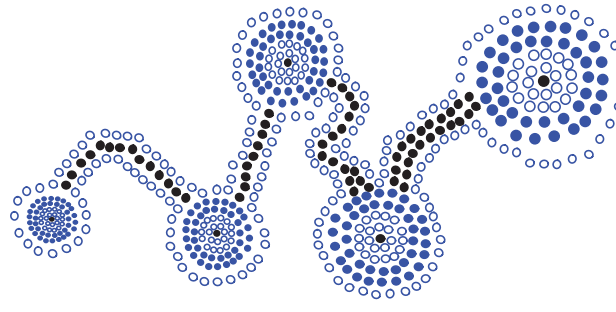
# Chapter 11 Sorting, Sets, and Selection



## Contents

# 11.1 Merge-Sort

In this section, we present a sorting technique, called ***merge-sort***, which can be described in a simple and compact way using recursion.

## 11.1.1 Divide-and-Conquer

Merge-sort is based on an algorithmic design pattern called ***divide-and-conquer***. The divide-and-conquer pattern consists of the following three steps:

1. ***Divide:*** If the input size is smaller than a certain threshold (say, one or two elements), solve the problem directly using a straightforward method and return the solution obtained. Otherwise, divide the input data into two or more disjoint subsets.
2. ***Recur:*** Recursively solve the subproblems associated with the subsets.
3. ***Conquer:*** Take the solutions to the subproblems and "merge" them into a solution to the original problem.

### Using Divide-and-Conquer for Sorting

Recall that in a sorting problem we are given a sequence of $n$ objects, stored in a linked list or an array, together with some comparator defining a total order on these objects, and we are asked to produce an ordered representation of these objects. To allow for sorting of either representation, we describe our sorting algorithm at a high level for sequences and explain the details needed to implement it for linked lists and arrays. To sort a sequence $S$ with $n$ elements using the three divide-and-conquer steps, the merge-sort algorithm proceeds as follows:

1. ***Divide:*** If $S$ has zero or one element, return $S$ immediately; it is already sorted. Otherwise ($S$ has at least two elements), remove all the elements from $S$ and put them into two sequences, $S_1$ and $S_2$, each containing about half of the elements of $S$; that is, $S_1$ contains the first $\lceil n/2 \rceil$ elements of $S$, and $S_2$ contains the remaining $\lfloor n/2 \rfloor$ elements.
2. ***Recur:*** Recursively sort sequences $S_1$ and $S_2$.
3. ***Conquer:*** Put back the elements into $S$ by merging the sorted sequences $S_1$ and $S_2$ into a sorted sequence.

In reference to the divide step, we recall that the notation $\lceil x \rceil$ indicates the ***ceiling*** of $x$, that is, the smallest integer $m$, such that $x \leq m$. Similarly, the notation $\lfloor x \rfloor$ indicates the ***floor*** of $x$, that is, the largest integer $k$, such that $k \leq x$.

We can visualize an execution of the merge-sort algorithm by means of a binary tree $T$, called the ***merge-sort tree***. Each node of $T$ represents a recursive invocation (or call) of the merge-sort algorithm. We associate the sequence $S$ that is processed by the invocation associated with $v$, with each node $v$ of $T$. The children of node $v$ are associated with the recursive calls that process the subsequences $S_1$ and $S_2$ of $S$. The external nodes of $T$ are associated with individual elements of $S$, corresponding to instances of the algorithm that make no recursive calls.

Figure 11.1 summarizes an execution of the merge-sort algorithm by showing the input and output sequences processed at each node of the merge-sort tree. The step-by-step evolution of the merge-sort tree is shown in Figures 11.2 through 11.4.

This algorithm visualization in terms of the merge-sort tree helps us analyze the running time of the merge-sort algorithm. In particular, since the size of the input sequence roughly halves at each recursive call of merge-sort, the height of the merge-sort tree is about $\log n$ (recall that the base of log is 2 if omitted).

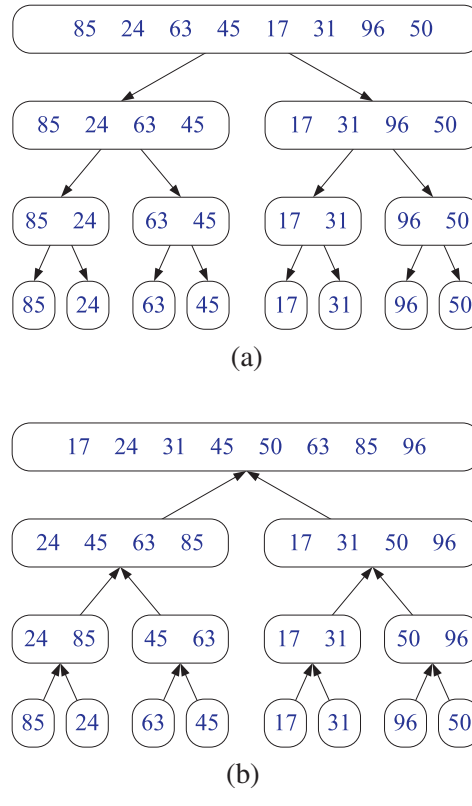



**Figure 11.1:** Merge-sort tree $T$ for an execution of the merge-sort algorithm on a sequence with eight elements: (a) input sequences processed at each node of $T$; (b) output sequences generated at each node of $T$.

