Analysis of Fast Insertion Sort

A Technical Report
presented to the faculty of the
School of Engineering and Applied Science
University of Virginia

by

Eric Thomas

April 24, 2021

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Eric Thomas

Technical advisor: Lu Feng, Department of Computer Science

Analysis of Fast Insertion Sort

Efficient Variants of Insertion Sort

Eric Thomas
University of Virginia
et6sw@virginia.edu

ABSTRACT

As a widespread problem in programming language and algorithm design, many sorting algorithms, such as Fast Insertion Sort, have been created to excel at particular use cases. While Fast Insertion Sort has been demonstrated to have a lower running time than Hoare's Quicksort in many practical cases, re-implementation is carried out to verify those results again in C++ as well as Python, which involves comparing the running times of Fast Insertion Sort, Merge Sort, Heapsort, and Quicksort on random and partially sorted arrays of different lengths. Furthermore, the advantages and disadvantages of Fast Insertion Sort in addition to other questions, flaws, insights, and directions for future work is elaborated on. Fast Insertion Sort offers improved worst-case time complexity over Insertion Sort, while maintaining Insertion Sort's advantages of being simple, stable, and adaptive.

1 Introduction

Although Insertion Sort has many advantages when compared to other comparison sorting algorithms, such as being in-place, adaptive, stable, and online (that is, Insertion Sort can sort an array as it is received) as well as having a worst-case space complexity of $\theta(1)$ (not including the input array), Insertion Sort has the downside of having a high worst-case time complexity of $\theta(n^2)$. Nonetheless, Insertion Sort has small constants, which allows it to be faster than most other sorting algorithms for small-sized input arrays. Currently, there are numerous sorting algorithms that are either variants of Insertion Sort (for example, Shellsort, Tree Sort, and Library Sort) or switch to Insertion Sort when the input array is small enough (for example, Timsort, Introsort, and Block Sort), thus accomplishing better average- and worse-case time complexity and preserving some, but not all, advantages of Insertion Sort.

Out of the aforementioned sorting algorithms, only Timsort and Block Sort are adaptive, stable, and achieve lower worst-case time complexity than Insertion Sort, though the former has a somewhat undesirable worst-case space complexity of $O(n)$ and the latter suffers from complicated implementation. According to Faro et al. [1], there does not seem to exist an iterative, in-place, and online variant of Insertion Sort that manages to have a worst-case time complexity of $o(n^2)$. Originating from Faro et al. [1], a family of Insertion Sort variants, referred to as Fast Insertion Sort, is the result of an endeavor to solve these problems.

2 Related Work

After reviewing research by "Mishra et al. [2], Yang et al. [3] and Beniwal et al. [4] [that] analyzed and compared different sorting algorithms in order to help users to find appropriate algorithm as per their need," Goel and Kumar [5] said that "an experimental study by Yang et al. [3] revealed that for smaller sized list, Insertion and Selection sort performs well, but for a larger sized list, merge and quicksort are good performers." As another notable use case for Insertion Sort, Goel and Kumar note that "dynamic sorting of modules in physical memory uses Insertion Sort algorithm to sort free spaces according to their size in fragmented memory," which is to "aggregate free spaces at one location and make room for new modules." Furthermore, they survey multiple improvements of Insertion Sort that have been made by many researchers in the past.

Goel and Kumar [5] explain how "a new sorting algorithm for a nearly sorted list of elements was devised from the combination of IS and quickersort algorithm," which uses "an auxiliary list . . . to remove the unordered pairs from the main list." After "sorting is done using IS or quickersort algorithm," they said that "results of both the lists are then merged to give a final ordered list." They point out another sorting algorithm, called "Fun-Sort," which uses a "repeated binary search approach on an unsorted array for IS and acquired $\theta(n^2 \log n)$ time complexity to sort an array of n elements." Since "gaps are left [in arrays] . . . to accelerate insertions," they mention that "gapped insertion sort," also known as Library Sort or Binary Insertion Sort, "is studied to devise an algorithm with $O(n \log n)$ time complexity with high probability." They touch on "an

