4 Sorting and Searching

Typical computer science students study the basic sorting algorithms at least three times before they graduate: first in introductory programming, then in data structures, and finally in their algorithms course. Why is sorting worth so much attention? There are several reasons:

- Sorting is the basic building block that many other algorithms are built around. By understanding sorting, we obtain an amazing amount of power to solve other problems.
- Most of the interesting ideas used in the design of algorithms appear in the context of sorting, such as divide-and-conquer, data structures, and randomized algorithms.
- Computers have historically spent more time sorting than doing anything else. A quarter of all mainframe cycles were spent sorting data [Knu98]. Sorting remains the most ubiquitous combinatorial algorithm problem in practice.
- Sorting is the most thoroughly studied problem in computer science. Literally dozens of different algorithms are known, most of which possess some particular advantage over all other algorithms in certain situations.

In this chapter, we will discuss sorting, stressing how sorting can be applied to solving other problems. In this sense, sorting behaves more like a data structure than a problem in its own right. We then give detailed presentations of several fundamental algorithms: heapsort, mergesort, quicksort, and distribution sort as examples of important algorithm design paradigms. Sorting is also represented by Section 14.1 (page 436) in the problem catalog.

S.S. Skiena, *The Algorithm Design Manual*, 2nd ed., DOI: 10.1007/978-1-84800-070-4_4,
© Springer-Verlag London Limited 2008

4.1 Applications of Sorting

We will review several sorting algorithms and their complexities over the course of this chapter. But the punch-line is this: clever sorting algorithms exist that run in $O(n \log n)$. This is a *big* improvement over naive $O(n^2)$ sorting algorithms for large values of n. Consider the following table:

n	$n^2/4$	$n \lg n$
10	*25*	33
100	2,500	*664*
1,000	250,000	*9,965*
10,000	25,000,000	*132,877*
100,000	2,500,000,000	*1,660,960*

You might still get away with using a quadratic-time algorithm even if $n = 10,000$, but quadratic-time sorting is clearly ridiculous once $n \geq 100,000$.

Many important problems can be reduced to sorting, so we can use our clever $O(n \log n)$ algorithms to do work that might otherwise seem to require a quadratic algorithm. An important algorithm design technique is to use sorting as a basic building block, because many other problems become easy once a set of items is sorted.

Consider the following applications:

- *Searching* – Binary search tests whether an item is in a dictionary in $O(\log n)$ time, provided the keys are all sorted. Search preprocessing is perhaps the single most important application of sorting.

- *Closest pair* – Given a set of n numbers, how do you find the pair of numbers that have the smallest difference between them? Once the numbers are sorted, the closest pair of numbers must lie next to each other somewhere in sorted order. Thus, a linear-time scan through them completes the job, for a total of $O(n \log n)$ time including the sorting.

- *Element uniqueness* – Are there any duplicates in a given set of n items? This is a special case of the closest-pair problem above, where we ask if there is a pair separated by a gap of zero. The most efficient algorithm sorts the numbers and then does a linear scan though checking all adjacent pairs.

- *Frequency distribution* – Given a set of n items, which element occurs the largest number of times in the set? If the items are sorted, we can sweep from left to right and count them, since all identical items will be lumped together during sorting.

  To find out how often an arbitrary element k occurs, look up k using binary search in a sorted array of keys. By walking to the left of this point until the first the element is not k and then doing the same to the right, we can find