Fast-Insertion-Sort: a New Family of Efficient Variants of the Insertion-Sort Algorithm*†

Simone Faro, Francesco Pio Marino, and Stefano Scafiti

Università di Catania, Viale A.Doria n.6, 95125 Catania, Italy

Abstract. In this paper we present Fast-Insertion-Sort, a sequence of efficient external variants of the well known Insertion-Sort algorithm which achieve by nesting an $O(n^{1+\varepsilon})$ worst-case time complexity, where $\varepsilon = \frac{1}{h}$, for $h \in \mathbb{N}$. Our new solutions can be seen as the generalization of Insertion-Sort to multiple elements block insertion and, likewise the original algorithm, they are stable, adaptive and very simple to translate into programming code. Moreover they can be easily modified to obtain in-place variations at the cost of a constant factor. Moreover, by further generalizing our approach we obtain a representative recursive algorithm achieving $O(n \log n)$ worst case time complexity. From our experimental results it turns out that our new variants of the Insertion-Sort algorithm are very competitive with the most effective sorting algorithms known in literature, outperforming fast implementations of the Hoare's Quick-Sort algorithm in many practical cases, and showing an $O(n \log n)$ behaviour in practice.

Keywords: Sorting · Insertion-Sort · Design of Algorithms.

1 Introduction

Sorting is one of the most fundamental and extensively studied problems in computer science, mainly due to its direct applications in almost all areas of computing and probably it still remains one of the most frequent tasks needed in almost all computer programs.

Formally sorting consists in finding a permutation of the elements of an input array such that they are organized in an ascending (or descending) order. In the comparison based model, it is well-known that the lower bound for sorting n distinct elements is $\Omega(n \log n)$, both in the worst case and in the average case.[1]

A huge number of efficient sorting algorithms have been proposed over the years with different features.[2] There are several factors indeed affecting the selec-

[*] We gratefully acknowledge support from "Università degli Studi di Catania, Piano della Ricerca 2016/2018 Linea di intervento 2".

[†] Copyright© 2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

[1] In this context the average case refers to the *random-permutation model* where we assume a uniform distribution of all input permutations

[2] See [5] for a selected set of the most considerable comparison-efficient sorting algorithms

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

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

Fig. 1. The pseudocode of Insertion-Sort. The algorithm divides the input array in a sorted portion aligned at the left, and an unsorted portion aligned on the right. At each iteration, the algorithm removes one element from the right portion, finds its correct location within the sorted portion, and shifts each element after this location ahead by one position in order to make room for the new insertion. Iterations repeat until all elements have been inserted.

tion of suitable sorting algorithm for an application [13]. Among them the time and space complexity of an algorithm, its stability, its adaptability, its capability to work online, the number of elements to be sorted, their distribution and the percentage of already sorted elements, are considered as direct factors [21] whereas programming complexity and data type of elements are included among the indirect factors [20]. In our discussion we will refer to the time complexity of a sorting algorithm as the overall number of *any* operation performed during its execution.

The space complexity of an algorithm concerns the total amount of working space (in addition to the array to be sorted) required in the worst case. An *in-place* algorithm requires only constant additional space, otherwise it is said to be an *external* algorithm. However, in literature, many recursive algorithms are said to work in-place (using ambiguous terminology) even if they do not, since they require a stack of logarithmic height. In this case we say that the algorithm is *internal* [5]. For instance Quick-Sort [11] is an internal algorithm, Merge-Sort adopts external sorting since it uses a linear amount of working space for merges, while Insertion-Sort is a real in-place algorithm.

The focus of many sorting algorithms lies instead on practical methods whose running time outperforms standard techniques even when comparisons are cheap. In this case expected (rather than worst case) performances is the main concern. Among this class of algorithms, despite its higher number of comparisons than for other methods, Quick-Sort and most of its variants are generally considered the fastest general purpose sorting methods.

The Insertion-Sort algorithm, particularly relevant for this paper, is also considered one of the best and most flexible sorting methods despite its quadratic worst case time complexity, mostly due to its stability, good performances, simplicity, in-place and online nature. It is a simple iterative sorting procedure that incrementally builds the final sorted array. Its pseudocode is depicted in Fig. 1. The i-th iteration of the algorithm takes at most $O(i)$ time, for a total $O(n^2)$ time complexity in both worst and average case.