improvement for IS . . . proposed by Min [6] as [a] 2-element insertion sort where two elements from unsorted part of a list are inserted into sorted part" as opposed to one-element insertion in Insertion Sort, allowing for improved time complexity but increased space complexity. They cover how "Dutta et al. [7] designed an approach for data which is in opposite order," demonstrating that the "number of comparison operations are much less, i.e., nearly equal to $\frac{3(n-2)}{2}$ in comparison to IS." In addition to an Insertion Sort that is based on 2-way expansion, called "an Adaptive Insertion Sort (AIS) . . . to reduce the number of comparisons and shift operations," they also mention a "Doubly Inserted Sort, an approach similar to the max-min sorting algorithm is used to scan list from both ends." Furthermore, they observe that "Mohammed et al. [8] proposed Bidirectional Insertion Sort (BCIS) based on the concept of IS, left and right comparators," reducing the number of "comparison and shift operations" to the point where the "average case and best case complexity of BCIS is $O(n^{1.5})$ and $O(4n)$ respectively." Lastly, they remark how "there is hardly any variant of IS which has claimed to maintain online sorting feature of IS," with some even deviating "from in-place and stable sort feature of IS to give better performance."

Lam and Wong [9] said that "Rotated Insertion Sort, or just Rotated Sort for short, is based on the idea of the implicit data structure called rotated list," which is "where the relative ordering of the elements is stored implicitly in the pattern of the data structure, rather than explicitly storing the relative ordering using offsets or pointers." They note that "rotated list achieves $O(n^{1.5}\log n)$ operations using constant $O(w)$ bits temporary space, or $O(n^{1.5})$ operations with extra $\theta(\sqrt{n}\log n)$ bits temporary space, regardless of w," where w is the size of a computer word. Additionally, they present their original "Rotated Library Sort that combines the advantages" of Library Sort and Rotated Insertion Sort. Petersson and Moffat [10] elaborate on adaptive sorting algorithms, such as "Straight Insertion Sort" and "Natural Mergesort."

3 System Design

3.1 Explanation of Fast Insertion Sort

The high-level idea of Fast Insertion Sort is to extend a sorted, left subdivision of the input array by inserting a sorted block of a specific size, k, for each iteration, all while maintaining the order during insertion. There are two primary types of Fast Insertion Sort. One is a sequence of nested algorithms, called Fast Insertion Sort $^{(h)}$ or Fast Insertion Sort Nested, where the h^{th} algorithm can be executed when $n > 2^h$ and n is the size of the input array. In this case, h symbolizes the depth of the algorithm nesting, because the h^{th} algorithm recursively calls the $h-1^{\text{th}}$ algorithm, stopping when $h = 0$. The other type of Fast Insertion Sort is a "purely recursive" derivation of Fast Insertion Sort Nested, called simply Fast Insertion Sort or Fast Insertion Sort Recursive, which dynamically computes h [1].

The aforementioned block size, k, is calculated from the following formula where h is the "input size degree" [1]. Since k is the size of an array, $k \in \mathbb{N}$.

$$k = n^{\frac{h-1}{h}} \tag{1}$$

Meanwhile, h is defined such that $h \in \mathbb{N}$ and $h \geq 1$, which is computed from an additional formula below where c is the "partitioning degree," $c \in \mathbb{R}$, and $c > 0$, or a positive floating point in practice [1].

$$h = \log_c n \tag{2}$$

The general intuition is that if $c^{h-1} < n \leq c^h$ (can be found with Formula 2), then Fast Insertion Sort partitions the input array into at most c blocks of size k. In other words, the block size, k, can be either increased by decreasing c or increasing h, or decreased by increasing c or decreasing h. This property allows the user to fine-tune the algorithm according to expected size of the input array.

