MaxSort: A Method for Data Sorting through Interactively Moving the Largest Element

Weile TAN[1]

[1]The Hong Kong University of Science and Technology

[1]wtanae@connect.ust.hk

Abstract

In many practical applications, the need to sort data in a specific order is essential for decision-making processes. This paper presents a comprehensive formulation of the sorting problem and propose a novel algorithm called MaxSort to solve the problem. The MaxSort algorithm iteratively selects the largest element from the unsorted list and appends it to the sorted list. Extensive experiments are conducted to test the MaxSort algorithm, and the results indicate that the proposed algorithm performs effectively and efficiently on two sets of synthetic datasets.

Keywords: sorting problem, sorting algorithm.

1. Introduction

In many real-life applications, data in a certain order is required. For instance, when procuring raw materials in a factory setting, it is essential to rank the quotes from various suppliers in order to select the most cost-effective option, thereby minimizing expenses. For another example, in the job application process, companies need to rank the applicants by their qualifications to hire the most suitable candidates. However, in many cases, the raw data is not presented in the order that satisfies the requirements. As a result, an effective method to sort data into a required order is highly desirable.

The problem of sorting data into an order can be defined as follows. Given a list of data $L[0:n-1]$ with length n and an order relation $\leq$ (e.g. numerically smaller or equal), the sorted output is a list of data $L^{'}[0:n]$ that satisfied two requirements. Firstly, the output is in monotonic order, i.e., each element in the output list is no smaller than the previous element. In formal language, $\forall i \in \{1, 2, \ldots, n-1\}$, $L^{'}[i-1] \leq L^{'}[i]$. Secondly, the output list is a permutation of the input list, i.e., the sets of elements in L and $L^{'}$ are identical. Only the order of the elements in L and $L^{'}$ can be different. For instance, given a list of numbers $L = [0, 4, 3, 5, 2, 1, 2]$ and the numerical order relation $\leq$, the sorted output is

$L^{'} = [0, 1, 2, 2, 3, 4, 5]$, since $L^{'}$ is in monotonic order ($0 \leq 1 \leq 2 \leq 2 \leq 3 \leq 4 \leq 5$) and $L^{'}$ is a permutation of L. To the best of our knowledge, we are the first to discuss effective methods of solving the sorting problem.

To tackle the lack of approaches for solving the sorting problem, we propose an algorithm called MaxSort. The main idea of the MaxSort algorithm is interactively searching for the greatest element. Initially, the sorted list is set as an empty list, and the unsorted list is set to be identical to the given list. The MaxSort algorithm proceeds by iteratively selecting the largest element in the unsorted list, removing it from the unsorted list, and appending it to the sorted list from the left. If the length of the given list of data is n, then after n iterations, the unsorted list is empty, and the sorted list is the desired output of the algorithm. For instance, given a list of numbers $L = [0, 4, 3, 5, 2, 1, 2]$ and the numerical order, the unsorted list is initialized as $[0, 4, 3, 5, 2, 1, 2]$, and the sorted list is initially empty. Elements in the unsorted list are moved to the leftmost of the sorted list one by one in the order of 5,4,3,2,2,1,0, so the resulting sorted list is $[0, 1, 2, 2, 3, 4, 5]$.

In summary, our contributions are shown as follows:

(1) We formulate the sorting problem. To the best of our knowledge, we are the first to formally define the problem of sorting data into a required order.

(2) We propose an algorithm called MaxSort, which interactively searches for the greatest element and outputs the sorted list. To the best of our knowledge, we are the first to propose an effective sorting algorithm.

(3) Extensive experiments are conducted to test the effectiveness and efficiency of the MaxSort algorithm. Experiment results demonstrate an effective and stable performance of MaxSort on two sets of synthetic datasets.