Algorithm	t_{wc}	c_{wc}	*space*	t_{ac}	t_{bc}	year
Insertion-Sort	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n^2)$	$O(n)$	–
Rotated-IS [22]	$O(n^{1.5} \log n)$	–	$O(w)^\star$	$O(n^2)$	–	1979
Controlled-Density-IS [18]	$O(n^{1.5})$	–	$O(n)$	$O(n \log n)$	–	1980
Library-Sort [1]	$O(n^2)$	$O(n \log n)$	$O(n)$	$O(n \log n)^*$	$O(n)$	2006
Binary-IS [27]	$O(n^2)$	$O(n \log n)$	$O(1)$	$O(n^2)$	$O(n \log n)$	2008
2-Element-IS [19]	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n^2)$	$O(n)$	2010
Enhanced-IS [13]	$O(n^2)$	$O(n^2)$	$O(n)$	$O(n^2)$	$O(n)$	2013
Adaptive-IS [23]	$O(n^2)$	$O(n \log n)$	$O(1)$	$O(n^2)$	$O(n)$	2014
Doubly-IS [24]	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n^2)$	$O(n)$	2015
Bidirectional-IS [21]	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n^{1.5})$	$O(n^2)$	2017
Brownian-Motus-IS [9]	$O(n^2)$	$O(n^2)$	$O(1)$	$O(n^2)$	$O(n)$	2018
Clustered-Binary-IS [9]	$O(n^2)$	$O(n \log n)$	$O(1)$	$O(n^2)$	$O(n)$	2018
Block-Insertion-Sort	$O(n^{1.5})$	$O(n \log n)$	$O(1)^\flat$	$O(n \log n)^\S$	$O(n)$	2019
Fast-Insertion-Sort$^{(h)}$	$O(n^{1+\frac{1}{h}})$	$O(n \log n)$	$O(1)^\flat$	$O(n \log n)^\S$	$O(n)$	2019
Fast-Insertion-Sort	$O(n^{1+\varepsilon})$	$O(n \log n)$	$O(1)^\flat$	$O(n \log n)^\S$	$O(n)$	2019

Table 1. Milestones of sorting methods based on Insertion-Sort. Algorithms are listed in publication time order and are portrayed in terms of worst case (t_{wc}), average case (t_{ac}) and best case (t_{bc}) time complexity, worst case comparison number (c_{wc}) and space complexity. The last three algorithms are introduced in this paper. In ($\star$) w represents the number of bits in a computer word. In ($*$) the given complexity is observed with *high probability*. In (§) the $O(n \log n)$ behaviour has been observed by experimental evaluations. In ($\flat$) we refer to the in-place variants (see Section 2.1).

The Insertion-Sort algorithm is also *adaptive*, achieving $O(\delta n)$ worst case time complexity when each element in the input array is no more than δ places away from its sorted position. Moreover it has an enviable linear best case time (which occurs when the input array is already sorted) and can be immediately translated into programming code (see Bentley [2] for a simple 3-lines implementation of Insertion-Sort in the **C** programming language).

Much more interesting, Insertion-Sort is one of the fastest algorithms for sorting very small arrays, even faster than Quicksort [33]. Indeed, efficient Quick Sort implementations use Insertion-Sort as a subroutine for arrays smaller than a given threshold [28, 29, 35]. Such threshold should be experimentally determined and depends on the machine. However it is commonly around 10 elements.

Due to such many advantages the Insertion-Sort algorithm has inspired much work and is the progenitor of several algorithmic variants which aim at reducing comparisons and element moves very efficiently. Table 1 presents a list of most of effective variants of Insertion-Sort portrayed in terms of several features.

For the sake of completeness we also report the some related algorithms obtained as the combination of Merge-Sort and Insertion-Sort [6, 5, 34, 17] which, although not of practical significance, remain of theoretical interest.

In this paper we present a new family of efficient external algorithms inspired by Insertion-Sort, where each element of the family, named Fast-Insertion-Sort$^{(h)}$ for $h \in \mathbb{N}$, achieves an $O(n^{1+\frac{1}{h}})$ worst-case time complexity by nesting previous algorithms of the same family. Although Fast-Insertion-Sort$^{(h)}$ requires $O(n^{1-\frac{1}{h}})$ additional space we show how to translate it into an in-place solution at the cost of a constant factor. In addition we also devise a purely recursive generalized version of our Fast-Insertion-Sort$^{(h)}$ algorithms, which achieves an $O(n \log n)$ worst case time complexity. Like Insertion-Sort our solutions are stable, adaptive and very simple to translate into programming code.

Surprisingly, it turns out from our experimental results that our variants of the Insertion-Sort algorithm are very effective in practical cases showing an $O(n \log n)$ behaviour in practice. Their performances are comparable to that of the most effective sorting algorithms known in literature, overcoming fast implementations of the Hoare's Quick-Sort algorithm in many practical cases.