The procedure of Fast Insertion Sort is a for loop of index i from 0 to $n-1$ where each iteration adds k to i. For each iteration, the minimum of k and $n-i$ is determined, which is used as the new value of k for the rest of iteration. Otherwise, if the case where $n-i < k$ is not accounted for, then the block size, k, would extend past the end of the input array.

Then Fast Insertion Sort recursively calls itself with a new starting index of i and a new input array size of k from the minimum function. In the case of Fast Insertion Sort Nested, the h argument is decremented by 1 in order to call the subsequent nested version of the algorithm. The final operation of each iteration is an "insert block" procedure that takes the sorted block from i to $k-1$ and inserts it into the partition from the starting index of the input array (of the current function call) to $k-1$, while also maintaining the order of that partition [1].

3.2 Properties of Fast Insertion Sort

In addition to keeping Insertion Sort's properties of being adaptive, stable, and online, Fast Insertion Sort achieves a lower worst-case time complexity at the downside of having a higher worst-case space complexity. If a subdivision of the input array is already sorted, the aforementioned insert-block procedure skips an entire loop of i iterations, thus making Fast Insertion Sort adaptive. Because Fast Insertion Sort uses an additional sorted block

array to preserve the original ordering of equal elements and mimics the insertion procedure of Insertion Sort, Fast Insertion Sort is also stable. By sorting from the left to the right and extending the sorted partition of the input array, Fast Insertion Sort is online as well, assuming that the size parameter is updated as needed.

While Faro et al. [1] said that both Fast Insertion Sort Nested and Recursive "achieves an $O(n^{1+\varepsilon})$ worst-case time complexity, where $\varepsilon = \frac{1}{h}$," a notable flaw with the paper is the absence of a proof. However, their statement about how Fast Insertion Sort uses "additional external memory of size $O(k)$" makes intuitive sense, because the additional sorted block of size k is only needed for one insert-block procedure [1]. As such, only one of those supplementary arrays has to be allocated in memory at a time.

3.3 In-Place Fast Insertion Sort

While the usual implementation of Fast Insertion Sort does not sort in-place, it can be modified to do so with some performance trade-offs. Instead of creating an additional array of size k and using that as a storage area for the insert-block procedure, a section of the input array that is to the right of the current working area can be utilized. This working area is specifically from i (in the aforementioned for loop) to $k-1$ in the input array, while the storage area is from $i+k$ to $2k-1$. However, swap operations must also replace the assignment operations in the insert-block procedure, because elements will be overwritten otherwise. When Fast Insertion Sort reaches near the end of the input array and $i > n-2k$, there is not enough additional space for in-place storing, so Faro et al. [1] suggest that the remaining elements can be inserted individually, similar to regular Insertion Sort. Although in-place sorting is indeed possible for Fast Insertion Sort, resorting to swap operations over assignment operations results in worse experimental performance, since each swap operation is made up of three assignment operations.

3.4 Block Insertion Sort

In particular, a notable version of Fast Insertion Sort Nested, called Block Insertion Sort, is found when $h = 2$, because the small number of nested calls makes it simple to convert the algorithm to an iterative implementation that is in-place, online, and adaptive. Block Insertion Sort has $O(n^{1.5})$ worst-case time complexity, based on the aforementioned worst-case time complexity of Fast Insertion Sort. According to Faro et al. [1], Block Insertion Sort is "the first . . . iterative in-place online variant of the Insertion-Sort algorithm [that] achieves a $o(n^2)$ complexity in the worst case."

3.5 Re-Implementation Challenges

In order to re-implement Fast Insertion Sort, the pseudocode provided by Faro et al. [1] was used as the foundation. However, when put into practice, some typos in the pseudocode became apparent, which is a noteworthy flaw of the paper. Specifically, the insert-block function has a for loop from $j = 0$ to $k-1$, where k is the sorted block size, while Fast Insertion Sort has a for loop from $i = 0$ to n, where n is the size of the input array. Since $n-1$ is the index of the last element of the input array, the latter for loop implies that that should be the last iteration. Despite that being the case, the former for loop already subtracts one, which unintentionally ignores the very last element. Therefore, the pseudocode should use either $k-1$ and $n-1$ or k and n as the endpoints to be consistent. Faro et al.'s pseudocode for Insertion Sort contains a similar inconsistency with a for loop that ends at $n-1$ [1].

