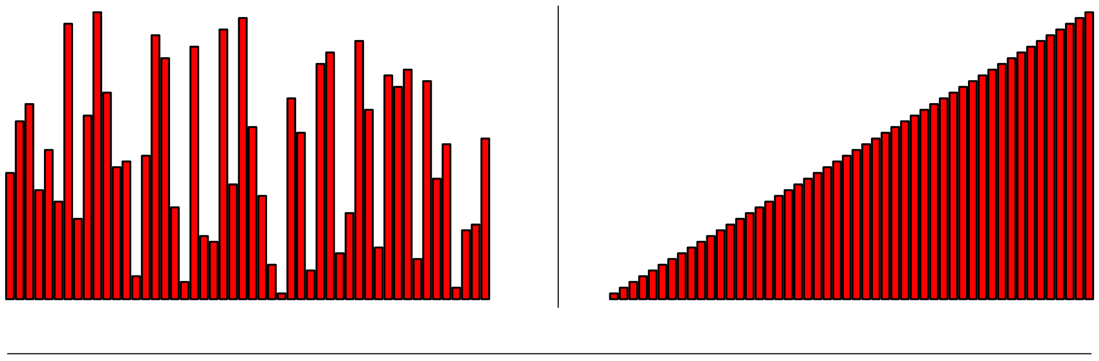

INPUT OUTPUT

14.1 Sorting

Input description: A set of n items.

Problem description: Arrange the items in increasing (or decreasing) order.

Discussion: Sorting is the most fundamental algorithmic problem in computer science. Learning the different sorting algorithms is like learning scales for a musician. Sorting is the first step in solving a host of other algorithm problems, as shown in Section 4.2 (page 107). Indeed, *"when in doubt, sort"* is one of the first rules of algorithm design.

Sorting also illustrates all the standard paradigms of algorithm design. The result is that most programmers are familiar with many different sorting algorithms, which sows confusion as to which should be used for a given application. The following criteria can help you decide:

- *How many keys will you be sorting?* – For small amounts of data (say $n \leq 100$), it really doesn't matter much which of the quadratic-time algorithms you use. Insertion sort is faster, simpler, and less likely to be buggy than bubblesort. Shellsort is closely related to, but much faster than, insertion sort, but it involves looking up the right insert sequences in Knuth [Knu98].

  When you have more than 100 items to sort, it is important to use an $O(n \lg n)$-time algorithm like heapsort, quicksort, or mergesort. There are various partisans who favor one of these algorithms over the others, but since it can be hard to tell which is fastest, it usually doesn't matter.

  Once you get past (say) 5,000,000 items, it is important to start thinking about external-memory sorting algorithms that minimize disk access. Both types of algorithm are discussed below.

- *Will there be duplicate keys in the data?* – The sorted order is completely defined if all items have distinct keys. However, when two items share the same key, something else must determine which one comes first. In many applications it doesn't matter, so any sorting algorithm suffices. Ties are often broken by sorting on a secondary key, like the first name or initial when the family names collide.

  Occasionally, ties need to be broken by their initial position in the data set. Suppose the 5th and 27th items of the initial data set share the same key. This means the 5th item must appear before the 27th in the final order. A *stable* sorting algorithm preserves the original ordering in case of ties. Most of the quadratic-time sorting algorithms are stable, while many of the $O(n \lg n)$ algorithms are not. If it is important that your sort be stable, it is probably better to explicitly use the initial position as a secondary key in your comparison function rather than trust the stability of your implementation.

- *What do you know about your data?* – You can often exploit special knowledge about your data to get it sorted faster or more easily. Of course, general sorting is a fast $O(n \lg n)$ operation, so if the time spent sorting is really the bottleneck in your application, you are a fortunate person indeed.

  - *Has the data already been partially sorted?* If so, certain algorithms like insertion sort perform better than they otherwise would.
  - *Do you know the distribution of the keys?* If the keys are randomly or uniformly distributed, a *bucket* or *distribution sort* makes sense. Throw the keys into bins based on their first letter, and recur until each bin is small enough to sort by brute force. This is very efficient when the keys get evenly distributed into buckets. However, bucket sort would perform very badly sorting names on the membership roster of the "Smith Society."
  - *Are your keys very long or hard to compare?* If your keys are long text strings, it might pay to use a relatively short prefix (say ten characters) of each key for an initial sort, and then resolve ties using the full key. This is particularly important in external sorting (see below), since you don't want to waste your fast memory on the dead weight of irrelevant detail.

    Another idea might be to use radix sort. This always takes time linear in the number of characters in the file, instead of $O(n \lg n)$ times the cost of comparing two keys.
  - *Is the range of possible keys very small?* If you want to sort a subset of, say, $n/2$ distinct integers, each with a value from 1 to n, the fastest algorithm would be to initialize an n-element bit vector, turn on the bits corresponding to keys, then scan from left to right and report the positions with true bits.

- *Do I have to worry about disk accesses?* – In massive sorting problems, it may not be possible to keep all data in memory simultaneously. Such a problem is called *external sorting*, because one must use an external storage device. Traditionally, this meant tape drives, and Knuth [Knu98] describes a variety of intricate algorithms for efficiently merging data from different tapes. Today, it usually means virtual memory and swapping. Any sorting algorithm will run using virtual memory, but most will spend all their time swapping.