Throughout the paper we will use the following terminology. Given an array $A[0..n-1]$, of size n, we denote by $A[i]$ the $(i+1)$-st element of A, for $0 \leq i < n$, and with $A[i..j]$ the portion of the array contained between the $(i+1)$-st and the $(j+1)$-st characters of A, for $0 \leq i \leq j < m$. We indicate by $(A+i)$ the array beginning at $(i+1)$-st position of A, so that $(A+i)[0..j] = A[i..j]$.

2 The Fast-Insertion-Sort Algorithms

In this section we present Fast-Insertion-Sort, a new family of efficient sorting algorithms obtained as natural generalizations to multiple elements insertion of the standard Insertion-Sort algorithm. The underlying idea is to extend, at each iteration, the left portion of the array by means of the insertion of a sorted block of k elements, with $k \geq 2$. Despite its simplicity, this approach surprisingly leads, under suitable conditions, to a family of very efficient algorithms, both in theory and in practice, whose behaviour moves close to a linear trend for increasing values of the input size.

The new family is a sequence of nested algorithms, Fast-Insertion-Sort$^{(h)}$, for $h \in \mathbb{N}$, where the h-th algorithm can be applied when $n > 2^h$. The pseudocode of the Fast-Insertion-Sort$^{(h)}$ algorithm and its auxiliary procedure Insert-Block are depicted in Fig. 2.

Each algorithm in the sequence solves the problem by nesting function calls to previous algorithms in the same sequence and is associated with an integer parameter $h > 0$ which substantially represents the *depth* of such a nesting. For this reason in the execution of the Fast-Insertion-Sort$^{(h)}$ algorithm it is assumed that the size of the input array is at least 2^h.

The following definition of *input size degree* is particularly useful for the characterization of our algorithm.

Definition 1 (Input size degree). *Given an input array of size n and a constant parameter $c \geq 2$, we say that its input size degree (or simply its degree) is h, with reference to c, if $c^{h-1} < n \leq c^h$. We refer to the constant value c as the partitioning factor. When c is clear from the context we simply say that the input size degree is h. We can easily compute the degree of the input n by $h = \lceil \log_c(n) \rceil$.*

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

```
FAST-INSERTION-SORT$^{(h)}$($A, n$)
1. if $n \leq 2^{h-1}$ then $h \leftarrow \lceil \log_2(n) \rceil$
2. $k \leftarrow n^{(h-1)/h}$
3. $T \leftarrow$ array[$k$]
4. for $i \leftarrow 0$ to $n$ (step $k$) do
5.     $b \leftarrow \min(k, n - i)$
6.     FAST-INSERTION-SORT$^{(h-1)}$($A + i, b$)
7.     INSERT-BLOCK($A, i, b, T$)
```

Fig. 2. The pseudocode of Fast-Insertion-Sort$^{(h)}$, with a nested structure of depth $h > 0$, and its auxiliary procedure INSERT-BLOCK.

Note 1 (Block partitioning). If the the input array has degree h then the algorithm performs a partitioning of the array in at most c blocks of size $k = n^{1-\frac{1}{h}}$. In the worst case, indeed, $n = c^h$ and the number of blocks is given by

$$\left\lceil \frac{n}{k} \right\rceil = \left\lceil \frac{n}{n^{1-\frac{1}{h}}} \right\rceil = \left\lceil n^{\frac{1}{h}} \right\rceil = \left\lceil \left(c^h\right)^{\frac{1}{h}} \right\rceil = c$$

In addition observe that each block of size k represents an array of degree $h - 1$. Indeed, if $c^{h-1} < n \leq c^h$ then $k = n^{1-\frac{1}{h}}$ is bounded by

$$c^{h-2} < c^{h-2+\frac{1}{h}} < \left(c^{h-1}\right)^{1-\frac{1}{h}} < k \leq \left(c^h\right)^{1-\frac{1}{h}} = c^{h-1}$$

Observe that if $n < c^h$ then the last block has size greater than 0 and less than k, thus we have no guarantees that such size has degree equal to $h - 1$.

The algorithm is based on an iterative cycle on the variable i. At the beginning of each iteration, the left portion of the array, $A[0..i - 1]$, is assumed to be already sorted. Then the algorithm sorts the elements in the block $A[i..i + k - 1]$, of size k, by a nested call to Fast-Insertion-Sort$^{(h-1)}$, and tidily insert all the elements of the block in the sorted portion of the array by means of procedure INSERT-BLOCK. The overall number of iterations is then equal to $\lceil n/k \rceil$, where the size of the last block my be less than k and specifically it has size $n - k(\lceil n/k \rceil - 1)$.

