MaxSort: An Efficient and Complete Sorting Algorithm

August 2023

Abstract

Despite order statistics and binary search playing increasingly crucial roles in real life, an efficient and complete sorting algorithm capable of sorting any numerical data remains lacking. To address these issues, we propose a sorting algorithm called MaxSort. In this algorithm, we iteratively retrieve the maximum, second maximum, third maximum, and so on to derive the descending sorted array. To the best of our knowledge, our algorithm is more efficient than any existing work and can correctly handle any numerical data. We conducted comprehensive experiment to demonstrate the superiority of our algorithm.

1 Introduction

Sorting is gaining more and more attention nowadays because it plays an essentially important role in providing order statistics [5]. For example, an efficient sorting algorithm can facilitate schools or other organizations to admit new members. Additionally, it can also help analyze data by providing median or the k-th smallest element. Moreover, an efficient sorting algorithm is key for efficient searching. For instance, the efficiency of the sorting algorithm can greatly impact the binary search [4], which requires first sorting the elements.

The task of sorting is, given an array of numerical elements, to sort the array such that the output array contains the same set of integers but in the ascending/descending order. For example, if the input array is $\{3, 1, 2, 1, 5, 4\}$, the descending output is $\{5, 4, 3, 2, 1, 1\}$. As deriving the ascending and descending sorted array are essentially the same, we only discuss the algorithm to derive the descending output for simplicity.

Since the sorting algorithm is the fundamental algorithm in various aspects, there have been quite a few attempts to develop a sorting algorithm. However, [2] can only handle arrays without duplicate elements, which is not necessarily true in real life. Additionally, though [3] can successfully sort the target array (including arrays with duplicate elements), it has an exponential computational complexity, which is extremely expensive and impractical.

To develop an efficient sorting algorithm that can handle duplicate elements, we propose the MaxSort algorithm. In this algorithm, we iteratively extract the maximum, the second maximum, the third maximum, and so on, to sort the arrays. Consider the existence of duplicate elements, we further make a sophisticated revision to the definition of k-th maximum, which will be discussed in detail in Section 4. Following this principle, again in the aforementioned example, for the array $\{3, 1, 2, 1, 5, 4\}$, we first extract 5, then extract 4, and then $3, 2, 1, 1$ respectively. Finally, we can derive the sorted output $\{5, 4, 3, 2, 1, 1\}$.

We summarize our contributions as follows.

- To the best of our knowledge, we are the first to completely define the sorting problem formally.
- To the best of our knowledge, we propose the most efficient sorting algorithm, even compared to [2].
- MaxSort is flexible with any kinds of numerical arrays, including arrays with duplicate elements.
- We conducted comprehensive experiments on both synthetic and real datasets to verify the correctness and efficiency of our sorting algorithm. In terms of the efficiency and the correct rate, our algorithm outperforms all existing relevant algorithms.

The rest of this paper is organized as follows. Section 2 reviews some related work trying to solve the sorting problem. Section 3 formally defines the complete sorting problem. Section 4 proposes our MaxSort algorithm. Finally, Section 5 presents extensive experiment demonstrating the superiority of our algorithm, and Section 6 discusses our conclusion.

2 Related Work

Though sorting is crucial in real life, there are still a very limited number of meaningful works in this field. In this section, we will review two most related work in sorting.

A. An Incomplete Sorting Algorithm

[2] attempted to sort an integer array using the Fake-Sorting algorithm (*imaginary*) but made an assumption that there are no duplicate elements. If there are any two duplicate elements, this algorithm may return the wrong position indices for these elements, resulting in sorting failure. Furthermore, as [2] did not consider other numerical element types such as floating points and it did not allow duplicate elements, the sorting problem definition in this paper is incomplete. In Section 3, we will formally define a complete sorting problem.

B. An Expensive Sorting Algorithm

Sorting arrays including duplicate elements is common in real life. As a result, [3] tried to sort such arrays. However, the Expensive-Sorting algorithm (*imaginary*) proposed in this work requires an exponential computational complexity, which is impractical to use in real life. Additionally, this work did not formally define the sorting problem.

3 Problem Definition

Let $A = \{a_1, a_2, \cdots, a_N\}$ denote an array of N numerical elements, where a_i ($i \in \{1, 2, \cdots, N\}$) can be any real number (e.g., a_i can be 2 or -0.7). Besides, denote $A[i]$ as the i-th element of A, and $A[i, \cdots, j] = \{a_i, a_{i+1}, \cdots, a_j\}$.

The goal of sorting is to rearrange the elements of A into a new array O such that the elements in O are in the ascending or descending order. Without loss of generality (W.L.O.G.), we will only focus on sorting the elements into descending order.

Formally, given $A = \{a_1, a_2, \cdots, a_N\}$, the goal is to produce an output array $O = \{a_{i_1}, a_{i_2}, \cdots, a_{i_N}\}$, where $a_{i_p} \geq a_{i_q}, \forall p < q$ and $\{i_1, \cdots, i_N\}$ is a permutation of $\{1, 2, \cdots, N\}$.

For example, given $A = \{3, 1, 2, 1, 5, 4\}$, $N = 6$, a valid sorted output would be $O = \{5, 4, 3, 2, 1, 1\}$.

