2.5 APPLICATIONS

SORTING ALGORITHMS and priority queues are widely used in a broad variety of applications. Our purpose in this section is to briefly survey some of these applications, consider ways in which the efficient methods that we have considered play a critical role in such applications, and discuss some of the steps needed to make use of our sort and priority-queue code.

A prime reason why sorting is so useful is that it is much easier to search for an item in a sorted array than in an unsorted one. For over a century, people found it easy to look up someone's phone number in a phone book where items are sorted by last name. Now digital music players organize song files by artist name or song title; search engines display search results in descending order of importance; spreadsheets display columns sorted by a particular field; matrix-processing packages sort the real eigenvalues of a symmetric matrix in descending order; and so forth. Other tasks are also made easier once an array is in sorted order: from looking up an item in the sorted index in the back of this book; to removing duplicates from a long list such as a mailing list, a list of voters, or a list of websites; to performing statistical calculations such as removing outliers, finding the median, or computing percentiles.

Sorting also arises as a critical subproblem in many applications that appear to have nothing to do with sorting at all. Data compression, computer graphics, computational biology, supply-chain management, combinatorial optimization, social choice, and voting are but a few of many examples. The algorithms that we have considered in this chapter play a critical role in the development of effective algorithms in each of the later chapters in this book.

Most important is the system sort, so we begin by considering a number of practical considerations that come into play when building a sort for use by a broad variety of clients. While some of these topics are specific to Java, they each reflect challenges that need to be met in any system.

Our primary purpose is to demonstrate that, even though we have used mechanisms that are relatively simple, the sorting implementations that we are studying are widely applicable. The list of proven applications of fast sorting algorithms is vast, so we can consider just a small fraction of them: some scientific, some algorithmic, and some commercial. You will find many more examples in the exercises, and many more than that on the booksite. Moreover, we will often refer back to this chapter to effectively address the problems that we later consider *in this book*!

Sorting various types of data Our implementations sort arrays of `Comparable` objects. This Java convention allows us to use Java's *callback* mechanism to sort arrays of objects of any type that implements the `Comparable` interface. As described in SECTION 2.1, implementing `Comparable` amounts to defining a `compareTo()` method that implements a *natural ordering* for the type. We can use our code immediately to sort arrays of type `String`, `Integer`, `Double`, and other types such as `File` and `URL`, because these data types all implement `Comparable`. Being able to use the same code for all of those types is convenient, but typical applications involve working with data types that are defined for use within the application. Accordingly it is common to implement a `compareTo()` method for user-defined data types, so that they implement `Comparable`, thus enabling client code to sort arrays of that type (and build priority queues of values of that type).

Transaction example. A prototypical breeding ground for sorting applications is commercial data processing. For example, imagine that a company engaged in internet commerce maintains a record for each transaction involving a customer account that contains all of the pertinent information, such as the customer name, date, amount, and so forth. Nowadays, a successful company needs to be able to handle millions and millions of such transactions. As we saw in EXERCISE 2.1.21, it is reasonable to decide that a natural ordering of such transactions is that they be ordered by amount, which we can implement by adding an appropriate `compareTo()` method in the class definition. With such a definition, we could process an array `a[]` of `Transactions` by, for example, first sorting it with the call `Quick.sort(a)`. Our sorting methods know nothing about our `Transaction` data type, but Java's `Comparable` interface allows us to define a natural ordering so that we can use any of our methods to sort `Transaction` objects. Alternatively, we might specify that `Transaction` objects are to be ordered by date by implementing `compareTo()` to compare the `Date` fields. Since `Date` objects are themselves `Comparable`, we can just invoke the `compareTo()` method in `Date` rather than having to implement it from scratch. It is also reasonable to consider ordering this data by it customer field; arranging to allow clients the flexibility to switch among multiple different orders is an interesting challenge that we will soon consider.

```
public int compareTo(Transaction that)
{  return this.when.compareTo(that.when);  }
```

Alternate compareTo() implementation for sorting transactions by date

Pointer sorting. The approach we are using is known in the classical literature as *pointer sorting*, so called because we process references to items and do not move the data itself. In programming languages such as C and C++, programmers explicitly decide whether to manipulate data or pointers to data; in Java, pointer manipulation is implicit. Except for primitive numeric types, we *always* manipulate references to objects (pointers), not the objects themselves. Pointer sorting adds a level of indirection: the array contains references to the objects to be sorted, not the objects themselves. We briefly consider some associated issues, in the context of sorting. With multiple reference arrays, we can have multiple different sorted representations of different parts of a single body of data (perhaps using multiple keys, as described below).

Keys are immutable. It stands to reason that an array might not remain sorted if a client is allowed to change the values of keys after the sort. Similarly, a priority queue can hardly be expected to operate properly if the client can change the values of keys between operations. In Java, it is wise to ensure that key values do not change by using immutable keys. Most of the standard data types that you are likely to use as keys, such as `String`, `Integer`, `Double`, and `File`, are immutable.

Exchanges are inexpensive. Another advantage of using references is that we avoid the cost of moving full items. The cost saving is significant for arrays with large items (and small keys) because the compare needs to access just a small part of the item, and most of the item is not even touched during the sort. The reference approach makes the cost of an exchange roughly equal to the cost of a compare for general situations involving arbitrarily large items (at the cost of the extra space for the references). Indeed, if the keys are long, the exchanges might even wind up being less costly than the compare. One way to study the performance of algorithms that sort arrays of numbers is to simply look at the total number of compares and exchanges they use, implicitly making the assumption that the cost of exchanges is the same as the cost of compares. Conclusions based on this assumption are likely to apply to a broad class of applications in Java, because we are sorting reference objects.