A call to procedure INSERT-BLOCK(A, i, k, T) has the effect to insert the block $A[i..i + k - 1]$, of size k, into the left side of the array $A[0..i - 1]$, using the array T as a working area. It is assumed that both $A[0..i - 1]$ and $A[i..i + k - 1]$ have both been sorted. During the insertion process elements of the left portion will be moved over the elements of the right block in order to make room for the insertions. The additional array T, of size k, is used to temporarily store the elements of the right block so that they are not overwritten.

Note 2. Since procedure INSERT-BLOCK uses external memory for temporarily storing the elements of the block $A[i..i+k-1]$, all swap operations (SWAP(a, b)) inside the procedure (lines 2, 7 and 9), may be replaced by element moves ($a \leftarrow b$), cutting down the number of assignements of a factor up to 3.

The insertion process iterates over the elements of the right block (now temporarily moved on T) proceeding from right to left (while cycle of line 5). For each element $T[\ell]$, with $0 \leq \ell < k$, the algorithm iterates over the elements $A[j]$ of the left block, proceeding from right to left, in order to find the correct position where $T[\ell]$ must be placed (while cycle of line 6). If the element $A[j]$ is greater than $T[\ell]$ then it is moved of ℓ position to the right (line 7). Once the first element $A[j]$, such that $A[j] \leq T[\ell]$, is found (or $j < 0$), $T[\ell]$ is moved at position $A[j + \ell + 1]$ and ℓ is decreased (lines 9-10).

Note 3. For simplicity procedure INSERT-BLOCK shown in Fig. 2 uses a linear search for locating the correct position where an element must be placed. It takes at most $O(i)$ comparisons, if i is the size of the sorted portion of the array. However it is more favourable to use a binary search [27] which would allow to reduce the number of comparisons to $O(\log i)$.

It is easy to prove that all Fast-Insertion-Sort$^{(h)}$ algorithms maintain all the features of the standard Insertion-Sort algorithm and specifically they guarantee a stable sorting and are very simple to implement. However a drawback of this family of algorithms is that they use additional external memory of size $O(k)$. In the Section 2.1 we show how to obtain in-place variants of such algorithms.

Observe that Fast-Insertion-Sort$^{(1)}$ corresponds to the standard Insertion-Sort algorithm where $k = n^0 = 1$ and each block consists in a single element.

The algorithm Fast-Insertion-Sort$^{(2)}$ devises a special mention since it can be easily translated into a purely iterative in-place online adaptive sorting algorithm. Due to its distinctiveness we give it a special name: Block-Insertion-Sort. It can be proved that if we set $k = \sqrt{n}$ then the worst-case time complexity of Block-Insertion-Sort is $O(n^{1.5})$. Although there are few variants [22, 18] achieving $o(n^2)$ worst case time complexity using additional space, to the best of our knowledge this is the first time an iterative in-place online variant of the Insertion-Sort algorithm achieves a $o(n^2)$ complexity in the worst case.

The sequence of algorithms Fast-Insertion-Sort$^{(h)}$, for $h \in \mathbb{N}$, converge to a purely recursive algorithm which could be seen as the representative of the family and which, for this reason, we name simply Fast-Insertion-Sort. The pseudocode of the Fast-Insertion-Sort algorithm is depicted in Fig. 4. It maintains the main structure of the Fast-Insertion-Sort$^{(h)}$ algorithms but dynamically computes the value of the input size degree h. Specifically, given a partitioning parameter $c \geq 2$, the algorithm computes the input size degree as $h = \lceil log_c(n) \rceil$. The choice of the partitioning parameter c could be critical for the practical performances of the algorithm[3]. Note that, for $c = 2$, Fast-Insertion-Sort exactly behaves like

[3] As discussed in Section 3 the best choice for the partitioning parameter, leading to the best performances in practical cases, is to set $c = 5$.

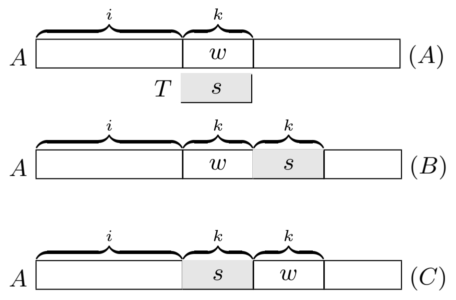


Fig. 3. Three schemas for the insertion of a sorted block of size k into $A[0..i-1]$. Two blocks are involved: the *working area* w which represents the block to be inserted, and a *storage area* s, which is used to temporarily store elements of the working area. In schema (A) an external storage area T is used. In schema (B) the contiguous block $A[i+k..i+2k-1]$ is used as storage area. In schema (C) we insert the block $A[i+k..i+2k-1]$ and use the block $A[i..i+k-1]$ as storage area.