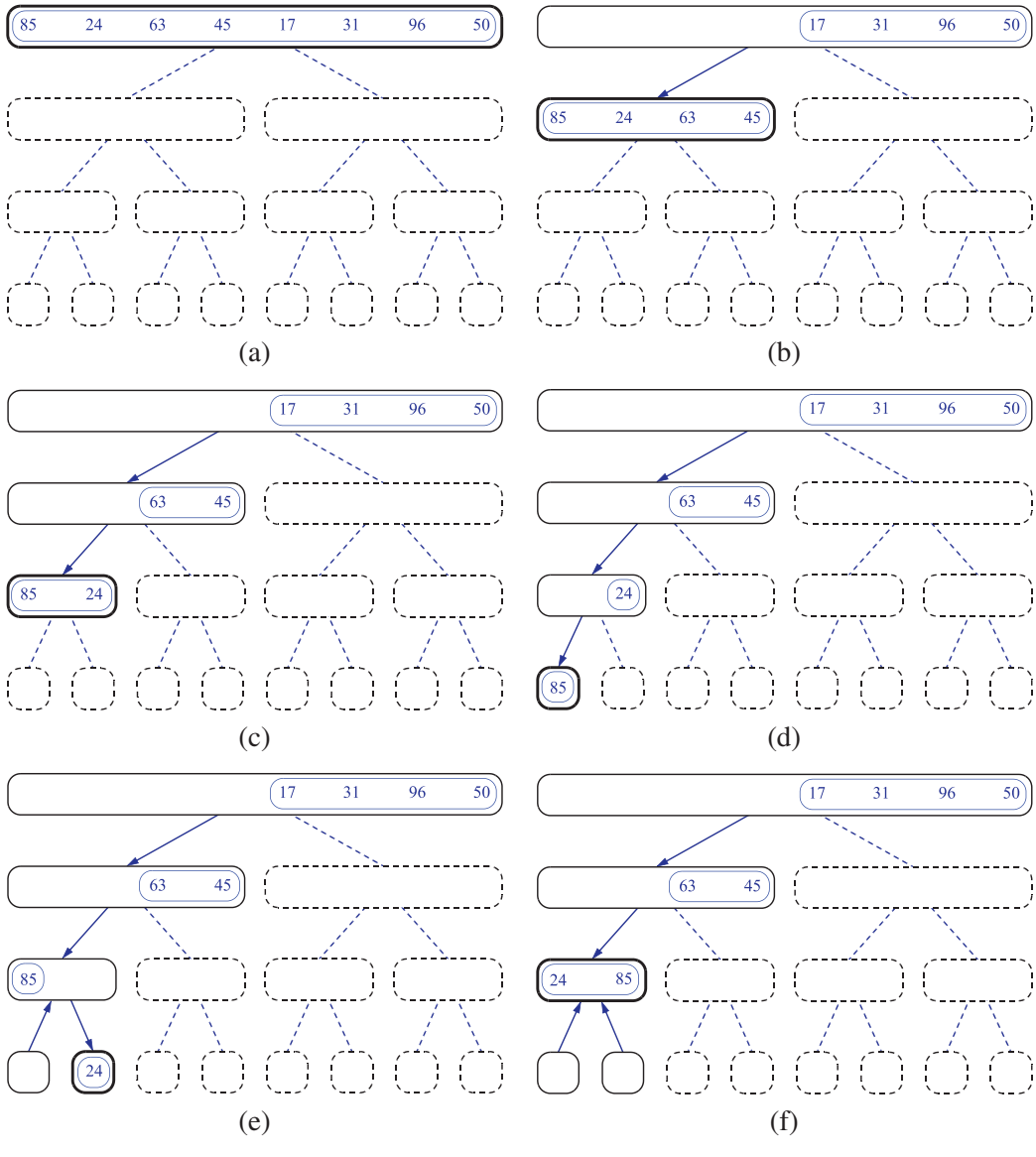



**Figure 11.2:** Visualization of an execution of merge-sort. Each node of the tree represents a recursive call of merge-sort. The nodes drawn with dashed lines represent calls that have not been made yet. The node drawn with thick lines represents the current call. The empty nodes drawn with thin lines represent completed calls. The remaining nodes (drawn with thin lines and not empty) represent calls that are waiting for a child invocation to return. (Continues in Figure 11.3.)