An additional typo is in Faro et al.'s insert-block function where the swap operation uses T, the sorted block, and h as the index for retrieving the target element from T [1]. However, h is an uninitialized variable in the context of the insert-block function, causing an error. Since the pseudocode already references local variable l as the index that is intended for T, it becomes clear that h should be l.

Another minor challenge with re-implementing the pseudocode arose from the ambiguity around what type of number that each of the aforementioned h, c, and $\frac{h-1}{h}$ are, which is important to know for implementations in statically-typed programming languages. Although Faro et al. [1] mention that "$h \in \mathbb{N}$," some consider zero to be a natural number, which makes it unclear as to whether or not h can be zero. While the division operations that come up later in the paper make it obvious that h cannot be zero, they could be more precise with their initial definition of h to eliminate any possible confusion. Furthermore, the pseudocode for Fast Insertion Nested is recursive and has no base case, which would be easier to understand if they added a simple base case that checked if $h = 0$, then the function returns.

On the other hand, Faro et al. [1] do not clarify anywhere in the paper that c can be any positive real number, which is even more misleading when the experimental tests only include incrementing integer values for c. Only after glancing at their source code where they define c as a float, does it become evident that c is a real number. Since k and h are natural numbers, it seems reasonable to consider the possibility that perhaps $\frac{h-1}{h}$ should be truncated or rounded before calculating Formula 1. Similar to the case with c, this exponential term is defined as a float in Faro et al.'s source code, which would have been beneficial to clarify in the paper [1].

While a goal of re-implementation was to follow the pseudocode as much as possible without referring to Faro et al.'s source code, the initial re-implementation of Fast Insertion Sort Recursive was very slow in experimental tests [1]. In only the source code, they include some significant performance improvements in the form of a base case that checks if $h \leq 1$, $n \leq k$, or $k \leq 5$, then use Insertion Sort instead of continuing with a drawn-out insert-block procedure on very small sorted blocks. These improvements were added to the re-implementation.

Since scope is a significant aspect of programming languages, the scope of the initialization of the sorted block was experimented with. For example, the following locations were attempted, listed from outermost scope to innermost: before Fast Insertion Sort is called (passed in as an initial argument), right before the main for loop (same as the pseudocode by Faro et al. [1]), and right before the insert-block procedure. In C++, right before the main for loop was the most efficient, perhaps because the outermost scope lacked the compiler's local-scope optimizations, and the innermost scope was repeating too much work inside the for loop. Unfortunately, only the innermost scope benefits from $O(k)$ worst-case space complexity, because the other locations are before the recursive call, which means that multiple sorted blocks may be allocated at once.

4 Results

4.1 Re-Implementation Tests

The tests of the re-implementation include many of the same parameters as the tests performed by Faro et al. [1]. Although they test three different versions of Quicksort, only one version of Quicksort is included in the re-implementation, which always picks the last element as the pivot. Aside from that difference, Merge Sort, Heapsort, Fast Insertion Sort Nested (for $2 \leq h \leq 10$), Fast Insertion Sort Recursive (for $2 \leq c \leq 10$), and Block Insertion Sort are included, matching their original tests. Similar to their tests, the tests of the re-implementation are performed on input arrays with a size $n = 2^i$ where $2 \leq i \leq 20$, while the output is the average time of 1000 runs. However, Python tests were restricted to $2 \leq i \leq 16$ due to memory limitations.

