Algorithms

FOURTH EDITION

ROBERT SEDGEWICK | KEVIN WAYNE

Algorithms

FOURTH EDITION

Proposition H. Bottom-up mergesort uses between ~ ½ $n \lg n$ and $n \lg n$ compares and at most $6n \lg n$ array accesses to sort an array of length n.

Proof: The number of passes through the array is precisely $\lceil \lg n \rceil$ (that is precisely the value of k such that $2^{k-1} < n \leq 2^k$). For each pass, the number of array accesses is at most $6n$ and the number of compares is at most n and no less than $\lfloor n/2 \rfloor$.

WHEN THE ARRAY LENGTH IS A POWER OF 2, top-down and bottom-up mergesort perform precisely the same compares and array accesses, just in a different order. When the array length is not a power of 2, the compares, array accesses, and subarray lengths for the two algorithms will typically be different (see EXERCISE 2.2.5).

A version of bottom-up mergesort is the method of choice for sorting data organized in a *linked list*. Consider the list to be sorted sublists of length 1, then pass through to make sorted sublists of length 2 linked together, then length 4, and so forth. This method rearranges the links to sort the list *in place* (without creating any new list nodes).

Both the top-down and bottom-up approaches to implementing a divide-and-conquer algorithm are intuitive. The lesson that you can take from mergesort is this: Whenever you encounter an algorithm based on one of these approaches, it is worth considering the other. Do you want to solve the problem by breaking it up into smaller problems (and solving them recursively) as in `Merge.sort()` or by building small solutions into larger ones as in `MergeBU.sort()`?

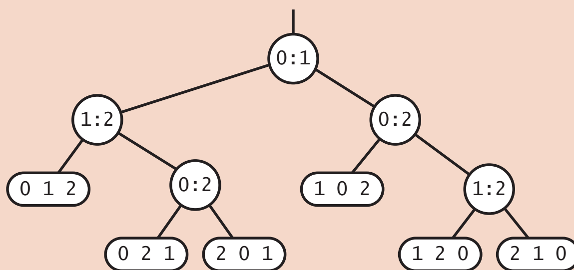
The complexity of sorting

One important reason to know about mergesort is that we use it as the basis for proving a fundamental result in the field of *computational complexity* that helps us understand the intrinsic difficulty of sorting. In general, computational complexity plays an important role in the design of algorithms, and this result in particular is directly relevant to the design of sorting algorithms, so we next consider it in detail.

The first step in a study of complexity is to establish a model of computation. Generally, researchers strive to understand the simplest model relevant to a problem. For sorting, we study the class of *compare-based* algorithms that make their decisions about items only on the basis of comparing keys. A compare-based algorithm can do an arbitrary amount of computation between compares, but cannot get any information about a key except by comparing it with another one. Because of our restriction to the `Comparable` API, all of the algorithms in this chapter are in this class (note that we are ignoring the cost of array accesses), as are many algorithms that we might imagine. In CHAPTER 5, we consider algorithms that are not restricted to `Comparable` items.

Proposition I. No compare-based sorting algorithm can guarantee to sort n items with fewer than $\lg(n!) \sim n \lg n$ compares.

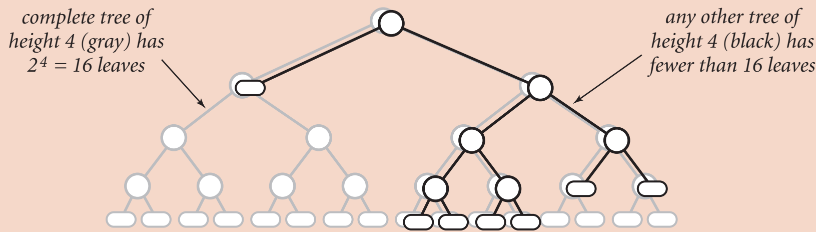
Proof: First, we assume that the keys are all distinct, since any algorithm must be able to sort such inputs. Now, we use a binary tree to describe the sequence of compares. Each *node* in the tree is either a *leaf* ($i_0\ i_1\ i_2\ \ldots\ i_{n-1}$) that indicates that the sort is complete and has discovered that the original inputs were in the order $a[i_0], a[i_1], \ldots a[i_{n-1}]$, or an *internal node* (i:j) that corresponds to a compare operation between `a[i]` and `a[j]`, with a left subtree corresponding to the sequence of compares in the case that `a[i]` is less than `a[j]`, and a right subtree corresponding to what happens if `a[i]` is greater than `a[j]`. Each path from the root to a leaf corresponds to the sequence of compares that the algorithm uses to establish the ordering given in the leaf. For example, here is a compare tree for $n = 3$:



