Weird Algorithms

Amy Wilder

April 15, 2024

1 Compare All Sort

The Compare All sorting algorithm is an alternative to Counting Sort which can be used with collections of integers of any value, or even non-integers with defined comparison operators.

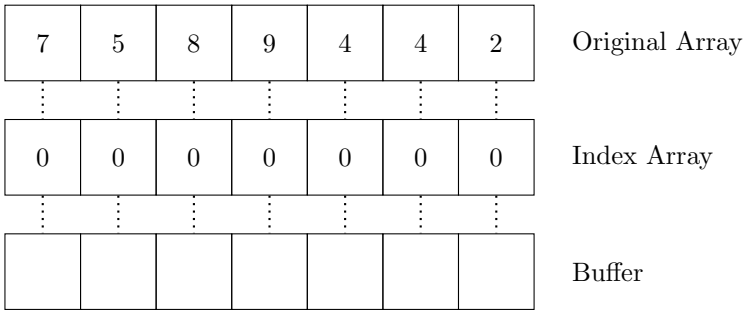
The algorithm achieves this by comparing each element in the collection against every other element in the collection to determine the sorted index of each element.

To operate, the algorithm requires one of two buffers in addition to the original array, each of which would have the same number of elements as the original array.

The first of these, hereafter referred to as the "Index Array", will store the indices of each element in the original array sorted by associated value. These are found by counting the number of elements in the original array whose values come before[1] that of the element at the corresponding position in the original array.

The second acts as a buffer for storing the unsorted values of the original array. This is used to prevent the Index Array's values from becoming invalidated while shifting the original array's elements. Whether the buffer is used to output the sorted array, or to store a copy of the unsorted array while overwriting the elements in the original, is up to implementation.

[1] The comparison may be strictly less than, or less than or equal to. This is dependent on the position in the original array of the element being inspected, relative to that of the element being sorted.



To execute the algorithm, iterate over each index i and each other index j in the original array

- Where $j < i$, count the number of j-elements with values less than or equal to the value of the i-element.
- Where $j = i$, do nothing.
- Where $j > i$, count the number of j-elements with values strictly less than the value of the i-element.

That is to say:

let A be the original array, I be the index array, and n be the length of A. Then,

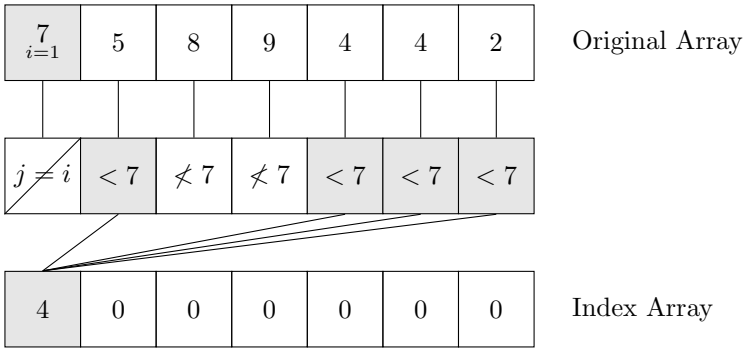
$$I_i = \sum_{j=1}^{n} \begin{cases} A_j \leq A_i & j < i \\ \text{skip} & j = i \\ A_j < A_i & j > i \end{cases}$$

The accumulated result once all elements j have been inspected is assigned to the Index Array at the position corresponding with element i.

For example:

Start at element $i = 1$, whose value is 7, skipping j where $j = i$:

4 values strictly less than 7 exist in the array: 5, 4, 4, and 2; so the value at position $i = 1$ of the Index Array is 4.

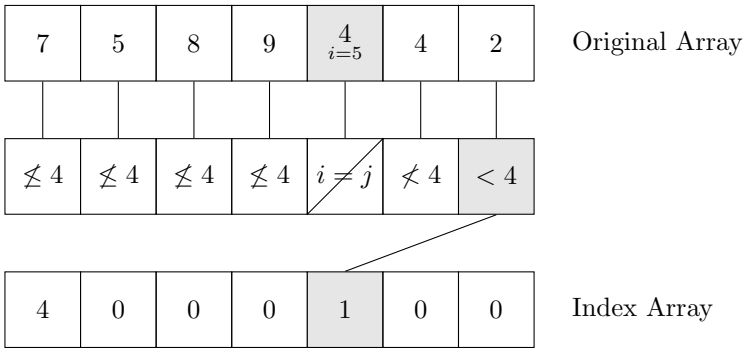


Because an array may contain duplicate values, elements of index $j < i$ are inspected with $A_j \leq A_i$ instead of $A_j < A_i$. This allows duplicates to remain in their original order with respect to each other, while sorting the duplicates with respect to all other elements.

For an example of this, consider $i = 5$ and $i = 6$.