Alternate orderings. There are many applications where we want to use different orders for the objects that we are sorting, depending on the situation. The Java `Comparator` interface allows us to build multiple orders within a single class. It has a single public method `compare()` that compares two objects. If we have a data type that implements this interface, we can pass a `Comparator` to `sort()` (which passes it to `less()`) as in the example on the next page. The `Comparator` mechanism allows us to sort arrays of any type of object, using any total order that we wish to define for them. Using a `Comparator` instead of working with `Comparable` types better separates the definition of the type from the definition of what it means to compare two objects of that type. Indeed, there are typically many possible ways to compare objects, and the `Comparator` mechanism allows us to choose among them. For instance, to sort an array `a[]` of strings without regard to whether characters are uppercase or lowercase you can just call `Insertion.sort(a, String.CASE_INSENSITIVE_ORDER)` which makes use of the `CASE_INSENSITIVE_ORDER` comparator defined in Java's `String` class. As you can imagine, the precise rules for ordering strings are complicated and quite different for various natural languages, so Java has many `String` comparators.

Items with multiple keys. In typical applications, items have multiple instance variables that might need to serve as sort keys. In our transaction example, one client may need to sort the transaction list by customer (for example, to bring together all transactions involving each customer); another client might need to sort the list by amount (for example, to identify high-value transactions); and other clients might need to use other fields as sort keys. The `Comparator` mechanism is precisely what we need to allow this flexibility. We can define multiple comparators, as in the alternate implementation of `Transaction` shown on the bottom of the next page. With this definition, a client can sort an array of `Transaction` objects by time with the call

```
Insertion.sort(a, new Transaction.WhenOrder())
```

or by amount with the call

```
Insertion.sort(a, new Transaction.HowMuchOrder()).
```

The sort does each compare through a *callback* to the `compare()` method in `Transaction` that is specified by the client code. To avoid the cost of making a new `Comparator` object for each sort, we could use `public final` instance variables to define the comparators (as Java does for `CASE_INSENSITIVE_ORDER`).

```
public static void sort(Object[] a, Comparator c)
{
   int N = a.length;
   for (int i = 1; i < N; i++)
      for (int j = i; j > 0 && less(c, a[j], a[j-1]); j--)
         exch(a, j, j-1);
}

private static boolean less(Comparator c, Object v, Object w)
{  return c.compare(v, w) < 0;  }

private static void exch(Object[] a, int i, int j)
{  Object t = a[i]; a[i] = a[j]; a[j] = t; }
```

Insertion sorting with a Comparator

Priority queues with comparators. The same flexibility to use comparators is also useful for priority queues. Extending our standard implementation in ALGORITHM 2.6 to support comparators involves the following steps:

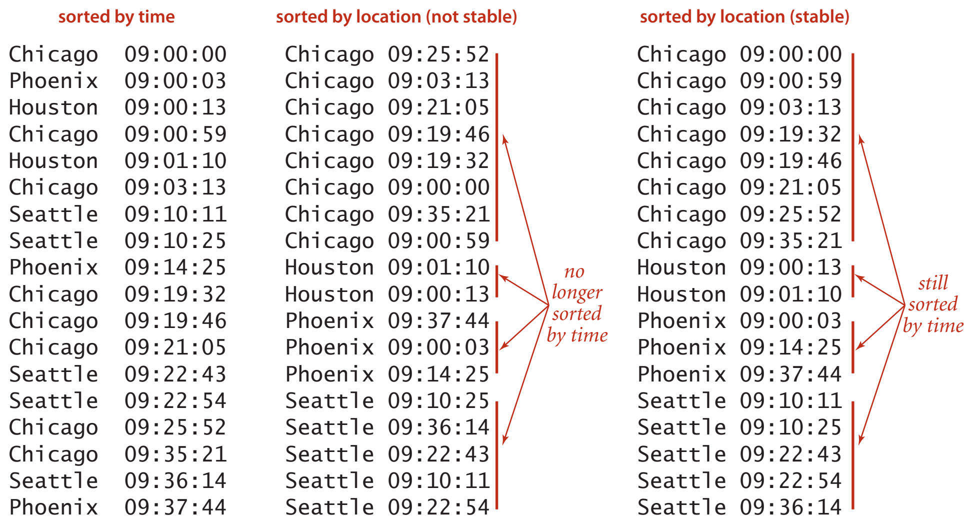
- Import `java.util.Comparator`.
- Add to `MaxPQ` an instance variable `comparator` and a constructor that takes a comparator as argument and initializes `comparator` to that value.
- Add code to `less()` that checks whether `comparator` is `null` (and uses it if it is not `null`).