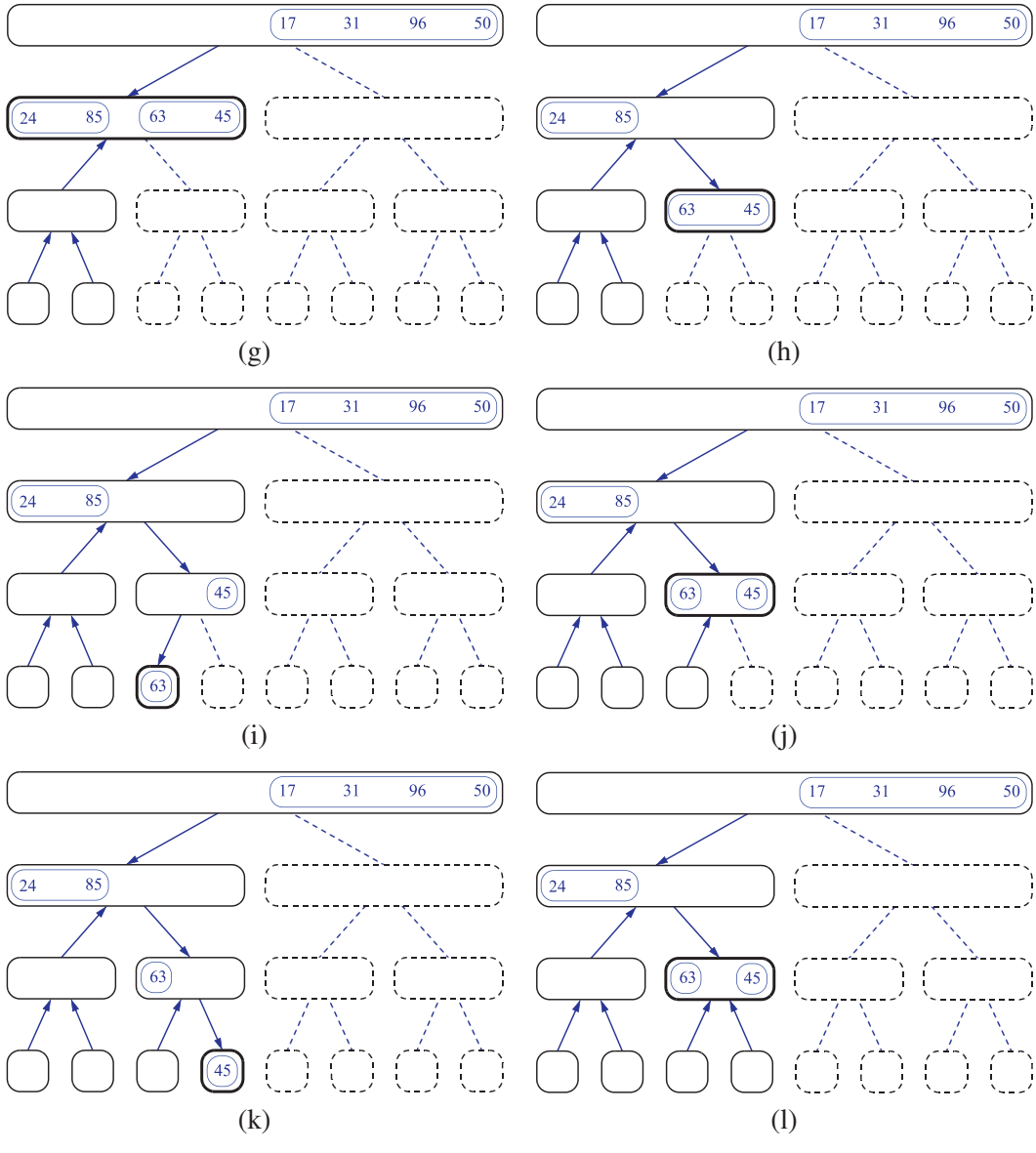



**Figure 11.3:** Visualization of an execution of merge-sort. (Continues in Figure 11.4.)

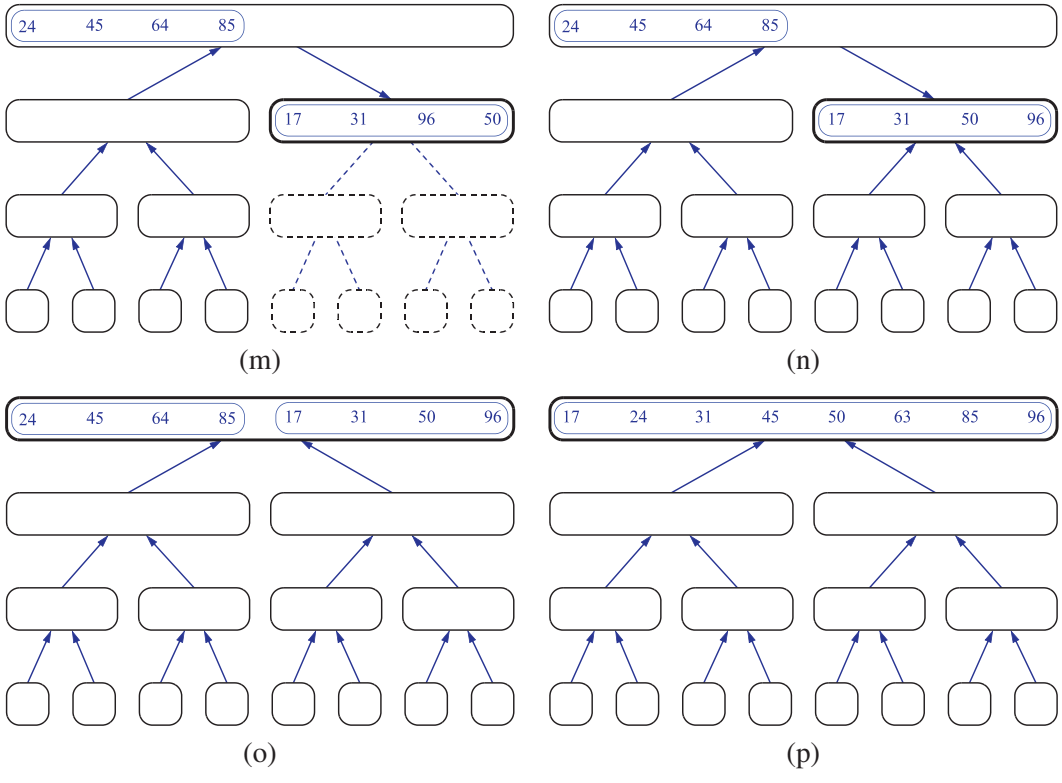



**Figure 11.4:** Visualization of an execution of merge-sort. Several invocations are omitted between (l) and (m) and between (m) and (n). Note the conquer step performed in step (p). (Continued from Figure 11.3.)

**Proposition 11.1:** *The merge-sort tree associated with an execution of merge-sort on a sequence of size $n$ has height $\lceil \log n \rceil$.*

We leave the justification of Proposition 11.1 as a simple exercise (R-11.4). We use this proposition to analyze the running time of the merge-sort algorithm.

Having given an overview of merge-sort and an illustration of how it works, let us consider each of the steps of this divide-and-conquer algorithm in more detail. The divide and recur steps of the merge-sort algorithm are simple; dividing a sequence of size $n$ involves separating it at the element with index $\lceil n/2 \rceil$, and the recursive calls simply involve passing these smaller sequences as parameters. The difficult step is the conquer step, which merges two sorted sequences into a single sorted sequence. Thus, before we present our analysis of merge-sort, we need to say more about how this is done.

### 11.1.2 Merging Arrays and Lists

To merge two sorted sequences, it is helpful to know if they are implemented as arrays or lists. We begin with the array implementation, which we show in Code Fragment 11.1. We illustrate a step in the merge of two sorted arrays in Figure 11.5.