The re-implementation code is available at github.com/awolffromspace/fast-insertion-sort. The re-implementation tests for C++ and Python were compiled and interpreted with the GNU C++ Compiler 9.3.0 and Python 3.8.5, respectively. In particular, the C++ tests used Faro et al.'s optimization options: -O2 -fno-guess-branch-probability [1]. Additionally, the re-implementation tests were computed on a PC with a 3.9 GHz 8-core AMD Ryzen 7 3800X processor, in an Ubuntu 20.04 VirtualBox that had 4 cores and 8 GB of memory allocated. The gettimeofday function in POSIX and the timeit module are used to time each sorting algorithm in C++ and Python, respectively. While the clock function in the C++ Standard Library could be used to measure with a hardware cycle counter like in Faro et al.'s tests, gettimeofday is chosen to more closely match Python's timeit module, which records real time rather than CPU cycles [1].

Following Faro et al.'s test format, the re-implementation tests time the sorting algorithms on both random and partially-sorted input arrays [1]. Each random element is generated by the rand and randint functions, where the max is the RAND_MAX constant and sys.maxsize value, in C++ and Python, respectively. Partially-sorted input arrays are generated by taking a sorted input array and swapping elements at random indices for $\frac{n}{4}$ iterations.

4.2 Analysis of Test Results

The most important takeaway from the re-implementation test results in Figures 1, 2, and 3 is that they verify the experimental $O(n \log n)$ running time that Faro et al. [1] observed for Fast Insertion Sort when the most optimal h and c values are used, which ended up being 5 for both on large-sized input arrays. Furthermore, Block Insertion Sort seems to have a polynomial running time as expected. While Faro et al. [1] said that Fast Insertion Sort outperforms Merge Sort and Heapsort, that is not the case for the re-implementation tests until sizes of 2^{17} are reached, which is probably due to the re-implementation being less optimized. The re-implementation's lack of optimization is also noticeable when observing Fast Insertion Sort Nested's poor performance on small-sized input arrays, which is about five times worse than Quicksort until sizes of 2^7 are reached.

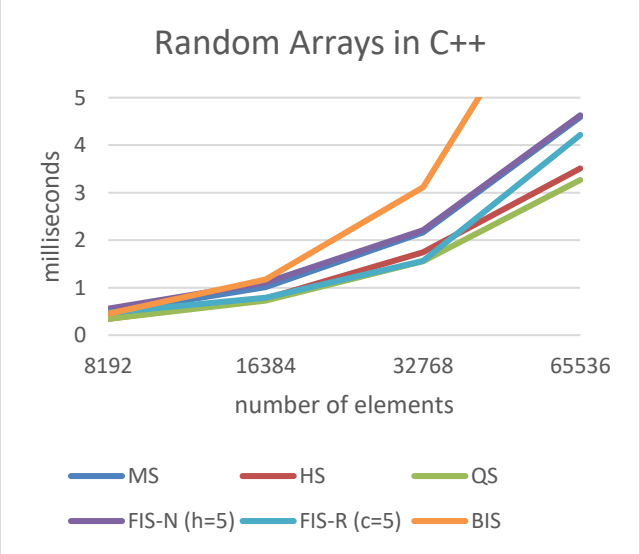

Figure 1: Running times of sorting algorithms on random input arrays in C++

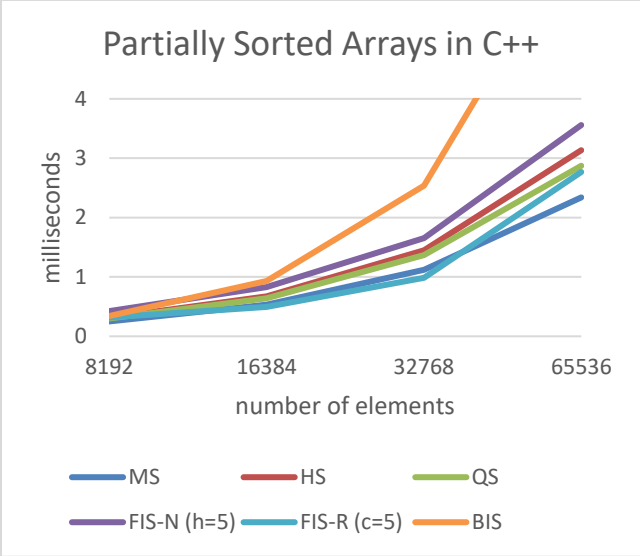

Figure 2: Running times of sorting algorithms on partially-sorted input arrays in C++