2. Problem Definition

In this section, we formally define the sorting problem. The objective is to organize a given set of data into a specified order while adhering to specific constraints. We define the sorting problem as follows:

Given a list of data $L[0 : n - 1]$ and a defined order relation $\leq$, our goal is to produce a sorted output list $L^{'}$ that meets the following criteria:

(1) Monotonic Order: The output list $L^{'}$ is in monotonic order, i.e., each element in the output list is no smaller than the previous element. In formal language,

$$\forall i \in \{1, 2, \ldots, n - 1\}, L^{'}[i - 1] \leq L^{'}[i].$$

(2) Permutation of input: The output list $L^{'}$ is a permutation of the input list. Let $L = [e_0, e_1, \ldots, e_{n-1}]$, then there exist an bijection function (one-to-one correspondence)

$\sigma : \{0, 1, \ldots, n-1\} \mapsto \{0, 1, \ldots, n-1\}$ s.t.

$$L^{'} = [e_{\sigma(0)}, e_{\sigma(1)}, \ldots, e_{\sigma(n-1)}].$$

For instance, given a list of numbers $L = [0, 4, 3, 5, 2, 1, 2]$ and the numerical order relation $\leq$, then our goal is to find the sorted list

$$L^{'} = [0, 1, 2, 2, 3, 4, 5]$$

where $L^{'}$ is in monotonic order ($0 \leq 1 \leq 2 \leq 2 \leq 3 \leq 4 \leq 5$) and $L^{'}$ is a permutation of L.

3. Algorithm

In this section, we describe the MaxSort algorithm in detail. The algorithm is designed to sort a list of data by iteratively selecting the greatest element from the unsorted list and appending it to the sorted list. We introduce the details of the MaxSort algorithm in section 3.1, followed by the theoretical analysis in section 3.2.

3.1 Algorithm Details

Algorithm 1 MaxSort Algorithm

Input: A list L

Output: The corresponding sorted list

```
1:  sortedList ⇐ []
2:  unsortedList ⇐ $L$
3:  while unsortedList is not empty do
4:      $e_{max} \Leftarrow -\infty$
5:      for each element $e$ in unsortedList do
6:          if $e_{max} \leq e$ then
7:              $e_{max} \Leftarrow e$
8:          end if
9:      end for
10:     Remove $e_{max}$ from unsortedList
11:     Append $e_{max}$ to the left of sortedList
12: end while
13: return sortedList
```

The MaxSort algorithm is presented in Algorithm 1. The algorithm takes a list of data L as input and outputs the sorted list. The algorithm initializes two lists: sortedList and unsortedList. The sortedList is initially an empty list, and the unsortedList is set to be identical to the input list L. The algorithm iteratively selects the largest element from the unsortedList, removes it from the unsortedList, and appends it to the left of the sortedList. The process continues until the unsortedList is empty. The algorithm returns the sortedList as the output.

An example of the MaxSort algorithm is shown as follows.

Table 1: An example of the MaxSort algorithm

iteration	unsortedList	e_{max}	sortedList
0	$[0, 4, 3, 5, 2, 1, 2]$	$-\infty$	$[]$
1	$[0, 4, 3, 2, 1, 2]$	5	$[\mathbf{5}]$
2	$[0, 3, 2, 1, 2]$	4	$[\mathbf{4}, 5]$
3	$[0, 2, 1, 2]$	3	$[\mathbf{3}, 4, 5]$
4	$[0, 2, 1]$	2	$[\mathbf{2}, 3, 4, 5]$
5	$[0, 1]$	2	$[\mathbf{2}, 2, 3, 4, 5]$
6	$[0]$	1	$[\mathbf{1}, 2, 2, 3, 4, 5]$
7	$[]$	0	$[\mathbf{0}, 1, 2, 2, 3, 4, 5]$

As shown in Table 1, the MaxSort algorithm initialized the unsortedList as $[0, 4, 3, 5, 2, 1, 2]$ and the sortedList as an empty list. In each iteration, the algorithm selects the largest element from the unsortedList and appends it to the left of the sortedList. After 7 iterations, the unsortedList is empty, and the sortedList is $[0, 1, 2, 2, 3, 4, 5]$, which is the desired output of the algorithm.