**Algorithm** merge($S_1, S_2, S$):
*Input:* Sorted sequences $S_1$ and $S_2$ and an empty sequence $S$, all of which are implemented as arrays
*Output:* Sorted sequence $S$ containing the elements from $S_1$ and $S_2$

```
i ← j ← 0
while i < S1.size() and j < S2.size() do
  if S1[i] ≤ S2[j] then
    S.insertBack(S1[i]) {copy ith element of S1 to end of S}
    i ← i + 1
  else
    S.insertBack(S2[j]) {copy jth element of S2 to end of S}
    j ← j + 1
while i < S1.size() do {copy the remaining elements of S1 to S}
  S.insertBack(S1[i])
  i ← i + 1
while j < S2.size() do {copy the remaining elements of S2 to S}
  S.insertBack(S2[j])
  j ← j + 1
```

**Code Fragment 11.1:** Algorithm for merging two sorted array-based sequences.

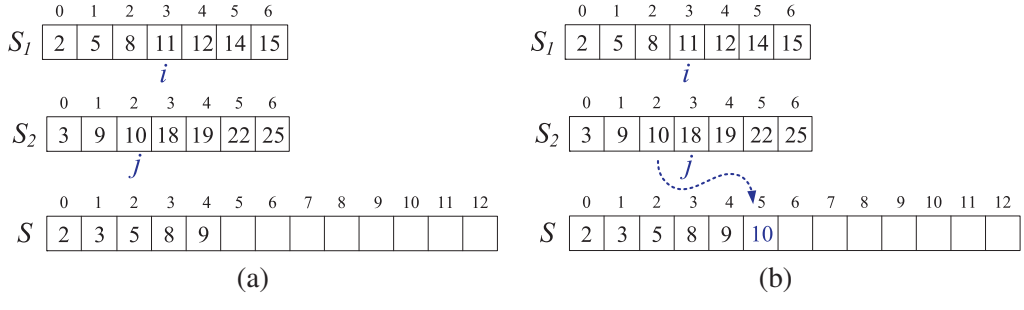



**Figure 11.5:** A step in the merge of two sorted arrays: (a) before the copy step; (b) after the copy step.

### Merging Two Sorted Lists

In Code Fragment 11.2, we give a list-based version of algorithm merge, for merging two sorted sequences, $S_1$ and $S_2$, implemented as linked lists. The main idea is to iteratively remove the smallest element from the front of one of the two lists and add it to the end of the output sequence, $S$, until one of the two input lists is empty, at which point we copy the remainder of the other list to $S$. We show an example execution of this version of algorithm merge in Figure 11.6.