For example, with these changes, you could build different priority queues with `Transaction` keys, using the time, place, or account number for the ordering. If you remove the `Key extends Comparable<Key>` phrase from `MinPQ`, you even can support keys with no natural order.

```
import java.util.Comparator;

public class Transaction
{
   ...
   private final String who;
   private final Date when;
   private final double amount;
   ...
   public static class WhoOrder implements Comparator<Transaction>
   {
      public int compare(Transaction v, Transaction w)
      {  return v.who.compareTo(w.when);  }
   }

   public static class WhenOrder implements Comparator<Transaction>
   {
      public int compare(Transaction v, Transaction w)
      {  return v.when.compareTo(w.when);  }
   }

   public static class HowMuchOrder implements Comparator<Transaction>
   {
      public int compare(Transaction v, Transaction w)
      {
         if (v.amount < w.amount) return -1;
         if (v.amount > w.amount) return +1;
         return 0;
      }
   }
}
```

Insertion sorting with a Comparator

Stability. A sorting method is *stable* if it preserves the relative order of equal keys in the array. This property is frequently important. For example, consider an internet commerce application where we have to process a large number of events that have locations and timestamps. To begin, suppose that we store events in an array as they arrive, so they are in order of the timestamp in the array. Now suppose that the application requires that the transactions be separated out by location for further processing. One easy way to do so is to sort the array by location. If the sort is unstable, the transactions for each city may *not* necessarily be in order by timestamp after the sort. Often, programmers who are unfamiliar with stability are surprised, when they first encounter the situation, by the way an unstable algorithm seems to scramble the data. Some of the sorting methods that we have considered in this chapter are stable (insertion sort and mergesort); many are not (selection sort, shellsort, quicksort, and heapsort). There are ways to trick any sort into stable behavior (see EXERCISE 2.5.18), but using a stable algorithm is generally preferable when stability is an essential requirement. It is easy to take stability for granted; actually, no practical method in common use achieves stability without using significant extra time or space (researchers have developed algorithms that do so, but applications programmers have judged them too complicated to be useful).



sorted by time		sorted by location (not stable)		sorted by location (stable)	
Chicago	09:00:00	Chicago	09:25:52	Chicago	09:00:00
Phoenix	09:00:03	Chicago	09:03:13	Chicago	09:00:59
Houston	09:00:13	Chicago	09:21:05	Chicago	09:03:13
Chicago	09:00:59	Chicago	09:19:46	Chicago	09:19:32
Houston	09:01:10	Chicago	09:19:32	Chicago	09:19:46
Chicago	09:03:13	Chicago	09:00:00	Chicago	09:21:05
Seattle	09:10:11	Chicago	09:35:21	Chicago	09:25:52
Seattle	09:10:25	Chicago	09:00:59	Chicago	09:35:21
Phoenix	09:14:25	Houston	09:01:10	Houston	09:00:13
Chicago	09:19:32	Houston	09:00:13	Houston	09:01:10
Chicago	09:19:46	Phoenix	09:37:44	Phoenix	09:00:03
Chicago	09:21:05	Phoenix	09:00:03	Phoenix	09:14:25
Seattle	09:22:43	Phoenix	09:14:25	Phoenix	09:37:44
Seattle	09:22:54	Seattle	09:10:25	Seattle	09:10:11
Chicago	09:25:52	Seattle	09:36:14	Seattle	09:10:25
Chicago	09:35:21	Seattle	09:22:43	Seattle	09:22:43
Seattle	09:36:14	Seattle	09:10:11	Seattle	09:22:54
Phoenix	09:37:44	Seattle	09:22:54	Seattle	09:36:14

Stability when sorting on a second key

Which sorting algorithm should I use? We have considered numerous sorting algorithms in this chapter, so this question is natural. Knowing which algorithm is best possible depends heavily on details of the application and implementation, but we have studied some general-purpose methods that can be nearly as effective as the best possible for a wide variety of applications.

The table at the bottom of this page is a general guide that summarizes the important characteristics of the sort algorithms that we have studied in this chapter. In all cases but shellsort (where the growth rate is only an estimate), insertion sort (where the growth rate depends on the order of the input keys), and both versions of quicksort (where the growth rate is probabilitic and may depend on the distribution of input key values), multiplying these growth rates by appropriate constants gives an effective way to predict running time. The constants involved are partly algorithm-dependent (for example, heapsort uses twice the number of compares as mergesort and both do many more array accesses than quicksort) but are primarily dependent on the implementation, the Java compiler, and your computer, which determine the number of machine instructions that are executed and the time that each requires. Most important, since they are constants, you can generally predict the running time for large N by running experiments for smaller N and extrapolating, using our standard doubling protocol.

algorithm	stable?	in place?	*order of growth to sort N items*		notes
			running time	extra space	
selection sort	no	yes	N^2	1	
insertion sort	yes	yes	between N and N^2	1	depends on order of items
shellsort	no	yes	$N \log N$? $N^{6/5}$?	1	
quicksort	no	yes	$N \log N$	$\lg N$	probabilistic guarantee
3-way quicksort	no	yes	between N and $N \log N$	$\lg N$	probabilistic, also depends on distribution of input keys
mergesort	yes	no	$N \log N$	N	
heapsort	no	yes	$N \log N$	1	

Performance characteristics of sorting algorithms

Property T. Quicksort is the fastest general-purpose sort.

Evidence: This hypothesis is supported by countless implementations of quicksort on countless computer systems since its invention decades ago. Generally, the reason that quicksort is fastest is that it has only a few instructions in its inner loop (and it does well with cache memories because it most often references data sequentially) so that its running time is ~c $N \lg N$ with the value of c smaller than the corresponding constants for other linearithmic sorts. With 3-way partitioning, quicksort becomes linear for certain key distributions likely to arise in practice, where other sorts are linearithmic.