3.2 Theoretical Analysis

In this section, we provide a theoretical analysis of the MaxSort algorithm. In section 3.2.1, we prove the correctness of the MaxSort algorithm. In section 3.2.2, we analyze the time complexity of the MaxSort algorithm. In section 3.2.3, we analyze the space complexity of the MaxSort algorithm.

3.2.1 Correctness

To prove the correctness of the MaxSort algorithm, we need to show that the output list is in monotonic order and is a permutation of the input list. Denote the input list as $L[0 : n-1]$, and the output list as $L^{'}[0 : n-1]$.

Monotonic Order: Since the MaxSort algorithm iteratively selects the largest element from the unsorted list and appends it to the left of the sorted list, for any $i \in \{1, 2, \ldots, n\}$, the element $L^{'}[n-i]$ is selected and appended to the sorted list in the i-th iteration.

In the i-th (where $i \in \{2, 3, \ldots, n\}$) iteration, since $L^{'}[n-i]$ is not selected in the previous iterations, it is not the largest element in the previous iterations, which implies that it is no larger than the previously selected elements, i.e.,

$$L^{'}[n-i] \leq L^{'}[n-j], \forall j \in \{1, 2, \ldots, i-1\}$$

As a result, we have

$$L^{'}[n-i] \leq L^{'}[n-i+1], \forall i \in \{2, 3, \ldots, n\}$$

i.e.,

$$L^{'}[i-1] \leq L^{'}[i], \forall i \in \{0, 1, \ldots, n-1\}$$

which indicates that the output list $L^{'}$ is in monotonic order.

Permutation of Input: Let the input list (and the initialized unsorted list) be $L = [e_0, e_1, \ldots, e_{n-1}]$. In each iteration, one element is moved from the unsorted list to the sorted list. Define a function $\sigma : \{0, 1, \ldots, n-1\} \mapsto \{0, 1, \ldots, n-1\}$. If e_j is the element moved in the i-th iteration, then let $\sigma(n-i) = j$. After n iterations, all elements in the input list are moved to the sorted list, and all the indexes in $\{0, 1, \ldots, n-1\}$ are covered by σ. Since an element can only be moved once, σ is a bijection function. We also have

$$L^{'} = [e_{\sigma(0)}, e_{\sigma(1)}, \ldots, e_{\sigma(n-1)}]$$

which indicates that the output list $L^{'}$ is a permutation of the input list L.

3.2.2 Time complexity

The time complexity of the MaxSort algorithm is analyzed as follows. We measure the time complexity of the MaxSort algorithm by counting the number of comparisons. In the i-th iteration, the algorithm needs to compare each element in the unsorted list to find the largest element, which requires $n-i$ comparisons. Since the algorithm iterates n times, the total number of comparisons is

$$\sum_{i=1}^{n-1} (n-i) = \frac{n(n-1)}{2}$$

Therefore, the time complexity of the MaxSort algorithm is $O(n^2)$.

3.2.3 Space complexity

The space complexity of the MaxSort algorithm is analyzed as follows. The algorithm initializes two lists: sortedList and unsortedList, each of which requires n units of space. No additional data structures are used that depend on the size of the input. Therefore, the space complexity of the MaxSort algorithm is $O(n)$.

4. Experiment

In this section, we present the experimental details of the MaxSort algorithm. We first introduce the experimental setup in section 4.1. We then present the experimental results in section 4.2. Finally, we summarize the experimental results in section 4.3.

4.1 Experimental Setup

(1) **Algorithms:** To the best of our knowledge, the MaxSort algorithm we proposed in this paper is the only algorithm proposed to solve the sorting problem, so the MaxSort algorithm is the only algorithm used in the experiments.