**Algorithm** merge$(S_1, S_2, S)$:
*Input:* Sorted sequences $S_1$ and $S_2$ and an empty sequence $S$, implemented as linked lists
*Output:* Sorted sequence $S$ containing the elements from $S_1$ and $S_2$

```
while S1 is not empty and S2 is not empty do
  if S1.front().element() ≤ S2.front().element() then
    {move the first element of S1 at the end of S}
    S.insertBack(S1.eraseFront())
  else
    {move the first element of S2 at the end of S}
    S.insertBack(S2.eraseFront())
{move the remaining elements of S1 to S}
while S1 is not empty do
  S.insertBack(S1.eraseFront())
{move the remaining elements of S2 to S}
while S2 is not empty do
  S.insertBack(S2.eraseFront())
```

**Code Fragment 11.2:** Algorithm merge for merging two sorted sequences implemented as linked lists.

### The Running Time for Merging

We analyze the running time of the merge algorithm by making some simple observations. Let $n_1$ and $n_2$ be the number of elements of $S_1$ and $S_2$, respectively. Algorithm merge has three **while** loops. Independent of whether we are analyzing the array-based version or the list-based version, the operations performed inside each loop take $O(1)$ time each. The key observation is that during each iteration of one of the loops, one element is copied or moved from either $S_1$ or $S_2$ into $S$ (and that element is no longer considered). Since no insertions are performed into $S_1$ or $S_2$, this observation implies that the overall number of iterations of the three loops is $n_1 + n_2$. Thus, the running time of algorithm merge is $O(n_1 + n_2)$.