We never explicitly construct such a tree—it is a mathematical device for describing the compares used by any algorithm.

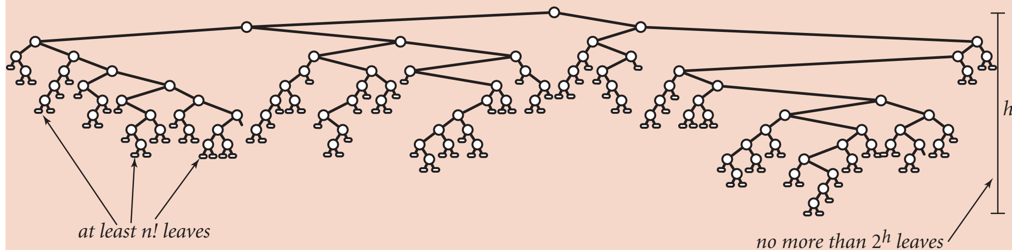
The first key observation in the proof is that the tree must have at least $n!$ leaves because there are $n!$ different permutations of n distinct keys. If there are fewer than $n!$ leaves, then some permutation is missing from the leaves, and the algorithm would fail for that permutation.

The number of internal nodes on a path from the root to a leaf in the tree is the number of compares used by the algorithm for some input. We are interested in the length of the longest such path in the tree (known as the tree *height*) since it measures the worst-case number of compares used by the algorithm. Now, it is a basic combinatorial property of binary trees that a tree of height h has no more than 2^h leaves—the tree of height h with the maximum number of leaves is perfectly balanced, or *complete*. An example for $h = 4$ is diagrammed on the next page.



Combining the previous two paragraphs, we have shown that any compare-based sorting algorithm corresponds to a compare tree of height h with

$$n! \le \textit{number of leaves} \le 2^h$$



The value of h is precisely the worst-case number of compares, so we can take the logarithm (base 2) of both sides of this equation and conclude that the number of compares used by any algorithm must be at least lg $(n!)$. The approximation lg $(n!) \sim n$ lg n follows immediately from Stirling's approximation to the factorial function (see page 185).

This result serves as a guide for us to know, when designing a sorting algorithm, how well we can expect to do. For example, without such a result, one might set out to try to design a compare-based sorting algorithm that uses half as many compares as does mergesort, in the worst case. The lower bound in PROPOSITION I says that such an effort is futile—*no such algorithm exists*. It is an extremely strong statement that applies to any conceivable compare-based algorithm.

PROPOSITION H asserts that the number of compares used by mergesort in the worst case is $\sim n$ lg n. This result is an *upper bound* on the difficulty of the sorting problem in the sense that a better algorithm would have to guarantee to use a smaller number of compares. PROPOSITION I asserts that no sorting algorithm can guarantee to use fewer

than $\sim n \lg n$ compares. It is a *lower bound* on the difficulty of the sorting problem in the sense that even the best possible algorithm must use at least that many compares in the worst case. Together, they imply:

Proposition J. Mergesort is an asymptotically optimal compare-based sorting algorithm.

Proof: Precisely, we mean by this statement that *both the number of compares used by mergesort in the worst case and the minimum number of compares that any compare-based sorting algorithm can guarantee are* $\sim n \lg n$. PROPOSITIONS H and I establish these facts.

It is important to note that, like the model of computation, we need to precisely define what we mean by an optimal algorithm. For example, we might tighten the definition of optimality and insist that an optimal algorithm for sorting is one that uses *precisely* $\lg(n!)$ compares. We do not do so because we could not notice the difference between such an algorithm and (for example) mergesort for large n. Or, we might broaden the definition of optimality to include any sorting algorithm whose worst-case number of compares is *within a constant factor* of $n \lg n$. We do not do so because we might very well notice the difference between such an algorithm and mergesort for large n.