Thus, in most practical situations, quicksort is the method of choice. Still, given the broad reach of sorting and the broad variety of computers and systems, a flat statement like this is difficult to justify. For example, we have already seen one notable exception: if stability is important and space is available, mergesort might be best. We will see other exceptions in CHAPTER 5. With tools like `SortCompare` and a considerable amount of time and effort, you can do a more detailed study of comparative performance of these algorithms and the refinements that we have discussed for your computer, as discussed in several exercises at the end of this section. Perhaps the best way to interpret PROPERTY T is as saying that you certainly should seriously consider using quicksort in any sort application where running time is important.

Sorting primitive types. In some performance-critical applications, the focus may be on sorting numbers, so it is reasonable to avoid the costs of using references and sort primitive types instead. For example, consider the difference between sorting an array of `double` values and sorting an array of `Double` values. In the former case, we exchange the numbers themselves and put them in order in the array; in the latter, we exchange references to `Double` objects, which contain the numbers. If we are doing nothing more than sorting a huge array of numbers, we avoid paying the cost of storing an equal number of references plus the extra cost of accessing the numbers through the references, not to mention the cost of invoking `compareTo()` and `less()` methods. We can develop efficient versions of our sort codes for such purposes by replacing `Comparable` with the primitive type name, and redefining `less()` or just replacing calls to `less()` with code like `a[i] < a[j]` (see EXERCISE 2.1.26).

Java system sort. As an example of applying the information given in the table on page 342, consider Java's primary system sort method, `java.util.Arrays.sort()`. With overloading of argument types, this name actually represents a collection of methods:

- A different method for each primitive type
- A method for data types that implement `Comparable`
- A method that uses a `Comparator`

Java's systems programmers have chosen to use quicksort (with 3-way partitioning) to implement the primitive-type methods, and mergesort for reference-type methods. The primary practical implications of these choices are, as just discussed, to trade speed and memory usage (for primitive types) for stability (for reference types).

THE ALGORITHMS AND IDEAS that we have been considering are an essential part of many modern systems, including Java. When developing Java programs to address an application, you are likely to find that Java's `Arrays.sort()` implementations (perhaps supplemented by your own implementation(s) of `compareTo()` and/or `compare()`) will meet your needs, because you will be using 3-way quicksort or mergesort, both proven classic algorithms.

In this book, we generally will use our own `Quick.sort()` (usually) or `Merge.sort()` (when stability is important and space is not) in sort clients. You may feel free to use `Arrays.sort()` unless you have a good reason to use another specific method.

Reductions The idea that we can use sorting algorithms to solve other problems is an example of a basic technique in algorithm design known as *reduction*. We consider reduction in detail in CHAPTER 6 because of its importance in the theory of algorithms—in the meantime, we will consider several practical examples. A *reduction* is a situation where an algorithm developed for one problem is used to solve another. Applications programmers are quite used to the concept of reduction (whether or not it is explicitly articulated)—every time you make use of a method that solves problem B in order to solve problem A, you are doing a reduction from A to B. Indeed, one goal in implementing algorithms is to facilitate reductions by making the algorithms useful for as wide a variety as possible of applications. We begin with a few elementary examples for sorting. Many of these take the form of algorithmic puzzles where a quadratic brute-force algorithm is immediate. It is often the case that sorting the data first makes it easy to finish solving the problem in linear additional time, thus reducing the total cost from quadratic to linearithmic.

Duplicates. Are there any duplicate keys in an array of `Comparable` objects? How many distinct keys are there? Which value appears most frequently? For small arrays, these kinds of questions are easy to answer with a quadratic algorithm that compares each array entry with each other array entry. For large arrays, using a quadratic algorithm is not feasible. With sorting, you can answer these questions in linearithmic time:

first sort the array, then make a pass through the sorted array, taking note of duplicate keys that appear consecutively in the ordered array. For example, the code fragment at right counts the distinct keys in an array. With simple modifications to this code, you can answer the questions above and perform tasks such as printing all the distinct values, all the values that are duplicated, and so forth, even for huge arrays.

```
Quick.sort(a);
int count = 1; // Assume a.length > 0.
for (int i = 1; i < a.length; i++)
   if (a[i].compareTo(a[i-1]) != 0)
      count++;
```

Counting the distinct keys in a[]

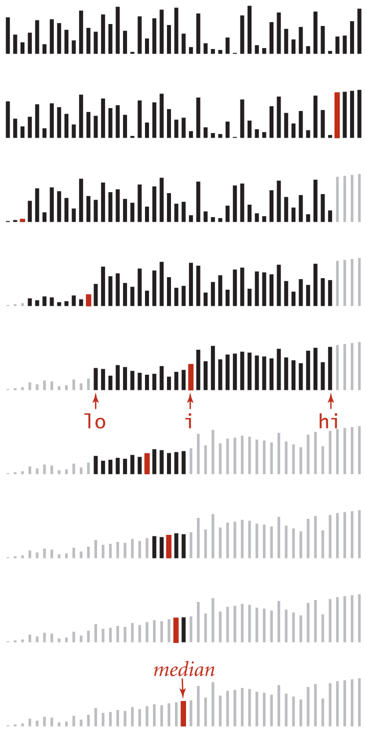
Rankings. A *permutation* (or *ranking*) is an array of N integers where each of the integers between 0 and N-1 appears exactly once. The *Kendall tau distance* between two rankings is the number of pairs that are in different order in the two rankings. For example, the Kendall tau distance between `0 3 1 6 2 5 4` and `1 0 3 6 4 2 5` is four because the pairs `0-1`, `3-1`, `2-4`, `5-4` are in different relative order in the two rankings, but all other pairs are in the same relative order. This statistic is widely used: in sociology to study social choice and voting theory, in molecular biology to compare genes using expression profiles, and in ranking search engine results on the web, among many other applications. The Kendall tau distance between a permutation and the identity permutation (where each entry is equal to its index) is the number of inversions in the permutation, and a quadratic algorithm based on insertion sort is not difficult to devise (recall PROPOSITION C in SECTION 2.1). Efficiently computing the Kendall tau distance is an interesting exercise for a programmer (or a student!) who is familiar with the classical sorting algorithms that we have studied (see EXERCISE 2.5.19).