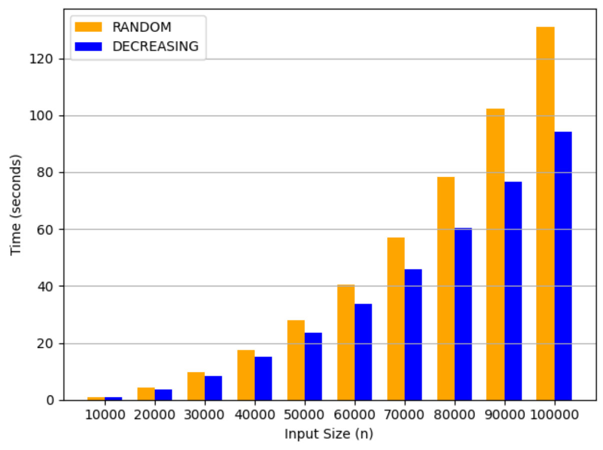
(2) **Datasets:** To fully demonstrate the effectiveness and efficiency of the MaxSort algorithm, we conduct experiments on two sets of synthetic datasets. The first dataset is called RAMDOM, which is a random permutation of the list of itergers $[0, 1, \ldots, n-1]$. The second dataset is called DECREASING, which is a list of integers from $n-1$ to 0 in decreasing order.

(3) **Factors:** The factor of the experiment is the size of the input list (i.e., n), which ranges from 10000 to 100000 with an interval of 10000.

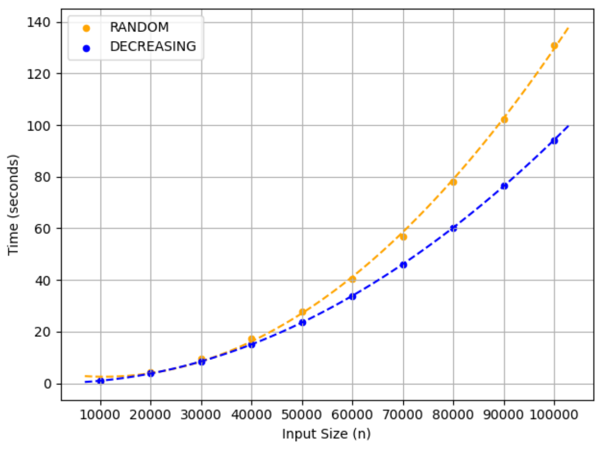
(4) **Evaluation Metrics:** To evaluate the efficiency of the MaxSort algorithm, we measure the running time and the memory usage of the algorithm on each dataset. We also check the correctness of the output to ensure the effectiveness of the algorithm.

(5) **Implementation Details:** The MaxSort algorithm is implemented in Python 3.12. The experiments are conducted on a Linux server with an Intel 1.9 GHz Intel Emerald Rapids CPU and 512GB of RAM.

4.2 Experimental Results



(a)



(b)

Figure 1: Run time of the MaxSort algorithm on the RANDOM and DECREASING datasets. The results of the RANDOM dataset are the average of 10 runs. In figure (a), the run time is shown in a bar plot. In figure (b), in addition to the run time scatter plots, the quadratic fitting curves are also shown in dotted line.

The experimental results of the time complexity of the MaxSort algorithm are shown in Figure 1. The results demonstrate the efficiency of the MaxSort algorithm on both the RANDOM and DECREASING datasets. When the size of the input list is set to be 100000, which is a significantly large size, the run time of the MaxSort algorithm is around 2 minute on both datasets. As shown in Figure 1 (b), for the experiments on each dataset, the run time of the MaxSort algorithm increases with the size of the input list, and the increasing pattern generally follows a quadratic curve, which is consistent with the theoretical analysis results, that the time complexity of the MaxSort algorithm is $O(n^2)$. As shown in Figure 1 (a), the run time of the MaxSort algorithm on the DECREASING dataset is generally smaller than that on the RANDOM dataset for the same size of the input list. This is because the MaxSort algorithm changes the value of the variable e_{max} less frequently in the DECREASING dataset than in the RANDOM dataset. When the input list is in decreasing order, the largest element of the unsorted list is always at the leftmost element. When searching for the largest element in the unsorted list, once the value of e_{max} is set to be the value of the leftmost element, it will not be changed in the following searching. Thus the value of e_{max} is changed less frequently, which results in a smaller run time of the MaxSort algorithm on the DECREASING dataset.