4 Methodology

Inspired by the definition of order statistics, an order sorted (descending) array is actually an array whose elements are from the largest to the smallest. For simplicity, if we do not consider duplicate elements at this moment, it means the k-th element in the sorted array is exactly the k-th largest element in the original array. Hence, to conduct sorting, we only need to iteratively extract the k-th maximum of the elements and place it in the k-th position in the output array. Interestingly, after a minor revision of the definition of k-th maximum, the aforementioned procedure can be directly applied to any general cases which allow duplicate elements. More details are discussed in the following sections.

4.1 Algorithm

Algorithm 1: MaxSort Algorithm

Input : A numerical array $A \leftarrow \{a_1, a_2, \cdots, a_N\}$
Output: A sorted array $O \leftarrow \{a_{i_1}, a_{i_2}, \cdots, a_{i_N}\}$

```
1 for k ← 1 to N do
2     for j ← k to N do
3         if A[j] > A[k] then
4             Interchange A[k] and A[j]
5 Output: Output the rearranged A as O
```

At first, we re-define k-th maximum here, called generalized k-th maximum.

Definition 1. ***Generalized K-th Maximum:*** *Given a set of numerical elements, we define the generalized k-th maximum, denoted as M_k, to be no larger than all previous k-1 maxima. Formally, $M_k \leq M_i, \forall i \in \{1, 2, \cdots, k-1\}$.*

After we generalize the definition of k-th maximum, then what Algorithm 1 does is exactly placing M_k into the k-th position in the output array. To be more detailed, the outer for loop is trying to localize the unsorted sub-array (i.e., $A[k, \cdots, N]$) and the inner for loop is trying to move the largest element in the aforementioned sub-array (i.e., M_k for A) to the k-th position. For example, for the array $A = \{3, 1, 2, 1, 5, 4\}$, after the first iteration of the outer for loop, A becomes $\{5, 1, 2, 1, 3, 4\}$. After the second iteration, the array becomes $\{5, 4, 1, 1, 2, 3\}$. As we can see, the first outer iteration move the 1-st largest element (i.e., M_1) to the first position and the second iteration move M_2 to the second position. By repeating this procedure, we can finally derive the sorted array.

4.2 Algorithm Analysis

We now analyze the correctness and time/space complexity of MaxSort.

4.2.1 Correctness of MaxSort

To prove the correctness of MaxSort, we only need to prove the correctness of the following three lemmas.

Lemma 1. *W.L.O.G, assume the outer for loop is in the k-th iteration ($k \in \{1, \cdots, N\}$), the inner for loop in Algorithm 1 is moving the generalized 1-st largest elements in $A[k, \cdots, N]$ into A[k].*

Proof. Assume the Lemma 1 is wrong, which equivalently means that, given a fix k, after the execution of the inner for loop, there exists some $p \in \{k+1, \cdots, N\}$ s.t. $A[p] > A[k]$. However, in line 4 in Algorithm 1, if $A[p] > A[k]$, an interchange between these two values will occur, causing contradiction. Hence, Lemma 1 is correct.

□

Lemma 2. *Each iteration, say the k-th iteration, in the outer for loop in Algorithm 1 is placing M_k of A into A[k].*

Proof. Again, assume the Lemma 2 is incorrect, which equivalently means that after the k-th iteration of the outer for loop, $A[k]$ is possibly not M_k. This assumption is two-fold. 1) There exists some $i \in \{1, 2, \cdots, k-1\}$ s.t. $A[i] < A[k]$. 2) There exists some $j \in \{k+1, \cdots, N\}$ s.t. $A[j] > A[k]$.

For 1), if we move back to the i-th iteration of the outer for loop, by Lemma 1, A[i] is the generalized 1-st maximum of $A[i, \cdots, N]$. Since $i \leq k \leq N$, then $A[i] \geq A[k]$, causing contradiction.

For 2), again, consider the k-th iteration of the outer for loop, by Lemma 1, $A[k]$ is the generalized 1-st maximum of $A[k, \cdots, N]$. Since $k \leq j \leq N$, then $A[k] \geq A[j]$, causing contradiction.

Hence, Lemma 2 is correct. □

Lemma 3. *If an array A satisfies $A[k] = M_k$, where M_k means the generalized k-th maximum of A, then A is sorted in descending order.*

Proof. Assume Lemma 3 is wrong, then there exists $1 \leq i < j \leq N$ s.t., $A[i] < A[j]$. However, by the definition of the generalized k-th maximum, $A[i] = M_i, A[j] = M_j, i < j$, then $A[i] \geq A[j]$, causing contradiction. Hence, Lemma 3 is correct. □

By Lemma 2 and Lemma 3, MaxSort algorithm correctly sorts any array.

4.2.2 Computational Complexity

Our MaxSort algorithm has computational complexity $O(N^2)$. To be more detailed, there are totally $\sum_{k=1}^{N} \sum_{j=k}^{N} = O(N^2)$ iterations and there are $O(1)$ operations in each inner iteration. Hence, our algorithm has $O(N^2)$ complexity, which is much more efficient than [3].