Priority-queue reductions. In SECTION 2.4, we considered two examples of problems that reduce to a sequence of operations on priority queues. `TopM`, on page 311, finds the M items in an input stream with the highest keys. `Multiway`, on page 322, merges M sorted input streams together to make a sorted output stream. Both of these problems are easily addressed with a priority queue of size M.

Median and order statistics. An important application related to sorting but for which a full sort is not required is the operation of finding the *median* of a collection of keys (the value with the property that half the keys are no larger and half the keys are no smaller). This operation is a common computation in statistics and in various other data-processing applications. Finding the median is a special case of *selection*: finding the kth smallest of a collection of numbers. Selection has many applications in the processing of experimental and other data. The use of the median and other *order* *statistics* to divide an array into smaller groups is common. Often, only a small part of a large array is to be saved for further processing; in such cases, a program that can select, say, the top 10 percent of the items of the array might be more appropriate than a full sort. Our `TopM` application of SECTION 2.4 solves this problem for an unbounded input stream, using a priority queue. An effective alternative to `TopM` when you have the items in an array is to just sort it: after the call `Quick.sort(a)` the k smallest values in the array are in the first k array positions for all k less than the array length. But this approach involves a sort, so the running time is linearithmic. Can we do better? Finding the k smallest values in an array is easy when k is very small or very large, but more challenging when k is a constant fraction of the array size, such as finding the median ($k = N/2$). You might be surprised to learn that it is possible to solve this problem in *linear* time, as in the `select()` method above (this implementation requires a client cast; for the more pedantic code needed to avoid this requirement, see the booksite). To do the job, `select()` maintains the variables `lo` and `hi` to delimit the subarray that contains the index `k` of the item to be selected and uses quicksort partitioning to shrink the size of the subarray. Recall that `partition()` rearranges an array `a[lo]` through `a[hi]` and returns an integer `j` such that `a[lo]` through `a[j-1]` are less than or equal to `a[j]`, and `a[j+1]` through `a[hi]` are greater than or equal to `a[j]`. Now, if `k` is equal to `j`, then we are done. Otherwise, if `k < j`, then we need to continue working in the left subarray (by changing the value of `hi` to `j-1`); if `k > j`, then we need to continue working in the right subarray (by changing `lo` to `j+1`). The loop maintains the invariant that no entry to the left of `lo` is larger and no entry to the right of `hi` is smaller than any element within `a[lo..hi]`. After partitioning, we preserve this invariant and shrink the interval until it consists just of

```
public static Comparable
select(Comparable[] a, int k)
{
   StdRandom.shuffle(a);
   int lo = 0, hi = a.length - 1;
   while (hi > lo)
   {
      int j = partition(a, lo, hi);
      if      (j == k)  return a[k];
      else if (j > k)  hi = j - 1;
      else if (j < k)  lo = j + 1;
   }
   return a[k];
}
```

Selecting the k smallest elements in a[]



Partitioning to find the median

`k`. Upon termination, `a[k]` contains the (k+1)st smallest entry, `a[0]` through `a[k-1]` are all smaller than (or equal to) `a[k]`, and `a[k+1]` through the end of the array are all greater than (or equal to) `a[k]`. To gain some insight into why this is a linear-time algorithm, suppose that partitioning divides the array exactly in half each time. Then the number of compares is $N + N/2 + N/4 + N/8 + \ldots$, terminating when the kth smallest item is found. This sum is less than $2N$. As with quicksort, it takes a bit of math to find the true bound, which is a bit higher. Also as with quicksort, the analysis depends on partitioning on a random item, so that the guarantee is probabilistic.

Proposition U. Partitioning-based selection is a linear-time algorithm, on average.

Proof: An analysis similar to, but significantly more complex than, the proof of PROPOSITION K for quicksort leads to the result that the average number of compares is $\sim 2N + 2k\ln(N/k) + 2(N - k)\ln(N/(N - k))$, which is linear for any allowed value of k. For example, this formula says that finding the median ($k = N/2$) requires $\sim (2 + 2\ln 2)N$ compares, on the average. Note that the worst case is quadratic but randomization protects against that possibility, as with quicksort.

Designing a selection algorithm that is guaranteed to use a linear number of compares in the *worst case* is a classic result in computational complexity, but it has not yet led to a useful practical algorithm.

A brief survey of sorting applications Direct applications of sorting are familiar, ubiquitous, and far too numerous for us to list them all. You sort your music by song title or by artist name, your email or phone calls by time or origin, and your photos by date. Universities sort student accounts by name or ID. Credit card companies sort millions or even billions of transactions by date or amount. Scientists sort not only experimental data by time or other identifier but also to enable detailed simulations of the natural world, from the motion of particles or heavenly bodies to the structure of materials to social interations and relationships. Indeed, it is difficult to identify a computational application that does *not* involve sorting! To elaborate upon this point, we describe in this section examples of applications that are more complicated than the reductions just considered, including several that we will examine in detail later in this book.