  The simplest approach to external sorting loads the data into a B-tree (see Section 12.1 (page 367)) and then does an in-order traversal of the tree to read the keys off in sorted order. Real high-performance sorting algorithms are based on multiway-mergesort. Files containing portions of the data are sorted into runs using a fast internal sort, and then files with these sorted runs are merged in stages using 2- or k-way merging. Complicated merging patterns and buffer management based on the properties of the external storage device can be used to optimize performance.

- *How much time do you have to write and debug your routine?* – If I had under an hour to deliver a working routine, I would probably just implement a simple selection sort. If I had an afternoon to build an efficient sort routine, I would probably use heapsort, for it delivers reliable performance without tuning.

The best general-purpose internal sorting algorithm is quicksort (see Section 4.2 (page 107)), although it requires tuning effort to achieve maximum performance. Indeed, you are much better off using a library function instead of coding it yourself. A poorly written quicksort will likely run more slowly than a poorly written heapsort.

If you are determined to implement your own quicksort, use the following heuristics, which make a big difference in practice:

- *Use randomization* – By randomly permuting (see Section 14.4 (page 448)) the keys before sorting, you can eliminate the potential embarrassment of quadratic-time behavior on nearly-sorted data.

- *Median of three* – For your pivot element, use the median of the first, last, and middle elements of the array to increase the likelihood of partitioning the array into roughly equal pieces. Some experiments suggest using a larger sample on big subarrays and a smaller sample on small ones.

- *Leave small subarrays for insertion sort* – Terminating the quicksort recursion and switching to insertion sort makes sense when the subarrays get small, say fewer than 20 elements. You should experiment to determine the best switchpoint for your implementation.

- *Do the smaller partition first* – Assuming that your compiler is smart enough to remove tail recursion, you can minimize run-time memory by processing the smaller partition before the larger one. Since successive stored calls are at most half as large as the previous one, only $O(\lg n)$ stack space is needed.

Before you get started, see Bentley's article on building a faster quicksort [Ben92b].

Implementations: The best freely available sort program is presumably GNU sort, part of the GNU core utilities library. See *http://www.gnu.org/software/coreutils/*. Be aware that there are also commercial vendors of high-performance external sorting programs. These include Cosort (*www.cosort.com*), Syncsort (*www.syncsort.com*) and Ordinal Technology (*www.ordinal.com*).

Modern programming languages provide libraries offering complete and efficient container implementations. Thus, you should never need to implement your own sort routine. The C standard library contains `qsort`, a generic implementation of (presumably) quicksort. The C++ *Standard Template Library* (STL) provides both `sort` and `stable_sort` methods. It is available with documentation at *http://www.sgi.com/tech/stl/*. See Josuttis [Jos99], Meyers [Mey01] and Musser [MDS01] for more detailed guides to using STL and the C++ standard library.

Java Collections (JC), a small library of data structures, is included in the `java.util` package of Java standard edition (*http://java.sun.com/javase/*). In particular, `SortedMap` and `SortedSet` classes are provided.

For a C++ implementation of an cache-oblivious algorithm (*funnelsort*), check out *http://kristoffer.vinther.name/projects/funnelsort/*. Benchmarks attest to its excellent performance.

There are a variety of websites that provide applets/animations of all the basic sorting algorithms, including bubblesort, heapsort, mergesort, quicksort, radix sort, and shellsort. Many of these are quite interesting to watch. Indeed, sorting is the canonical problem for algorithm animation. Representative examples include Harrison (*http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html*) and Bentley [Ben99] (*http://www.cs.bell-labs.com/cm/cs/pearls/sortanim.html*).

Notes: Knuth [Knu98] is the best book that has been written on sorting and indeed is the best book that will ever be written on sorting. It is now over thirty years old, but remains fascinating reading. One area that has developed since Knuth is sorting under presortedness measures, surveyed in [ECW92]. A noteworthy reference on sorting is [GBY91], which includes pointers to algorithms for partially sorted data and includes implementations in C and Pascal for all of the fundamental algorithms.

Expositions on the basic internal sorting algorithms appear in every algorithms text. Heapsort was first invented by Williams [Wil64]. Quicksort was invented by Hoare [Hoa62], with careful analysis and implementation by Sedgewick [Sed78]. Von Neumann is credited with having produced the first implementation of mergesort on the EDVAC in 1945. See Knuth for a full discussion of the history of sorting, dating back to the days of punch-card tabulating machines.

The primary competitive forum for high-performance sorting is an annual competition initiated by the late Jim Gray. See *http://research.microsoft.com/barc/SortBenchmark/* for current and previous results, which are are either inspiring or depressing depending upon how you look at it. The magnitude of progress is inspiring (the million-record instances of the original benchmarks are now too small to bother with) but it is depressing (to me) the extent that systems/memory management issues thoroughly trump the combinatorial/algorithmic aspects of sorting.

Modern attempts to engineer high-performance sort programs include work on both cache-conscious [LL99] and cache-oblivious [BFV07] sorting.

Sorting has a well-known $\Omega(n \lg n)$ lower bound under the algebraic decision tree model [BO83]. Determining the exact number of comparisons required for sorting n elements, for small values of n, has generated considerable study. See [Aig88, Raw92] for expositions and Peczarski [Pec04, Pec07] for the latest results.

This lower-bound does not hold under different models of computation. Fredman and Willard [FW93] present an $O(n\sqrt{\lg n})$ algorithm for sorting under a model of computation that permits arithmetic operations on keys. See Andersson [And05] for a survey of algorithms for fast sorting on such nonstandard models of computation.

Related Problems: Dictionaries (see page 367), searching (see page 441), topological sorting (see page 481).