Merge-Sort, since the process of inserting a sorting block into another one can be viewed as a merging procedure. It can be proved that the Fast-Insertion-Sort algorithm has a $O(n \log n)$ worst case time complexity. Moreover such algorithm shows also very good performances in practical cases, as discussed in Section 3.

2.1 In-Place Sorting

In this section we briefly describe how to allow Fast-Insertion-Sort algorithms to use an internal storage area to temporarily save the elements of the working area representing the block which should be inserted during the current iteration, making it an in-place sorting algorithm[4].

In order to make sure that the contents of the storage area is not lost, it is sufficient to guarantee that whenever an element should be moved from the working area, it is swapped with the element occupying that respective position in the storage area. Procedure INSERT-BLOCK, shown in Fig. 2, makes this guarantees.

In this context it should be possible to use the contiguous block $A[i+k..i+2k-1]$, of size k, as storage area, as depicted in Schema B of Fig. 3. This could be obtained by replacing the procedure call INSERT-BLOCK(A, i, k, T) (line 6) with INSERT-BLOCK$(A, i, k, A+k)$. The main advantage of this approach is that the part of the array that is not currently being sorted can be used as temporary storage area. This yields a fast variant for the Fast-Insertion-Sort$^{(h)}$ algorithm which works in-place requiring only constant extra space.

However, since the insertion of a block of size k requires a contiguous block of k elements, at the last iteration, when $i > n-2k$, there is not enough space to allow the algorithm to work in-place. Although many solutions may be adopted to address such case, we will prove it is enough to insert the elements of the last

[4] This idea is not original since it has been successfully used in literature. The first sorting algorithm which introduced such a technique was the Quick-Heap-Sort algorithm by Cantone and Cincotti [3]. Recently it has been also adopted to obtain fast internal variants of many recursive external sorting algorithms [5]

```
FAST-INSERTION-SORT($A, n$)
1. if $n \leq 2$ then
2.     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.     $b \leftarrow \min(k, n-i)$
8.     FAST-INSERTION-SORT($A+i, b$)
9.     INSERT-BLOCK($A, i, b, T$)
```

Fig. 4. The pseudocode of the Fast-Insertion-Sort algorithm, the purely recursive sorting algorithm, where the input size degree h is automatically selected as $h = \lceil \log_c(n) \rceil$, for a given partitioning parameter $c \geq 2$.

block, whose size is at most $2k-1$, by individual insertions, as done by standard Insertion-Sort.

Thus at the beginning of each iteration the two blocks $A[i..i+k-1]$ and $A[i+k..i+2k-1]$ will be swapped. Subsequently, as in the previous case, the algorithm sorts the block $A[i..i+k-1]$, of size k, and insert it in the sorted portion of the array, using the block $A[i+k..i+2k-1]$, as storage area.

However, since all elements of the array must be inserted, sooner or later, it is possible to avoid the k swap operation necessary to move the elements in the storage area (lines 1-2 of procedure INSERT-BLOCK) by interchanging the two blocks involved in the insertion process. Specifically we can directly insert the block $A[i+k..i+2k-1]$ using the block placed in the middle, i.e. $A[i..i+k-1]$, as storage area, as depicted in Schema C of Fig. 3.

It is easy to prove that the resulting in-place variants of the Fast-Insertion-Sort$^{(h)}$ algorithms maintain most of the features listed above: they work online and are very simple to implement. Unfortunately, due to the swap operations performed on the storage area, they don't guarantee a stable sorting and their performance are a bit worse than the corresponding external variants.

3 Experimental Evaluation

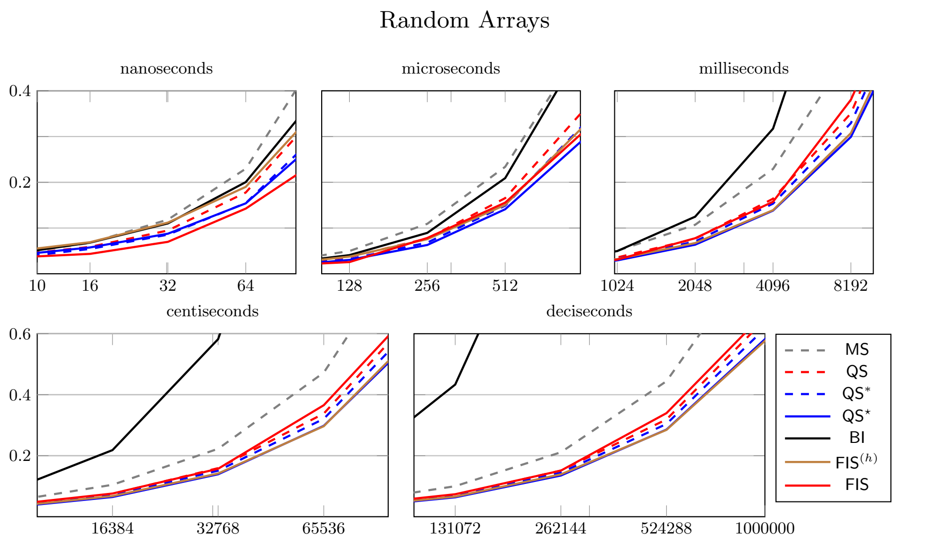
In this section we present and comment the experimental results relative to a comparison of our newly presented algorithms against some of the most effective sorting algorithms known in literature. Specifically we tested the following algorithms:

- Merge-Sort (MS), the external algorithm with a $\Theta(n \log n)$ time and $O(n)$ space complexity. Here we implemented a version using only an array of size $\lceil n/2 \rceil$ as storage space.
- Heap-Sort (HS), an internal algorithm with a $\Theta(n \log n)$ time complexity.

– Quick-Sort (QS), an internal algorithm with a $O(n^2)$ time complexity and $O(n \log n)$ expected time. Here we implemented the standard Hoare's algorithm with a random selection of the pivot element.
– Quick-Sort* (QS*), an optimized version of the Hoare's algorithm [11] which uses Insertion-Sort for input size less or equal to 10.
– Block-Insertion-Sort (BI), the in-place variant of algorithm described in Section 2.
– Fast-Insertion-Sort$^{(h)}$ (FIS$^{(h)}$), the external variants of the nested algorithms described in Section 2, for $2 \leq h \leq 10$.
– Fast-Insertion-Sort (FIS), the external variant of the recursive algorithm described in Section 2, for $2 \leq c \leq 10$.
– Quick-Sort* (QS*), an optimized version of the Hoare's algorithm which uses Block-Insertion-Sort for input sizes less or equal to 512.

All algorithms have been implemented in the C++ programming language and have been compiled with the `GNU C++ Compiler 8.3.0`, using the optimization options `-O2 -fno-guess-branch-probability`. The codes of tested algorithms are available at `https://github.com/ostafen/Fast-Insertion-Sort`. All tests have been performed on a PC with a 3.40 GHz Intel Quad Core i5-4670 processor, with 6144 KB cache memory, and running Linux Ubuntu 19.04. Running times have been measured with a hardware cycle counter, available on modern CPUs. We tested our algorithms on input arrays with a size $n = 2^i$, with $2 \leq i \leq 20$. For each value of n we reported the mean over the running times of 1000 runs.

For each input size n, tests have been performed on arrays of integers and specifically on random arrays, partially ordered arrays, and reverse order arrays. Random input sequences have been generated with a uniform distribution in the interval $[0 \ldots 2^{31}]$, using the C++ random number generator. Partially ordered arrays have been obtained by sorting random generated arrays and subsequently executing a number of swaps equal to 1/4 of the input size. Fig. 5 summarizes the overall behaviour of the algorithms for the discussed generation strategies. It turns out that the newly presented algorithms show in all cases a $O(n \log n)$ behaviour in practice, outperforming Merge-Sort and Heap-Sort (which are asymptotically optimal algorithms). Our new solutions are also very competitive against the Quick-Sort algorithm, even for large sizes of the input arrays. It is worth to notice that Fast-Insertion-Sort$^{(h)}$ algorithms show the lowest running times for $2 \leq h \leq 4$, while Fast-Insertion-Sort achieves the best performance for c = 5. For small input size arrays, Block-Insertion-Sort obtains good results, and this is why QS* turns out to be the Quick-Sort-based algorithm with the best performances, outperforming the Quick-Sort implementation which uses Insertion-Sort as a subroutine. The recursive algorithm Fast-Insertion-Sort turns out to be faster than standard Quick-Sort in almost all cases, and the gap turns out to be more sensible for very small and for very large input arrays. In addition, while reverse order arrays represents the Quick-Sort worst case scenario, algortithms in the Fast-Insertion-Sort family mantain their optimal execution time once again, confirming to be among the most fast and flexible general purpose sorting alghoritms.



