-Insertion sort to be a more flexible option than Quicksort.



Graph 3. Comparison of the Fast-Insertion Sort Algorithms

If we take a closer look at the two Fast-Insertion sort algorithms, the recursive version is significantly faster than the nested version. This is at the cost of the stableness of the algorithm. For the recursive version, there is no guarantee that it will be stable, as there is a swap operation performed in the storage area.

6. Conclusion

From our analysis of the Fast-Insertion-Sort algorithm, we obtained an average and worst case time complexity of O(n log n). We also understood that the Fast-Insertion-Sort uses the normal Insertion sort algorithm for its base case with the reasoning that the Insertion sort will sort in O(1) time complexity. For input arrays longer than $n = 2$, the algorithm will iterate through the input array to partition the array into blocks of size k which is determined beforehand using the partitioning parameter specified by the user. These blocks will then recursively call the Fast-Insertion-Sort algorithm on itself until it reaches the base case where the normal insertion sort algorithm is used. To merge the partitioned blocks, the Insert-Block function is called to efficiently join these blocks.

With the result we obtained in the experimentation stage, we noticed that the Fast-Insertion-Sort algorithm did run on O(n log n). However, the data shown are not consistent with the original paper's result. We are not sure whether this is an error on our implementation or whether it is a hardware issue. Despite that, it is proven that the Fast-Insertion-Sort can compete with other algorithms such as the Timsort and the Merge sort and only lose to the Quicksort. Even with a smaller input array, the Fast-Insertion-Sort still

took longer than the Quicksort and also the Timsort. A possibility for this inconsistency may be due to the difference in programming language used. Our implementation uses the Java programming language and can be found here https://github.com/raizazaa/paper-daa-fast-insertion-sort.

References

1. Faro, S., Marino, F. P., & Scafiti, S. (2020). Fast-Insertion-Sort: a New Family of Efficient Variants of the Insertion-Sort Algorithm. In SOFSEM (Doctoral Student Research Forum) (pp. 37-48) https://www.dmi.unict.it/faro/papers/conference/faro55.pdf
2. Mridha, P., & Datta, B. K. (2021). An Algorithm for Analysis the Time Complexity for Iterated Local Search (ILS) https://www.questjournals.org/jram/papers/v7-i6/I07065254.pdf
3. Kaiser, C. (2020). Big O Time Complexity: What it is and Why it Matters For Your Code https://levelup.gitconnected.com/big-o-time-complexity-what-it-is-and-why-it-matters-for-your-code-6c08dd97ad59
4. Thomas, E. (2021). Analysis of Fast Insertion Sort https://libraetd.lib.virginia.edu/downloads/mc87pr030?filename=Thomas_Eric_Technical_Report.pdf
5. C. A. R. Hoare, Quicksort, The Computer Journal, Volume 5, Issue 1, 1962, Pages 10–16, https://doi.org/10.1093/comjnl/5.1.10
6. Parves, F. (2022) Merge Sort Using C, C++, Java, and Python | What is Merge Sort and Examples of it? https://www.mygreatlearning.com/blog/merge-sort/
7. Skerritt, A. (2022) Timsort—the fastest sorting algorithm you've never heard of https://skerritt.blog/timsort/
8. R. Srivastava, T. Tiwari, S. Singh, Bidirectional expansion - Insertion algorithm for sorting, Second International Conference on Emerging Trends in Engineering and Technology, ICETET, ISBN: 9780769538846, pp. 59-62 (2009). http://dx.doi.org/10.1109/ICETET.2009.48.