As expected, Fast Insertion Sort performs better on partially-sorted input arrays than on random input arrays. Oddly enough, all of the sorting algorithms achieve better results on partially-sorted input arrays.

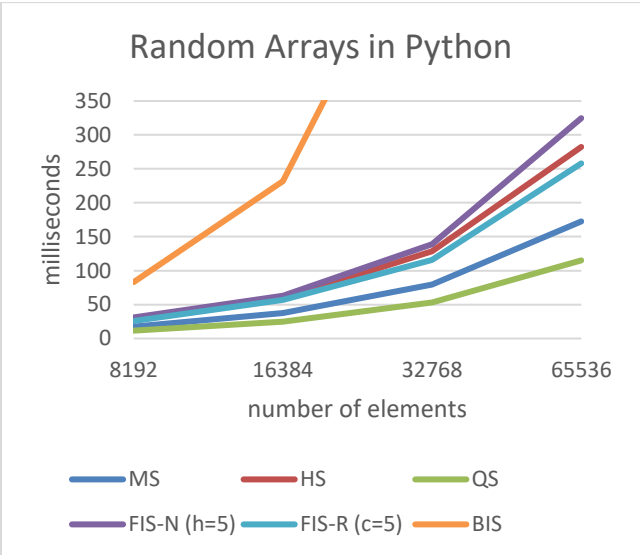

Figure 3: Running times of sorting algorithms on random input arrays in Python

As a consequence of the convenient features that Python provides to users, such as interpreting code on the fly, dynamic typing, and garbage collection, the sorting algorithms perform about 100 times slower on Python compared to C++, as shown in Figures 1 and 2. Python interpreters usually step through the provided code one line at a time, converting the code to bytecode as the interpreter progresses. Because C++ compilers compile all provided code in advance, they are able to make optimizations that require analyzing the entire code, which would be difficult for interpreted programming languages to do without making very accurate guesses as to what the rest of the code would entail. In contrast to Python's dynamic typing, C++ enforces static typing, allowing it to make optimizations that require knowing the type of a given variable. For example, C++ can reduce the amount of memory that a particular data structure uses if the compiler knows the exact sizes of each element, which can pack that data in the cache and memory in a more compact way. Since the data in C++ can be closer together, C++ also takes advantage of performance enhancements, such as locality of reference. Because Python does not require the programmer to free up pointers and references, it has to routinely run a garbage collector, which inevitably adds more operations and thus CPU cycles to program execution.

5 Conclusions

Fast Insertion Sort is a family of Insertion Sort variants that achieve $O(n^{1+\varepsilon})$ worst-case time complexity where $\varepsilon = \frac{1}{h}$, while preserving many of the advantages of Insertion Sort, such as being adaptive, stable, and online. Furthermore, Fast Insertion Sort has an experimental running time of $O(n \log n)$, outperforming some $O(n \log n)$ sorting algorithms, such as Merge Sort and Heapsort, on large-sized input arrays. However, Fast Insertion Sort has the downside of requiring $O(k)$ additional space. At the cost of adding more operations to Fast Insertion Sort, it can be modified to sort in-place. Block Insertion Sort is a notable version of Fast Insertion Sort, because it is purely iterative, in-place, online, and has $o(n^2)$ worst-case time complexity. By comparing the timing of sorting algorithms in C++ and Python, some insight can be found with respect to the advantages and disadvantages of different approaches in programming language design.