Commercial computing. The world is awash in information. Government organizations, financial institutions, and commercial enterprises organize much of this information by sorting it. Whether the information is accounts to be sorted by name or number, transactions to be sorted by date or amount, mail to be sorted by postal code or address, files to be sorted by name or date, or whatever, processing such data is sure to involve a sorting algorithm somewhere along the way. Typically, such information is organized in huge databases, sorted by multiple keys for efficient search. An effective strategy that is widely used is to collect new information, add it to the database, sort it on the keys of interest, and merge the sorted result for each key into the existing database. The methods that we have discussed have been used effectively since the early days of computing to build a huge infrastructure of sorted data and methods for processing it that serve as the basis for all of this commercial activity. Arrays having millions or even billions of entries are routinely processed today—without linearithmic sorting algorithms, such arrays could not be sorted, making such processing extremely difficult or impossible.

Search for information. Keeping data in sorted order makes it possible to efficiently search through it using the classic *binary search* algorithm (see CHAPTER 1). You will also see that the same scheme makes it easy to quickly handle many other kinds of queries. How many items are smaller than a given item? Which items fall within a given range? In CHAPTER 3, we consider such questions. We also consider in detail various extensions to sorting and binary search that allow us to intermix such queries with operations that insert and remove objects from the set, still guaranteeing logarithmic performance for all operations.

Operations research. The field of *operations research* (OR) develops and applies mathematical models for problem-solving and decision-making. We will see several examples in this book of relationships between OR and the study of algorithms, beginning here with the use of sorting in a classic OR problem known as *scheduling*. Suppose that we have N jobs to complete, where job j requires t_j seconds of processing time. We need to complete all of the jobs but want to maximize customer satisfaction by minimizing the average completion time of the jobs. The *shortest processing time first* rule, where we schedule the jobs in increasing order of processing time, is known to accomplish this goal. Therefore we can sort the jobs by processing time or put them on a minimum-oriented priority queue. With various other constraints and restrictions, we get various other scheduling problems, which frequently arise in industrial applications and are well-studied. As another example, consider the *load-balancing problem*, where we have M identical processors and N jobs to complete, and our goal is to schedule all of the jobs on the processors so that the time at which the last job completes is as early as possible. This specific problem is *NP*-hard (see CHAPTER 6) so we do not expect to find a practical way to compute an optimal schedule. One method that is known to produce a good schedule is the *longest processing time first* rule, where we consider the jobs in descending order of processing time, assigning each job to the processor that becomes available first. To implement this algorithm, we first sort the jobs in reverse order. Then we maintain a priority queue of M processors, where the priority is the sum of the processing times of its jobs. At each step, we delete the processor with the minimum priority, add the next job to the processor, and reinsert that processor into the priority queue.

Event-driven simulation. Many scientific applications involve simulation, where the point of the computation is to model some aspect of the real world in order to be able to better understand it. Before the advent of computing, scientists had little choice but to build mathematical models for this purpose; such models are now well-complemented by computational models. Doing such simulations efficiently can be challenging, and use of appropriate algorithms certainly can make the difference between being able to complete the simulation in a reasonable amount of time and being stuck with the choice of accepting inaccurate results or waiting for the simulation to do the computation necessary to get accurate results. We will consider in CHAPTER 6 a detailed example that illustrates this point.

Numerical computations. Scientific computing is often concerned with *accuracy* (how close are we to the true answer?). Accuracy is extremely important when we are performing millions of computations with estimated values such as the floating-point representation of real numbers that we commonly use on computers. Some numerical algorithms use priority queues and sorting to control accuracy in calculations. For example, one way to do numerical integration (quadrature), where the goal is to estimate the area under a curve, is to maintain a priority queue with accuracy estimates for a set of subintervals that comprise the whole interval. The process is to remove the least accurate subinterval, split it in half (thus achieving better accuracy for the two halves), and put the two halves back onto the priority queue, continuing until a desired tolerance is reached.

Combinatorial search. A classic paradigm in artificial intelligence and in coping with intractable problems is to define a set of *configurations* with well-defined *moves* from one configuration to the next and a priority associated with each move. Also defined is a *start* configuration and a *goal* configuration (which corresponds to having solved the problem). The well-known *A* algorithm* is a problem-solving process where we put the start configuration on the priority queue, then do the following until reaching the goal: remove the highest-priority configuration and add to the queue all configurations that can be reached from that with one move (excluding the one just removed). As with event-driven simulation, this process is tailor-made for priority queues. It reduces solving the problem to defining an effective priority function. See EXERCISE 2.5.32 for an example.

BEYOND SUCH DIRECT APPLICATIONS (and we have only indicated a small fraction of those), sorting and priority queues are an essential abstraction in algorithm design, so they will surface frequently throughout this book. We next list some examples of applications from later in the book. All of these applications depend upon the efficient implementations of sorting algorithms and the priority-queue data type that we have considered in this chapter.

Prim's algorithm and Dijkstra's algorithm are classical algorithms from CHAPTER 4. That chapter is about algorithms that process *graphs*, a fundamental model for *items* and *edges* that connect pairs of items. The basis for these and several other algorithms is *graph search*, where we proceed from item to item along edges. Priority queues play a fundamental role in organizing graph searches, enabling efficient algorithms.