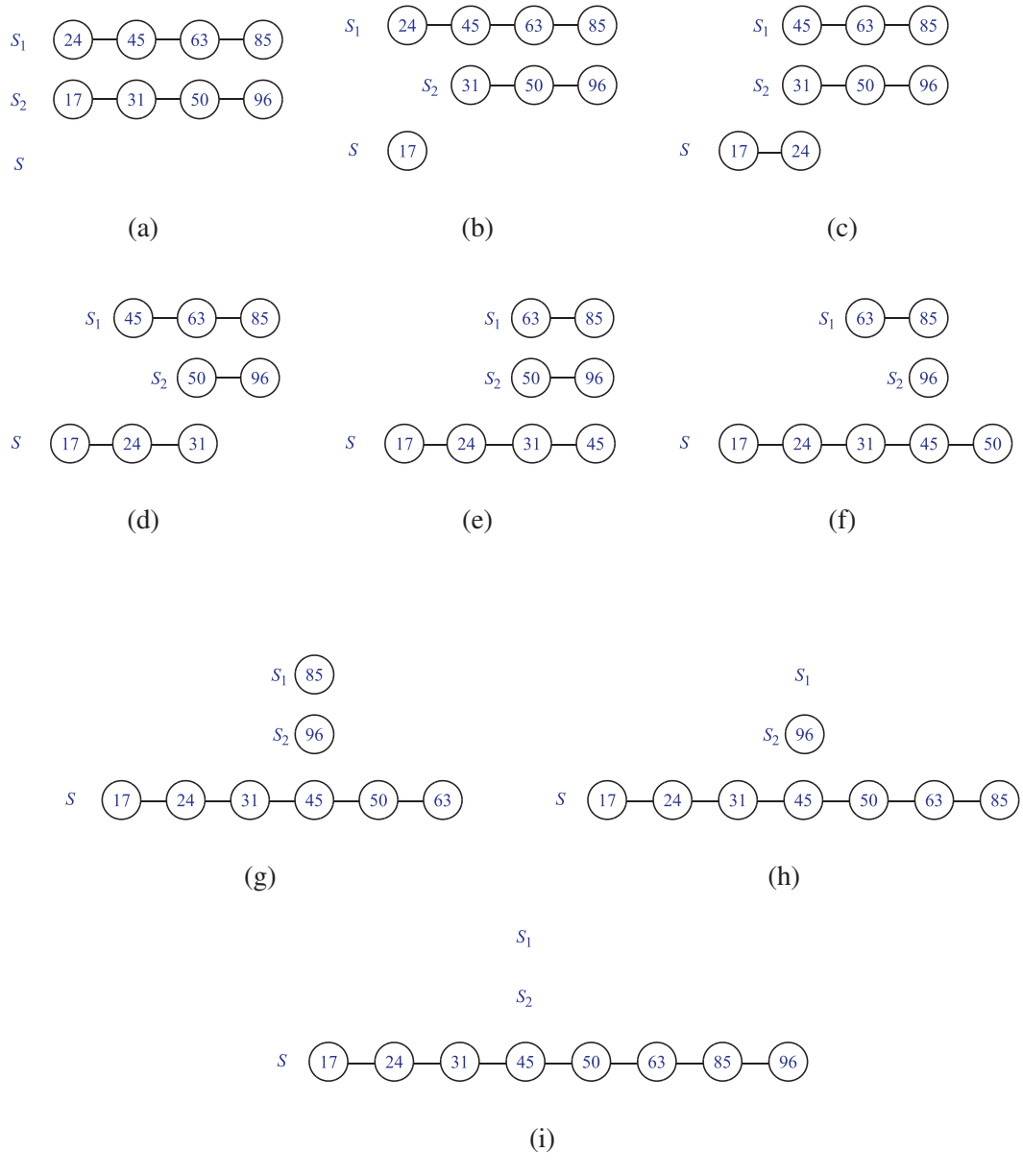



**Figure 11.6:** An execution of the algorithm merge shown in Code Fragment 11.2.

### 11.1.3 The Running Time of Merge-Sort

Now that we have given the details of the merge-sort algorithm in both its array-based and list-based versions, and we have analyzed the running time of the crucial merge algorithm used in the conquer step, let us analyze the running time of the entire merge-sort algorithm, assuming it is given an input sequence of $n$ elements. For simplicity, we restrict our attention to the case where $n$ is a power of 2. We leave it as an exercise (Exercise R-11.7) to show that the result of our analysis also holds when $n$ is not a power of 2.

As we did in the analysis of the merge algorithm, we assume that the input sequence $S$ and the auxiliary sequences $S_1$ and $S_2$, created by each recursive call of merge-sort, are implemented by either arrays or linked lists (the same as $S$), so that merging two sorted sequences can be done in linear time.

As we mentioned earlier, we analyze the merge-sort algorithm by referring to the merge-sort tree $T$. (Recall Figures 11.2 through 11.4.) We call the ***time spent at a node*** $v$ of $T$ the running time of the recursive call associated with $v$, excluding the time taken waiting for the recursive calls associated with the children of $v$ to terminate. In other words, the time spent at node $v$ includes the running times of the divide and conquer steps, but excludes the running time of the recur step. We have already observed that the details of the divide step are straightforward; this step runs in time proportional to the size of the sequence for $v$. In addition, as discussed above, the conquer step, which consists of merging two sorted subsequences, also takes linear time, independent of whether we are dealing with arrays or linked lists. That is, letting $i$ denote the depth of node $v$, the time spent at node $v$ is $O(n/2^i)$, since the size of the sequence handled by the recursive call associated with $v$ is equal to $n/2^i$.

Looking at the tree $T$ more globally, as shown in Figure 11.7, we see that, given our definition of "time spent at a node," the running time of merge-sort is equal to the sum of the times spent at the nodes of $T$. Observe that $T$ has exactly $2^i$ nodes at depth $i$. This simple observation has an important consequence, for it implies that the overall time spent at all the nodes of $T$ at depth $i$ is $O(2^i \cdot n/2^i)$, which is $O(n)$. By Proposition 11.1, the height of $T$ is $\lceil \log n \rceil$. Thus, since the time spent at each of the $\lceil \log n \rceil + 1$ levels of $T$ is $O(n)$, we have the following result.

**Proposition 11.2:** *Algorithm merge-sort sorts a sequence $S$ of size $n$ in $O(n \log n)$ time, assuming two elements of $S$ can be compared in $O(1)$ time.*

In other words, the merge-sort algorithm asymptotically matches the fast running time of the heap-sort algorithm.

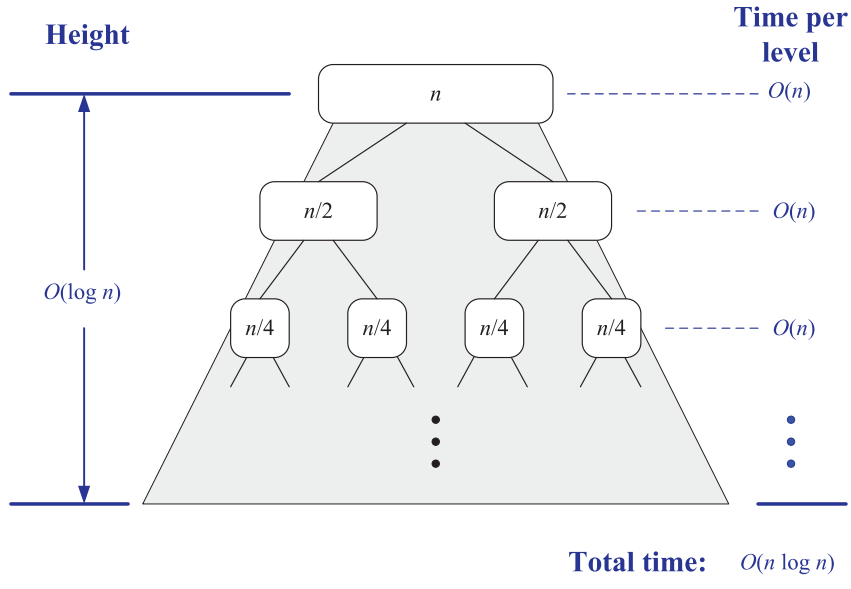



**Figure 11.7:** A visual time analysis of the merge-sort tree $T$. Each node is shown labeled with the size of its subproblem.