6 Future Work

Faro et al.'s claim that Fast Insertion Sort has a worst-case time complexity of $O(n^{1+\varepsilon})$ where $\varepsilon = \frac{1}{h}$ should be proven [1]. While using the logarithmic function to determine the size of each partition of the input array is effective, more research could be done to identify partitioning methods that potentially lead to better performance. Since Faro et al. [1] showed that Block Insertion Sort outperforms a version of Quicksort that uses Insertion Sort as a subroutine, Block

Insertion Sort could replace or accompany Insertion Sort in sorting algorithms, such as Timsort and Introsort. Lastly, an iterative version of Fast Insertion Sort for optimal values of c and h may net slight performance gains.

REFERENCES

[1] Simone Faro, Francesco Pio Marino and Stefano Scafiti. 2020. Fast-Insertion-Sort: A New Family of Efficient Variants of the Insertion-Sort Algorithm. In *Proceedings of the SOFSEM 2020 Doctoral Student Research Forum (SOFSEM 2020)*, January 20 - 24, 2020, Limassol, Cyprus. University of Cyprus, Nicosia, Cyprus, 37-48. https://www.dmi.unict.it/faro/papers/conference/faro55.pdf

[2] Aditya Dev Mishra and Deepak Garg. 2008. Selection of Best Sorting Algorithm. *International Journal of Intelligent Information Processing* 2, 2 (July - Dec. 2008), 363-368. https://www.academia.edu/download/28569137/Selection_of_best_sorting_algorithm.pdf

[3] You Yang, Ping Yu and Yan Gan. 2011. Experimental Study on the Five Sort Algorithms. In *Proceedings of Second International Conference on Mechanic Automation and Control Engineering (MACE 2011)*, July 15 - 17, 2011, Inner Mongolia, China. IEEE, New York, NY, 1314-1317. DOI:10.1109/MACE.2011.5987184

[4] Sonal Beniwal and Deepti Grover. 2013. Comparison of Various Sorting Algorithms: A Review. *International Journal of Emerging Research in Management and Technology* 2, 5 (May 2013), 83-86.

[5] Shubham Goel and Ravinder Kumar. 2018. Brownian Motus and Clustered Binary Insertion Sort Methods: An Efficient Progress Over Traditional Methods. *Future Generation Computer Systems* 86, Apr. 2018, 266-280. DOI:10.1016/j.future.2018.04.038

[6] Wang Min. 2010. Analysis on 2-Element Insertion Sort Algorithm. In *Proceedings of International Conference on Computer Design and Applications (ICCDA 2010)*, June 25 - 27, 2010, Qinhuangdao, China. IEEE, New York, NY, 143-146. DOI:10.1109/ICCDA.2010.5541165

[7] Partha Sarathi Dutta. 2013. An Approach to Improve the Performance of Insertion Sort Algorithm. *International Journal of Computer Science & Engineering Technology* 4, 5 (May 2013), 503-505. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.404.412&rep=rep1&type=pdf

[8] Adnan Saher Mohammed, Şahin Emrah Amrahov and Fatih V. Çelebi. 2017. Bidirectional Conditional Insertion Sort Algorithm: An Efficient Progress on the Classical Insertion Sort. *Future Generation Computer Systems* 71, June 2017, 102-112. DOI:10.1016/j.future.2017.01.034

[9] Franky Lam and Raymond K. Wong. 2013. Rotated Library Sort. In *Proceedings of 19th Computing: Australasian Theory Symposium (CATS 2013)*, January – February, 2013, Adelaide, South Australia. Australian Computer Society, Inc., Darlinghurst, Australia, 21-26. https://www.cse.unsw.edu.au/~wong/papers/cats13.pdf

[10] Ola Petersson and Alistair Moffat. 2005. A Framework for Adaptive Sorting. In *Proceedings of Third Scandinavian Workshop on Algorithm Theory (SWAT 1992)*, July 8 - 10, 1992, Helsinki, Finland. Springer, Berlin, Germany. DOI:10.1007/3-540-55706-7_38