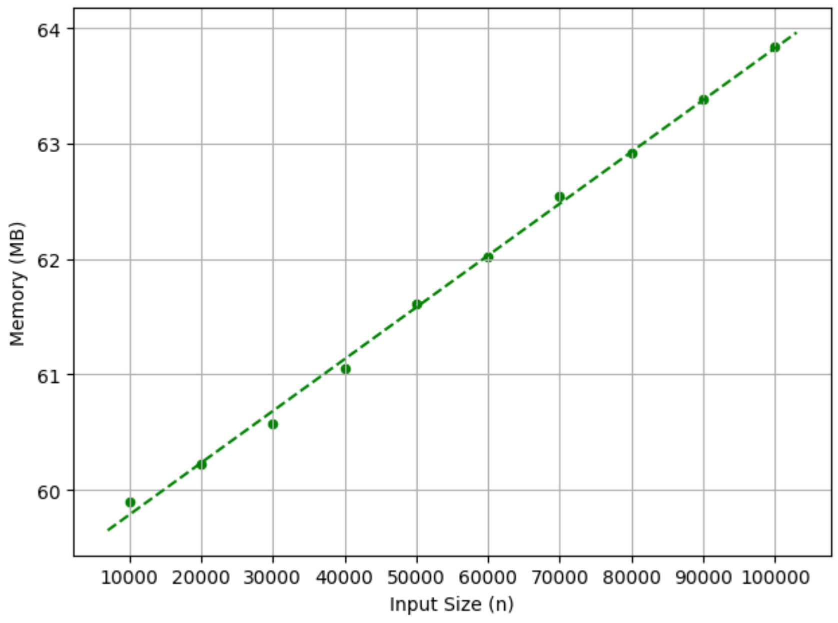


Figure 2: Memory usage of the MaxSort algorithm on the RANDOM and DECREASING datasets. Since the memory usage is identical for the experiments on both datasets, only the results of the RANDOM dataset are shown. In addition to the scatter plot of the memory usage of the MaxSort algorithm on the RANDOM dataset, the linear fitting curve is also shown in dotted line.

The experimental results of the space complexity of the MaxSort algorithm are shown in Figure 2. When the size of the input list is set to be 100000, the memory usage of the MaxSort algorithm is around 64MB. As shown in the figure, the memory usage of the MaxSort algorithm is linearly increasing with the size of the input list. This is consistent with the theoretical analysis results that the space complexity of the MaxSort algorithm is $O(n)$.

4.3 Summary

The experiments demonstrate that the MaxSort algorithm is efficient in terms of both time and space complexity. The run time increases quadratically with the size of the input list, and the memory usage increases linearly with the size of the input list, which are consistent with our theoretical analysis. Even when the size of the input list is set to 100000, which is significantly large, the MaxSort algorithm can still run efficiently on both the RANDOM and DECREASING datasets, finishing the sorting task in around 2 minutes and using a reasonable amount of memory.

5. Conclusion

In this paper, we address the sorting problem, which is to reorder a list of data into a specific order. We present a comprehensive formulation of the sorting problem and propose a novel algorithm called MaxSort to solve the problem. The MaxSort algorithm iteratively selects the largest element from the unsorted list and appends it to the sorted list. Extensive experiments are conducted to test the effectiveness and efficiency of the MaxSort algorithm. Results indicate that the MaxSort algorithm performs reliably and efficiently on two sets of synthetic datasets. Future work may involve exploring the application of the MaxSort algorithm in various domains and further optimizing the algorithm to enhance its performance.