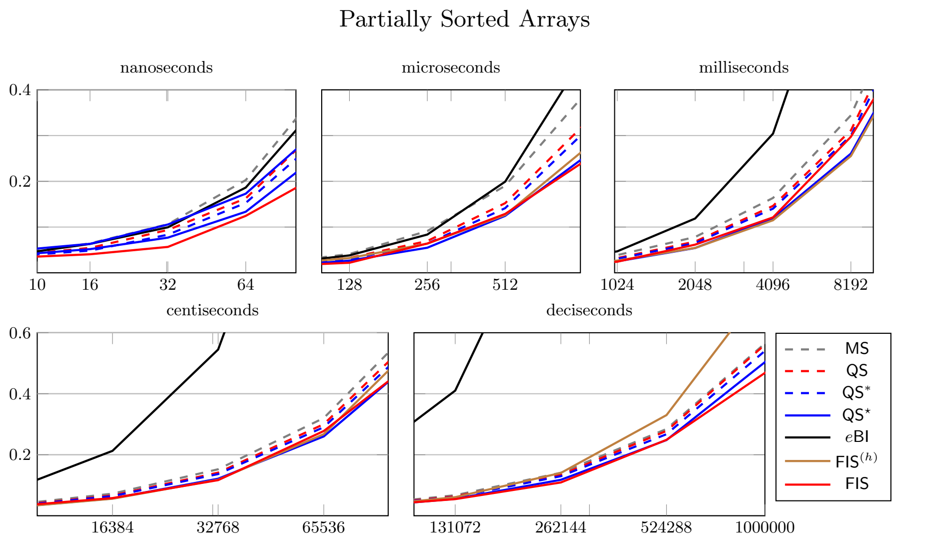


Fig. 5. Running times obtained on partially sorted arrays of size 2^k, with $3 \leq k \leq 20$. Reported times are the means of 1.000 runs. Plots are divided in classes of sizes, and specifically: sizes 10^1-10^2 (expressed in nanoseconds), 10^2-10^3 (expressed in microseconds), 10^3-10^4 (expressed in milliseconds), 10^4-10^5 (expressed in centiseconds) and 10^5-10^6 (expressed in deciseconds). Previous algorithms are depicted in dashed lines, while new algorithms are depicted in solid lines.

4 Conclusions

In this paper we presented a new family of efficient, flexible, stable, simple sorting algorithms, named Fast-Insertion-Sort. The algorithms of such a new family generalize the Insertion-Sort algorithm to multiple elements block insertion and achieve an $O(n^{1+\varepsilon})$ worst-case time complexity, where $\varepsilon = \frac{1}{h}$, for $h \in \mathbb{N}$. Moreover we generalized the basic idea obtaining a recursive algorithm achieving $O(n \log n)$ worst case time complexity. We also discussed how to obtain in-place variations of such algorithms by maintaining their main features. From our experimental results it turns out that our solutions are very competitive with the most effective sorting algorithms, outperforming fast implementations of the Hoare's Quick-Sort algorithm in many practical cases.

References

1. M. Bender, M. Farach-Colton, and Mosteiro, *M. Theory Comput Syst*, 39: 391 (2006). https://doi.org/10.1007/s00224-005-1237-z
2. J. Bentley, Programming Pearls, *ACM Press/Addison-Wesley*, pp. 116, 121 (2000)
3. D. Cantone and G. Cincotti, QuickHeapsort: an efficient mix of classical sorting algorithms. *Theoretical Computer Science*, 285(1) pp. 25-42 (2002) DOI: 10.1016/S0304-3975(01)00288-2
4. P. S. Dutta, An approach to improve the performance of insertion sort algorithm, *International Journal of Computer Science and Engineering Technology*, 4, 503-505 (2013).
5. S. Edelkamp, A. WeiB, S. Wild, QuickXsort - A Fast Sorting Scheme in Theory and Practice. *CoRR abs*/1811.01259 (2018)
6. L.R. Ford and S.M.Jr. Johnson, A tournament problem, *American Mathematical Monthly*, 66: 387-389 (1959), doi:10.2307/2308750
7. R. M. Frank and R. B. Lazarus. A High-Speed Sorting Procedure. *Communications of the ACM.* 3 (1): 20-22 (1960) doi:10.1145/366947.366957.
8. V. Geffert, J. Katajainen, T. Pasanen. Asymptotically efficient in-place merging. *Theoretical Computer Science.* 237: 159-181 (2000). doi:10.1016/S0304-3975(98)00162-5.
9. S. Goel and R. Kumar. Brownian Motus and Clustered Binary Insertion Sort methods: An efficient progress over traditional methods. *Future Generation Computer Systems*, vol.86, pp. 266-280 (2018)
10. T. N. Hibbard, An Empirical Study of Minimal Storage Sorting. *Communications of the ACM.* 6 (5): 206-213 (1963) doi:10.1145/366552.366557.
11. C. A. R. Hoare. Algorithm 65: Find. Commun. ACM, 4(7):321-322, July 1961. URL: `http://doi.acm.org/10.1145/366622.366647`, doi:10.1145/366622.366647.
12. B. Huang and M.A. Langston, Practical In-Place Merging. *Communications of the ACM.* 31 (3): 348-352 (1988). doi:10.1145/42392.42403.
13. M. Khairullah, Enhancing worst sorting algorithms, *International Journal of Advanced Science and Technology*, 56, 13-26 (2013)
14. F. Lam, R. K. Wong, Rotated library sort, *Proceedings of the 19th Computing: The Australasian Theory Symposium*, Volume 141, Australian Computer Society, Inc., pp. 21-26, 2013
15. J. Katajainen, T. Pasanen and J. Teuhola, Practical in-place mergesort. *Nordic J. Computing.* 3 (1): 27-40 (1996)