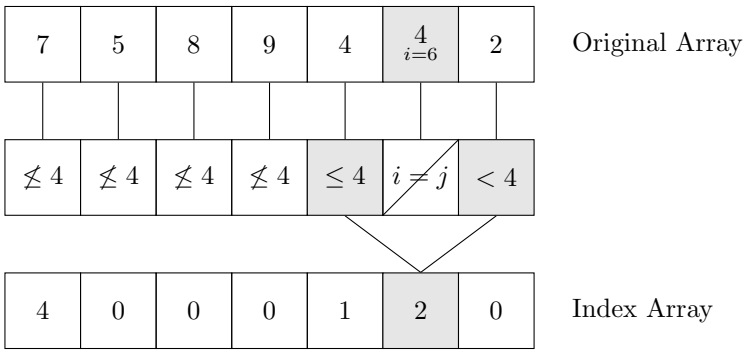
First, $i = 5$:

The element $i = 5$ has a value of 4. So only one value, 2, meets the criteria to be counted; making the value at position $i = 5$ of the Index Array 1.

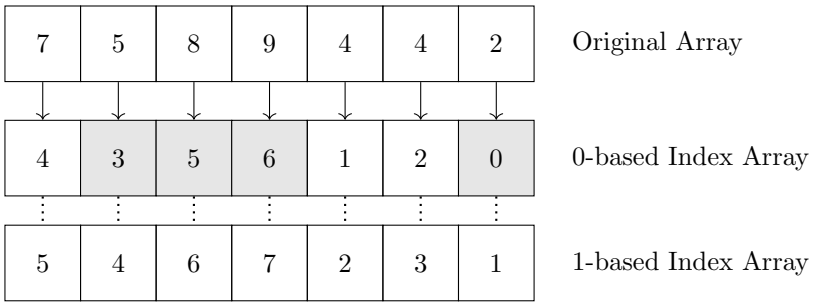


Now, $i = 6$:

The element $i = 6$ also has a value of 4. So two values, 4 and 2, meet the criteria to be counted; making the value at position $i = 6$ of the Index Array 2.

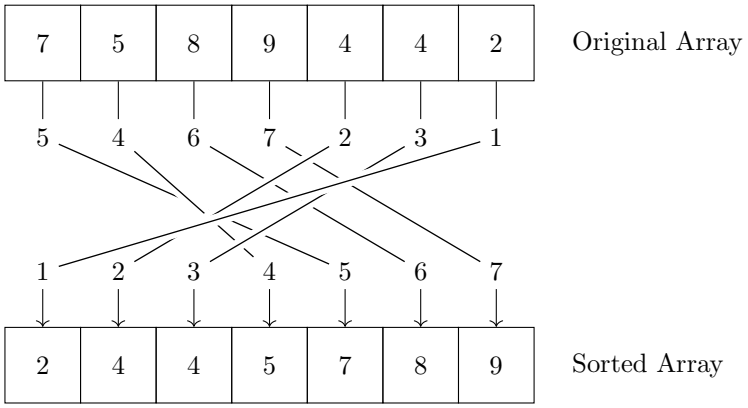


By applying this method to each element i of the original array, we get the following result:



Finally, the elements of the original array are ordered such that each one's new index is that of the value in the Index Array corresponding with its current index. So,

$$A'_i = A_{I_i}$$



The clear drawbacks of this algorithm are that it has a time complexity of $\mathcal{O}(n^2)$ (every element must be compared with every other element) and a space complexity of $\mathcal{O}(n)$ (either a buffer of the original collection's elements, or a collection of its sorted indices).

Of note however, the algorithm's time complexity is consistent regardless of the original array's order prior to sorting. $\mathcal{O}(n^2)$ is not only the average case, but also the worst *and* best case.

Time Complexity			Space Complexity
Best Case	Average Case	Worst Case	Worst Case
$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$

The following is an example of a Compare All Sort algorithm optimized for focus on iterators instead of obtaining a sorted copy of the original array.

This version has time complexity $\mathcal{O}(n^2)$ in all cases. If only iterating, it has space complexity $\mathcal{O}(n)$. If this version is used to obtain a sorted copy of the original array, it will have space complexity $\mathcal{O}(2n)$. For this reason, it is recommended to use algorithm 2 if a sorted copy is desired.

Algorithm 1: Compare All Sort — Indexed

```
Data: A
Result: I
S ← start-iterator of A
E ← end-iterator of A
I ← collection of A-iterators
foreach i ← S to E do
    I_i ← S
    foreach j ← S to i do
        if A_j ≤ A_i then
            I_i ← I_i + 1
        end
    end
    foreach j ← i + 1 to E do
        if A_j < A_i then
            I_i ← I_i + 1
        end
    end
    /* Final sorted index of A_i is already known here.     */
end
/* I now holds each of A's iterators sorted by value.      */
```

The following is an example of a Compare All Sort algorithm optimized for obtaining a sorted copy of the original array without storing a collection of iterators.

This version has identical time complexity, but space complexity $\mathcal{O}(n)$.

Algorithm 2: Compare All Sort — Buffered

```
Data: A
Result: B
S ← start-iterator of A
E ← end-iterator of A
B ← collection of (E − S) A-elements
foreach i ← S to E do
    n ← S
    foreach j ← S to i do
        if A_j ≤ A_i then
            n ← n + 1
        end
    end
    foreach j ← i + 1 to E do
        if A_j < A_i then
            n ← n + 1
        end
    end
    B_i ← A_n
end
```