Kruskal's algorithm is another classic algorithm for graphs whose edges have weights that depends upon processing the edges in order of their weight. Its running time is dominated by the cost of the sort.

Huffman compression is a classic *data compression* algorithm that depends upon processing a set of items with integer weights by combining the two smallest to produce a new one whose weight is the sum of its two constituents. Implementing this operation is immediate, using a priority queue. Several other data-compression schemes are based upon sorting.

String-processing algorithms, which are of critical importance in modern applications in cryptology and in genomics, are often based on sorting (generally using one of the specialized string sorts discussed in CHAPTER 5). For example, we will discuss in CHAPTER 6 algorithms for finding the *longest repeated substring* in a given string that is based on first sorting suffixes of the strings.

Q & A

Q. Is there a priority-queue data type in the Java library?

A. Yes, see `java.util.PriorityQueue`.

EXERCISES

2.5.1 Consider the following implementation of the `compareTo()` method for `String`. How does the third line help with efficiency?

```
public int compareTo(String that)
{
   if (this == that) return 0;  // this line
   int n = Math.min(this.length(), that.length());
   for (int i = 0; i < n; i++)
   {
      if      (this.charAt(i) < that.charAt(i)) return -1;
      else if (this.charAt(i) > that.charAt(i)) return +1;
   }
   return this.length() - that.length();
}
```

2.5.2 Write a program that reads a list of words from standard input and prints all two-word compound words in the list. For example, if `after`, `thought`, and `afterthought` are in the list, then `afterthought` is a compound word.

2.5.3 Criticize the following implementation of a class intended to represent account balances. Why is `compareTo()` a flawed implementation of the `Comparable` interface?

```
public class Balance implements Comparable<Balance>
{
   ...
   private double amount;
   public int compareTo(Balance that)
   {
      if (this.amount < that.amount - 0.005) return -1;
      if (this.amount > that.amount + 0.005) return +1;
      return 0;
   }
   ...
}
```

Describe a way to fix this problem.

2.5.4 Implement a method `String[] dedup(String[] a)` that returns the objects in `a[]` in sorted order, with duplicates removed.

2.5.5 Explain why selection sort is not stable.

EXERCISES *(continued)*

2.5.6 Implement a recursive version of `select()`.

2.5.7 About how many compares are required, on the average, to find the smallest of N items using `select()`?

2.5.8 Write a program `Frequency` that reads strings from standard input and prints the number of times each string occurs, in descending order of frequency.

2.5.9 Develop a data type that allows you to write a client that can sort a file such as the one shown at right.

2.5.10 Create a data type `Version` that represents a software version number, such as `115.1.1`, `115.10.1`, `115.10.2`. Implement the `Comparable` interface so that `115.1.1` is less than `115.10.1`, and so forth.

2.5.11 One way to describe the result of a sorting algorithm is to specify a permutation `p[]` of the numbers `0` to `a.length-1`, such that `p[i]` specifies where the key originally in `a[i]` ends up. Give the permutations that describe the results of insertion sort, selection sort, shellsort, mergesort, quicksort, and heapsort for an array of seven equal keys.

input (DJI volumes for each day)

```
  1-Oct-28      3500000
  2-Oct-28      3850000
  3-Oct-28      4060000
  4-Oct-28      4330000
  5-Oct-28      4360000
...
 30-Dec-99    554680000
 31-Dec-99    374049984
  3-Jan-00    931800000
  4-Jan-00   1009000000
  5-Jan-00   1085500032
  ...
```

output

```
19-Aug-40  130000
26-Aug-40  160000
24-Jul-40  200000
10-Aug-42  210000
23-Jun-42  210000
...
23-Jul-02  2441019904
17-Jul-02  2566500096
15-Jul-02  2574799872
19-Jul-02  2654099968
24-Jul-02  2775559936
```

CREATIVE PROBLEMS

2.5.12 *Scheduling.* Write a program `SPT.java` that reads job names and processing times from standard input and prints a schedule that minimizes average completion time using the shortest processing time first rule, as described on page 349.

2.5.13 *Load balancing.* Write a program `LPT.java` that takes an integer `M` as a command-line argument, reads job names and processing times from standard input and prints a schedule assigning the jobs to `M` processors that approximately minimizes the time when the last job completes using the longest processing time first rule, as described on page 349.

2.5.14 *Sort by reverse domain.* Write a data type `Domain` that represents domain names, including an appropriate `compareTo()` method where the natural order is in order of the *reverse* domain name. For example, the reverse domain of `cs.princeton.edu` is `edu.princeton.cs`. This is useful for web log analysis. *Hint*: Use `s.split("\\.")` to split the string `s` into tokens, delimited by dots. Write a client that reads domain names from standard input and prints the reverse domains in sorted order.

2.5.15 *Spam campaign.* To initiate an illegal spam campaign, you have a list of email addresses from various domains (the part of the email address that follows the @ symbol). To better forge the return addresses, you want to send the email from another user at the same domain. For example, you might want to forge an email from `wayne@princeton.edu` to `rs@princeton.edu`. How would you process the email list to make this an efficient task?