## 11.1.4 C++ Implementations of Merge-Sort

In this subsection, we present two complete C++ implementations of the merge-sort algorithm, one for lists and one for vectors. In both cases a comparator object (see Section 8.1.2) is used to decide the relative order of the elements. Recall that a comparator is a class that implements the less-than operator by overloading the "()" operator. For example, given a comparator object *less*, the relational test $x < y$ can be implemented with $less(x,y)$, and the test $x \leq y$ can be implemented as $!less(y,x)$.

First, in Code Fragment 11.3, we present a C++ implementation of a list-based merge-sort algorithm as the recursive function mergeSort. We represent each sequence as an STL list (Section 6.2.4). The merge process is loosely based on the algorithm presented in Code Fragment 11.2. The main function mergeSort partitions the input list $S$ into two auxiliary lists, $S_1$ and $S_2$, of roughly equal sizes. They are each sorted recursively, and the results are then combined by invoking the function merge. The function merge repeatedly moves the smaller element of the two lists $S_1$ and $S_2$ into the output list $S$.

Functions from our list ADT, such as front and insertBack, have been replaced by their STL equivalents, such as begin and push_back, respectively. Access to elements of the list is provided by list iterators. Given an iterator $p$, recall that $*p$ accesses the current element, and $*p{+}{+}$ accesses the current element and increments the iterator to the next element of the list.

Each list is modified by insertions and deletions only at the head and tail; hence, each list update takes $O(1)$ time, assuming any list implementation based on doubly linked lists (see Table 6.2). For a list $S$ of size $n$, function mergeSort$(S, c)$ runs in time $O(n \log n)$.

```
template <typename E, typename C>                    // merge-sort S
void mergeSort(list<E>& S, const C& less) {
  typedef typename list<E>::iterator Itor;           // sequence of elements
  int n = S.size();
  if (n <= 1) return;                                // already sorted
  list<E> S1, S2;
  Itor p = S.begin();
  for (int i = 0; i < n/2; i++) S1.push_back(*p++); // copy first half to S1
  for (int i = n/2; i < n; i++) S2.push_back(*p++); // copy second half to S2
  S.clear();                                         // clear S's contents
  mergeSort(S1, less);                               // recur on first half
  mergeSort(S2, less);                               // recur on second half
  merge(S1, S2, S, less);                            // merge S1 and S2 into S
}

template <typename E, typename C>                    // merge utility
void merge(list<E>& S1, list<E>& S2, list<E>& S, const C& less) {
  typedef typename list<E>::iterator Itor;           // sequence of elements
  Itor p1 = S1.begin();
  Itor p2 = S2.begin();
  while(p1 != S1.end() && p2 != S2.end()) {          // until either is empty
    if(less(*p1, *p2))                               // append smaller to S
      S.push_back(*p1++);
    else
      S.push_back(*p2++);
  }
  while(p1 != S1.end())                              // copy rest of S1 to S
    S.push_back(*p1++);
  while(p2 != S2.end())                              // copy rest of S2 to S
    S.push_back(*p2++);
}
```

**Code Fragment 11.3:** Functions mergeSort and merge implementing a list-based merge-sort algorithm.

Next, in Code Fragment 11.4, we present a nonrecursive vector-based version of merge-sort, which also runs in $O(n \log n)$ time. It is a bit faster than recursive list-based merge-sort in practice, as it avoids the extra overheads of recursive calls and node creation. The main idea is to perform merge-sort bottom-up, performing the merges level-by-level going up the merge-sort tree. The input is an STL vector $S$.

We begin by merging every odd-even pair of elements into sorted runs of length

two. In order to keep from overwriting the vector elements, we copy elements from an input vector *in* to an output vector *out*. For example, we merge $in[0]$ and $in[1]$ into the subvector $out[0..1]$, then we merge $in[2]$ and $in[3]$ into the subvector $out[2..3]$, and so on.

We then swap the rolls of *in* and *out*, and we merge these runs of length two into runs of length four. For example we merge $in[0..1]$ with $in[2..3]$ into the subvector $out[0..3]$, then we merge $in[4..5]$ with $in[6..7]$ into the subvector $out[4..7]$. We then merge consecutive runs of length four into new runs of length eight, and so on, until the array is sorted.

The variable $b$ stores the start of the runs and $m$ stores their length. The variables $i$, $j$, and $k$ store the current indices in the various subvectors. When swapping vectors, we do not copy their entire contents. Instead, we maintain pointers to the two arrays and swap these pointers at the end of each round of subvector merges.

