Worst-Case Number of Comparisons in Sorting

Nitin Verma
mathsanew.com

November 29, 2021

Insertion Sort, *Selection Sort*, *Quick Sort* and *Merge Sort* are a few examples of *Comparison-based Sorting Algorithms*. Let us denote by A any algorithm in this class of sorting algorithms. In this article, we will create a model of algorithm A and find out how to construct an adverse input for A using this model. This will help us in finding a lower-bound on the number of comparisons which A must do in the worst-case.

Let $a = (a_0, a_1, a_2, \ldots, a_{n-1})$ be an array of n distinct elements, which needs to be sorted by A in increasing order using a comparison function f. For any two elements a_i and a_j, $f(a_i, a_j)$ will return true to indicate $a_i < a_j$, and false otherwise. Since A is a comparison-based sorting algorithm, the only information about the elements which it can use for sorting is their pairwise order obtained using f; it cannot use the value of their keys directly.

Since A can be any comparison-based sorting algorithm, we don't know how it actually works; its structure, loops and conditionals are all unknown. We only know that it would do a sequence of comparisons and output a permutation of the elements of a which is sorted. There are total $n!$ possible permutations of the n elements of a, and exactly 1 of them is sorted.

A Model of Algorithm A

When algorithm A begins, each of the $n!$ permutations can possibly be the sorted permutation. When A makes its 1^{st} comparison between some elements a_i and a_j, if the order comes out to be $a_i < a_j$ (say), the best it "knows" is that any permutation in which a_i appears after a_j cannot be the sorted one. Thus, after this comparison, A is left with all those permutations in which a_i appears before a_j, and each of these again can possibly be the sorted one. It is only by another comparison between some elements that A would perhaps be further reducing this left-out set of permutations.

Copyright © 2021 Nitin Verma. All rights reserved.

Thus, at any point during execution of A, there is a set of permutations each of which can possibly be the sorted permutation, and the sorted permutation must belong to this set. We will call this set the "set of candidate permutations", and denote it as S. By a sequence of comparisons, A keeps gathering "knowledge" about the elements' order, which effectively keeps reducing S, until it is left with only 1 permutation in S; this single permutation must be the sorted one.

What we have discussed above is only a model of the algorithm A; it may be called a *Model of Candidate Permutations*. Many examples of A, like Insertion Sort and Quick Sort, do not actually maintain any set of candidate permutations. The "knowledge" they gather by doing comparisons is recorded by them in the form of their program state, like a sorted subarray or array partitions created based on a pivot. The partial sorted order thus recorded is in fact a representation of the set of candidate permutations.

The above described model of A can be expressed as below. a_i is represented as `a_i` and a_j as `a_j`. P is the set of all permutations in S with a_i occurring before a_j, and Q is the set of all the remaining ones ($S \setminus P$).

```
S <- (set of all n! permutations)

while(|S| > 1)
{
  if(f(a_i, a_j))
    S <- P
  else
    S <- Q
}

/* Output the single permutation left in S as the sorted one */
```

Note that after any k number of iterations, each permutation in S is consistent with the output of all k comparisons made so far. For example, if two of these comparisons returned $a_2 < a_1$ and $a_5 < a_2$, each permutation in S must have a_5 appearing before a_2, and a_2 appearing before a_1.

Also, since subsets P and Q are simply two partitions of set S, the larger of P and Q must have cardinality at least $|S|/2$.

This model of algorithm A extracts from A only its sequence of comparisons and attaches to that sequence a set S of all permutations possible under the comparison responses gathered so far. As long as S contains more than one permutation, A has no way to tell the correct sorted permutation; if it makes any "guess", that can be wrong. So, A cannot terminate until S has only one permutation left.

Constructing an Adverse Input

For an input array a of n elements, algorithm A may require a varying number of comparisons depending upon the initial order of the elements. We are interested in the maximum among these number of comparisons, which can be called the number of comparisons in the "worst-case", and denoted C_w.

Since we do not know how exactly A works, it is impossible to construct its worst-case input. What we will instead do is construct an adverse input and count the least number of comparisons C_d which A would do for that input. Since we must have $C_w \geq C_d$, C_d will give us a lower-bound on C_w.

To construct an adverse input, we will take some array a with its keys uninitialized and modify the comparison function f as follows. f will keep track of the set S. To compare a_i and a_j, instead of using their keys, it will check how many permutations in S have a_i occurring before a_j (the subset P defined earlier). If it finds $|P| \geq |Q|$, it will respond with output "$a_i < a_j$" (true), otherwise it will respond with "$a_i \geq a_j$" (false). This will result in the algorithm A always left with the larger of the two subsets P and Q, hence more number of candidate permutations to work with. As mentioned earlier, this larger subset must have cardinality at least $|S|/2$.

Since this modified function f does not use the keys, throughout the execution of algorithm A the keys would remain unused; that's why we could leave them uninitialized. Now, how can we really initialize the keys of the input array a, so that A would execute in exactly the same manner, but with the original comparison function f? In other words, how can we construct such adverse input?

For that, we will use the sorted output permutation from above execution of A. The keys of input array a can be initialized such that their sorted order would be the same as this permutation. For example, with $n = 5$, if the output permutation obtained above is $(a_3, a_0, a_2, a_4, a_1)$, the keys of a should be chosen such that $a_3 < a_0 < a_2 < a_4 < a_1$. For integer keys with f based on their numerical order, this can be done, for example, by choosing the keys for a_3, a_0, a_2, a_4 and a_1 as $1, 2, 3, 4$ and 5 respectively.

We know that the output permutation must be consistent with the result of all the comparisons made in above execution of A. So, the adverse input constructed using its order must cause exactly the same comparison results and same execution flow of A.

Having constructed an adverse input, let us now count C_d. After each comparison, A is left with the subset having cardinality at least $|S|/2$. Since

the cardinality of S is $n!$ initially, after k comparisons, it must be at least $(n!)/2^k$. So algorithm A will require at least C_d comparisons till the cardinality becomes 1, where:

$$\begin{aligned} & \frac{n!}{2^{C_d}} \leq 1 \\ \Leftrightarrow \quad & C_d \geq \log_2(n!) \end{aligned}$$

And since $C_w \geq C_d$, we have:

$$C_w \geq \log_2(n!) \tag{1}$$

It can be proved that $\log_2(n!)$ belongs to the set $\Theta(n \log_2 n)$ (see [1]: chapter 3 section "Factorials", and exercise 8.1-2). So we conclude that, any comparison-based sorting algorithm will do at least $\log_2(n!)$ ($\Theta(n \log_2 n)$) comparisons for its worst-case input.

■

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*, Third Edition. The MIT Press (2009).