2.5.16 *Unbiased election.* In order to thwart bias against candidates whose names appear toward the end of the alphabet, California sorted the candidates appearing on its 2003 gubernatorial ballot by using the following order of characters:

```
R W Q O J M V A H B S G Z X N T C I E K U P D Y F L
```

Create a data type where this is the natural order and write a client `California` with a single static method `main()` that sorts strings according to this ordering. Assume that each string is composed solely of uppercase letters.

2.5.17 *Check stability.* Extend your `check()` method from Exercise 2.1.16 to call `sort()` for a given array and return `true` if `sort()` sorts the array in order *in a stable manner*, `false` otherwise. Do not assume that `sort()` is restricted to move data only with `exch()`.

CREATIVE PROBLEMS *(continued)*

2.5.18 *Force stability.* Write a wrapper method that makes any sort stable by creating a new key type that allows you to append each key's index to the key, call `sort()`, then restore the original key after the sort.

2.5.19 *Kendall tau distance.* Write a program `KendallTau.java` that computes the Kendall tau distance between two permutations in linearithmic time.

2.5.20 *Idle time.* Suppose that a parallel machine processes N jobs. Write a program that, given the list of job start and finish times, finds the largest interval where the machine is idle and the largest interval where the machine is *not* idle.

2.5.21 *Multidimensional sort.* Write a `Vector` data type for use in having the sorting methods sort multidimensional vectors of d integers, putting the vectors in order by first component, those with equal first component in order by second component, those with equal first and second components in order by third component, and so forth.

2.5.22 *Stock market trading.* Investors place buy and sell orders for a particular stock on an electronic exchange, specifying a maximum buy or minimum sell price that they are willing to pay, and how many shares they wish to trade at that price. Develop a program that uses priority queues to match up buyers and sellers and test it through simulation. Maintain two priority queues, one for buyers and one for sellers, executing trades whenever a new order can be matched with an existing order or orders.

2.5.23 *Sampling for selection.* Investigate the idea of using sampling to improve selection. *Hint*: Using the median may not always be helpful.

2.5.24 *Stable priority queue.* Develop a *stable* priority-queue implementation (which returns duplicate keys in the same order in which they were inserted).

2.5.25 *Points in the plane.* Write three `static` comparators for the `Point2D` data type of page 77, one that compares points by their x coordinate, one that compares them by their y coordinate, and one that compares them by their distance from the origin. Write two non-`static` comparators for the `Point2D` data type, one that compares them by their distance to a specified point and one that compares them by their polar angle with respect to a specified point.

2.5.26 *Simple polygon.* Given N points in the plane, draw a simple polygon with N

points as vertices. *Hint*: Find the point p with the smallest y coordinate, breaking ties with the smallest x coordinate. Connect the points in increasing order of the polar angle they make with p.

2.5.27 *Sorting parallel arrays.* When sorting parallel arrays, it is useful to have a version of a sorting routine that returns a permutation, say `index[]`, of the indices in sorted order. Add a method `indirectSort()` to `Insertion` that takes an array of `Comparable` objects `a[]` as argument, but instead of rearranging the entries of `a[]` returns an integer array `index[]` so that `a[index[0]]` through `a[index[N-1]]` are the items in ascending order.

2.5.28 *Sort files by name.* Write a program `FileSorter` that takes the name of a directory as a command-line argument and prints out all of the files in the current directory, sorted by file name. *Hint*: Use the `File` data type.

2.5.29 *Sort files by size and date of last modification.* Write comparators for the type `File` to order by increasing/decreasing order of file size, ascending/descending order of file name, and ascending/descending order of last modification date. Use these comparators in a program `LS` that takes a command-line argument and lists the files in the current directory according to a specified order, e.g., `"-t"` to sort by timestamp. Support multiple flags to break ties. Be sure to use a stable sort.

2.5.30 *Boerner's theorem.* True or false: If you sort each column of a matrix, then sort each row, the columns are still sorted. Justify your answer.

EXPERIMENTS

2.5.31 *Duplicates.* Write a client that takes integers `M`, `N`, and `T` as command-line arguments, then uses the code given in the text to perform T trials of the following experiment: Generate N random `int` values between 0 and $M - 1$ and count the number of duplicates. Run your program for $T = 10$ and $N = 10^3, 10^4, 10^5$, and 10^6, with $M = N/2$, and N, and $2N$. Probability theory says that the number of duplicates should be about $(1 - e^{-\alpha})$ where $\alpha = N/M$—print a table to help you confirm that your experiments validate that formula.

2.5.32 *8 puzzle.* The 8 puzzle is a game popularized by S. Loyd in the 1870s. It is played on a 3-by-3 grid with 8 tiles labeled 1 through 8 and a blank square. Your goal is to rearrange the tiles so that they are in order. You are permitted to slide one of the available tiles horizontally or vertically (but not diagonally) into the blank square. Write a program that solves the puzzle using the *A* algorithm*. Start by using as priority the sum of the number of moves made to get to this board position plus the number of tiles in the wrong position. (Note that the number of moves you must make from a given board position is at least as big as the number of tiles in the wrong place.) Investigate substituting other functions for the number of tiles in the wrong position, such as the sum of the Manhattan distance from each tile to its correct position, or the sums of the squares of these distances.

2.5.33 *Random transactions.* Develop a generator that takes an argument N, generates N random `Transaction` objects (see EXERCISES 2.1.21 and 2.1.22), using assumptions about the transactions that you can defend. Then compare the performance of shellsort, mergesort, quicksort, and heapsort for sorting N transactions, for N=10^3, 10^4, 10^5, and 10^6.