COMPUTATIONAL COMPLEXITY MAY SEEM RATHER ABSTRACT, but fundamental research on the intrinsic difficulty of solving computational problems hardly needs justification. Moreover, when it does apply, it is emphatically the case that computational complexity affects the development of good software. First, good upper bounds allow software engineers to provide performance guarantees; there are many documented instances where poor performance has been traced to someone using a quadratic sort instead of a linearithmic one. Second, good lower bounds spare us the effort of searching for performance improvements that are not attainable.

But the optimality of mergesort is not the end of the story and should not be misused to indicate that we need not consider other methods for practical applications. That is not the case because the theory in this section has a number of limitations. For example:

- Mergesort is not optimal with respect to space usage.
- The worst case may not be likely in practice.
- Operations other than compares (such as array accesses) may be important.
- One can sort certain data without using *any* compares.

Thus, we shall be considering several other sorting methods in this book.

Q&A

Q. Is mergesort faster than shellsort?

A. In practice, their running times are within a small constant factor of one another (when shellsort is using a well-tested increment sequence like the one in ALGORITHM 2.3), so comparative performance depends on the implementations.

```
% java SortCompare Merge Shell 100000 100
For 100000 random Double values
    Merge is 1.2 times faster than Shell
```

In theory, no one has been able to prove that shellsort is linearithmic for random data, so there remains the possibility that the asymptotic growth of the average-case performance of shellsort is higher. Such a gap has been proven for worst-case performance, but it is not relevant in practice.

Q. Why not make the `aux[]` array local to `merge()`?

A. To avoid the overhead of creating an array for every merge, even the tiny ones. This cost would dominate the running time of mergesort (see EXERCISE 2.2.26). A more proper solution (which we avoid in the text to reduce clutter in the code) is to make `aux[]` local to `sort()` and pass it as an argument to `merge()` (see EXERCISE 2.2.9).

Q. How does mergesort fare when there are duplicate values in the array?

A. If all the items have the same value, the running time is linear (with the extra test to skip the merge when the array is sorted), but if there is more than one duplicate value, this performance gain is not necessarily realized. For example, suppose that the input array consists of n items with one value in odd positions and n items with another value in even positions. The running time is linearithmic for such an array (it satisfies the same recurrence as for items with distinct values), not linear.

EXERCISES

2.2.1 Give a trace, in the style of the trace given at the beginning of this section, showing how the keys `A E Q S U Y E I N O S T` are merged with the abstract in-place `merge()` method.

2.2.2 Give traces, in the style of the trace given with ALGORITHM 2.4, showing how the keys `E A S Y Q U E S T I O N` are sorted with top-down mergesort.

2.2.3 Answer EXERCISE 2.2.2 for bottom-up mergesort.

2.2.4 Does the abstract in-place merge produce proper output if and only if the two input subarrays are in sorted order? Prove your answer, or provide a counterexample.

2.2.5 Give the sequence of subarray lengths in the merges performed by both the top-down and the bottom-up mergesort algorithms, for $n = 39$.

2.2.6 Write a program to compute the exact value of the number of array accesses used by top-down mergesort and by bottom-up mergesort. Use your program to plot the values for n from 1 to 512, and to compare the exact values with the upper bound $6n \lg n$.

2.2.7 Show that the number of compares used by mergesort is monotonically increasing ($C(n+1) > C(n)$ for all $n > 0$).

2.2.8 Suppose that ALGORITHM 2.4 is modified to skip the call on `merge()` whenever `a[mid] <= a[mid+1]`. Prove that the number of compares used to mergesort a sorted array is linear.

2.2.9 Use of a static array like `aux[]` is inadvisable in library software because multiple clients might use the class concurrently. Give an implementation of `Merge` that does not use a static array. Do *not* make `aux[]` local to `merge()` (see the Q&A for this section). *Hint*: Pass the auxiliary array as an argument to the recursive `sort()`.