Besides, the space complexity of our MaxSort algorithm is $O(N)$ since we do not require any extra space except for $O(1)$ space for interchange.

5 Experiment

5.1 Setup

We conducted experiemnts on a machine with 3.10GHz CPU and 16GB of RAM. All programs were implemented in C++ on the MacOS system.

Statistics	Synthetic	USTA	HKS
No. of Arrays	$10,000$	33	33
Average Length	$50,000$	$78,000$	$7,000,000$

Table 1: Statistics of datasets used in the experiments.

5.1.1 Dataset

We conducted experiments on both synthetic and real datasets:

- **Synthetic Dataset**. We generated $10,000$ arrays with length $50,000$ using the random library [1]in Python. In each array, each element a_i has a random numerical value in $[-10000, 10000]$.
- **Real Dataset**. We used two real datasets for experiments, the *HKUST DSE Admission* dataset (USTA) and the *HK Individual Salary* dataset (HKS). We used the data records from year $1991 - 2023$.

Some statistics of the datasets are shown in Table 1.

5.1.2 Algorithm

Since there is no efficient and complete sorting algorithm, we compared our algorithm to the two most relevant algorithms discussed in Section 2, Fake-Sorting (FS) algorithm and Expensive-Sorting (ES) algorithm.

5.1.3 Parameter Setting

We evaluated the performance of algorithms by varying the following parameters. 1) Length of the arrays to sort. For all the datasets including the synthetic and the real datasets, they all have arrays with length greater than $50,000$ ($50k$). As a result, for each array, we select the top $30k$, $40k$, and $50k$ elements as sub-arrays for performance comparison.

5.1.4 Performance Measurement

Considering the FS algorithm may return wrong indices for duplicate elements and the ES algorithm has an exponential time complexity, we evaluated the algorithms by the following measurements. 1) *Correct Rate*, which is the percentage of correctly sorted arrays. 2) *Average Processing Time*, which is the average processing time to sort each array.

5.2 Results

Since sorting on the synthetic dataset and real datasets does not make a significant difference, we do not split them and discuss the experimental results

Correct Rate	MaxSort	FS	ES
Synthetic-30k	100%	93%	100%
USTA-30k	100%	94%	100%
HKS-30k	100%	93%	100%
Synthetic-40k	100%	92%	100%
USTA-40k	100%	92%	100%
HKS-40k	100%	92%	100%
Synthetic-50k	100%	91%	100%
USTA-50k	100%	90%	100%
HKS-50k	100%	88%	100%

Table 2: The correct rate of different algorithms.

Avg. Time	MaxSort	FS	ES
Synthetic-30k	2.73	3.84	23.66
USTA-30k	2.66	4.12	23.47
HKS-30k	2.68	3.79	25.09
Synthetic-40k	3.19	4.77	30.89
USTA-40k	3.07	5.15	32.89
HKS-40k	2.99	4.89	31.74
Synthetic-50k	3.78	6.27	60.98
USTA-50k	3.74	6.54	68.28
HKS-50k	3.69	6.18	65.46

Table 3: The average processing time of different algorithms (in seconds).

on both synthetic and real datasets together. Following Section 5.1.3, we conducted experiments on arrays with length $30k, 40k$, and $50k$ and the results are shown as follows.

5.2.1 Correct Rate

As shown in Table 2, both MaxSort and ES algorithms achieved 100% correct rate while the FS algorithm suffered from the duplicate elements. As the array length increased, the correct rate of the FS algorithm decreased because there were potentially more duplicate elements for longer arrays.

5.2.2 Averaged Processing Time

In Table 3, the average processing time of our MaxSort algorithm was significantly less than the other two algorithms. What is more, the ES algorithm which can correctly sort the arrays suffered from an extremely high computational time, making it impractical for real-life use.

5.3 Summary

The experiments demonstrated our algorithm's superiority over the best-known existing ones. 1) Our algorithm is complete, which can apply to any numerical arrays with duplicate elements. Since there is no fault tolerance in sorting, the FS algorithm is meaningless since it may produce incorrect outputs. 2) Our algorithm is efficient. Though the ES algorithm is also a complete algorithm, its high computational complexity makes it unpractical to use in real life. To sum up, our algorithm made significant improvements in both the completeness and efficiency for sorting algorithms.

6 Conclusion

In this paper, we first give a complete definition of sorting. What is more, we also present the MaxSort algorithm for completely and efficiently solving the sorting problem. Extensive experiments showed that our algorithm outperforms all best-known existing algorithms, particularly in terms of correct rate and average processing time. As for future work, we believe that it is important to further improve the efficiency of the sorting algorithms (e.g., reduce the complexity into $O(logN)$ or even linear $O(N)$ time). By further improving the efficiency, we can exploit better for applications replying on sorted data.

References

[1] https://github.com/python/cpython/blob/3.11/Doc/library/random.rst.

[2] Sorting algorithm 1. *Sorting Journal*, 2023.

[3] Sorting algorithm 2. *Sorting Journal*, 2023.

[4] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[5] Herbert A David and Haikady N Nagaraja. *Order statistics*. John Wiley & Sons, 2004.