16. P. Kim and A. Kutzner. Stable Minimum Storage Merging by Symmetric Comparisons. *European Symp. Algorithms.* Lecture Notes in Computer Science 3221. pp. 714-723 (2004). doi:10.1007/978-3-540-30140-0_63. ISBN 978-3-540-23025-0
17. A. Kutzner, P. Kim, Ratio Based Stable In-Place Merging. *Lecture Notes in Computer Science*, 4978. Springer Berlin Heidelberg. pp. 246-257 (2008)
18. R. Melville and D. Gries. Controlled density sorting. In *Information Processing Letters*,10:4,pages169-172,1980.
19. W. Min, Analysis on 2-element insertion sort algorithm. *Proceedings of International Conference on Computer Design and Applications* (ICCDA), Vol. 1, IEEE, pp. 143-146 (2010). doi:10.1109/ICCDA.2010.5541165.
20. A. D. Mishra, D. Garg, Selection of best sorting algorithm, *International Journal of Intelligent Information Processing*, 435 2 (2) (2008) 363-368.
21. A. S. Mohammed, S. E. Amrahov, F. V. Celebi, Bidirectional conditional insertion sort algorithm; An efficient progress on the classical insertion sort, *Future Generation Computer Systems*, 71, pp.102-112 (2017). doi:10.1016/j.future.2017.01.034.
22. J. I. Munro and H. Suwanda, Implicit data structures, in STOC 79, *Proceedings of the eleventh annual ACM symposium on Theory of computing*, ACM Press, pp.108-117 (1979)
23. K. Nenwani, V. Mane, S. Bharne, Enhancing adaptability of insertion sort through 2-way expansion, *Proceedings of 5th International Conference on Confluence The Next Generation Information Technology Summit (Confluence)*, IEEE, pp. 843-847 (2014). doi:10.1109/CONFLUENCE.2014.6949294.
24. S. Paira, A. Agarwal, S. S. Alam, S. Chandra, Doubly inserted sort: A partially insertion based dual scanned sorting algorithm, *Emerging Research in Computing, Information, Communication and Applications*, Springer, pp.11-19 (2015). doi:10.1007/978-81-322-2550-8_2.
25. A. A. Papernov and G. V. Stasevich. A Method of Information Sorting in Computer Memories. *Problems of Information Transmission.* 1 (3): 63-75 (1965).
26. V. R.Pratt. Shellsort and Sorting Networks (Outstanding Dissertations in the Computer Sciences). *Garland* (1979). ISBN 978-0-8240-4406-0.
27. B. R. Preiss, Data Structures and Algorithms with Object-oriented Design Patterns in C++, *John Wiley & Sons*, 2008.
28. Robert Sedgewick, The analysis of Quicksort programs, Acta Inform. (ISSN: 0001-5903) 7 (4) (1977) 327-355. Available on http://dx.doi.org/10. 1007/BF00289467.
29. Robert Sedgewick, Implementing Quicksort programs, *Commun. ACM* (ISSN: 00010782) 21 (10) (1978) 847-857. http://dx.doi.org/10.1145/359619.359631.
30. R. Sedgewick. A New Upper Bound for Shellsort. *Journal of Algorithms.* 7 (2): 159-173 (1986). doi:10.1016/0196-6774(86)90001-5.
31. D. L. Shell, A High-Speed Sorting Procedure. *Communications of the ACM.* 2 (7): 30-32 (1959) doi:10.1145/368370.368387.
32. T. SinghSodhi, S. Kaur and S. Kaur, Enhanced insertion sort algorithm, *Int. J. Comput. Appl.* 64 (21) (2013) 35-39.
33. 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.
34. G. van den Hoven. Binary Merge Sort. https://docs.google.com/file/d/0B8KIVX-AaaGiYzcta0pFUXJnNG8/edit
35. S. Wild, M. E. Nebel, R. Neininger, Average case and distributional analysis of dual-pivot quicksort, *ACM Trans. Algorithms*, 11 (3) 22 (2015)