### 11.1.5 Merge-Sort and Recurrence Equations ⋆

There is another way to justify that the running time of the merge-sort algorithm is $O(n \log n)$ (Proposition 11.2). Namely, we can deal more directly with the recursive nature of the merge-sort algorithm. In this section, we present such an analysis of the running time of merge-sort, and in so doing introduce the mathematical concept of a ***recurrence equation*** (also known as ***recurrence relation***).

Let the function $t(n)$ denote the worst-case running time of merge-sort on an input sequence of size $n$. Since merge-sort is recursive, we can characterize function $t(n)$ by means of an equation where the function $t(n)$ is recursively expressed in terms of itself. In order to simplify our characterization of $t(n)$, let us focus our attention on the case where $n$ is a power of 2. We leave the problem of showing that our asymptotic characterization still holds in the general case as an exercise (Exercise R-11.7). In this case, we can specify the definition of $t(n)$ as

$$t(n) = \begin{cases} b & \text{if } n \leq 1 \\ 2t(n/2) + cn & \text{otherwise.} \end{cases}$$

An expression such as the one above is called a ***recurrence equation***, since the function appears on both the left- and right-hand sides of the equal sign. Although such a characterization is correct and accurate, what we really want is a big-Oh type of characterization of $t(n)$ that does not involve the function $t(n)$ itself. That is, we want a ***closed-form*** characterization of $t(n)$.

We can obtain a closed-form solution by applying the definition of a recurrence equation, assuming $n$ is relatively large. For example, after one more application of the equation above, we can write a new recurrence for $t(n)$ as

$$\begin{aligned} t(n) &= 2(2t(n/2^2) + (cn/2)) + cn \\ &= 2^2 t(n/2^2) + 2(cn/2) + cn = 2^2 t(n/2^2) + 2cn. \end{aligned}$$

```
template <typename E, typename C>                    // merge-sort S
void mergeSort(vector<E>& S, const C& less) {
  typedef vector<E> vect;
  int n = S.size();
  vect v1(S);  vect* in = &v1;                       // initial input vector
  vect v2(n);  vect* out = &v2;                      // initial output vector
  for (int m = 1; m < n; m *= 2) {                   // list sizes doubling
    for (int b = 0; b < n; b += 2*m) {               // beginning of list
      merge(*in, *out, less, b, m);                  // merge sublists
    }
    std::swap(in, out);                              // swap input with output
  }
  S = *in;                                           // copy sorted array to S
}
                              // merge in[b..b+m-1] and in[b+m..b+2*m-1]
template <typename E, typename C>
void merge(vector<E>& in, vector<E>& out, const C& less, int b, int m) {
  int i = b;                                         // index into run #1
  int j = b + m;                                     // index into run #2
  int n = in.size();
  int e1 = std::min(b + m, n);                       // end of run #1
  int e2 = std::min(b + 2*m, n);                     // end of run #2
  int k = b;
  while ((i < e1) && (j < e2)) {
    if(!less(in[j], in[i]))                          // append smaller to S
      out[k++] = in[i++];
    else
      out[k++] = in[j++];
  }
  while (i < e1)                                     // copy rest of run 1 to S
    out[k++] = in[i++];
  while (j < e2)                                     // copy rest of run 2 to S
    out[k++] = in[j++];
}
```

**Code Fragment 11.4:** Functions mergeSort and merge implementing a vector-based merge-sort algorithm. We employ the STL functions swap, which swaps two values, and min, which returns the minimum of two values. Note that the call to swap swaps only the pointers to the arrays and, hence, runs in $O(1)$ time.

If we apply the equation again, we get $t(n) = 2^3 t(n/2^3) + 3cn$. At this point, we should see a pattern emerging, so that after applying this equation $i$ times we get

$$t(n) = 2^i t(n/2^i) + icn.$$

The issue that remains, then, is to determine when to stop this process. To see when to stop, recall that we switch to the closed form $t(n) = b$ when $n \leq 1$, which occurs when $2^i = n$. In other words, this occurs when $i = \log n$. Making this substitution, then, yields

$$\begin{aligned} t(n) &= 2^{\log n} t(n/2^{\log n}) + (\log n)cn \\ &= nt(1) + cn\log n \\ &= nb + cn\log n. \end{aligned}$$

That is, we get an alternative justification of the fact that $t(n)$ is $O(n\log n)$.

## 11.2 Quick-Sort

The next sorting algorithm we discuss is called ***quick-sort***. Like merge-sort, this algorithm is also based on the ***divide-and-conquer*** paradigm, but it uses this technique in a somewhat opposite manner, as all the hard work is done ***before*** the recursive calls.

### High-Level Description of Quick-Sort

The quick-sort algorithm sorts a sequence $S$ using a simple recursive approach. The main idea is to apply the divide-and-conquer technique, whereby we divide $S$ into subsequences, recur to sort each subsequence, and then combine the sorted subsequences by a simple concatenation. In particular, the quick-sort algorithm consists of the following three steps (see Figure 11.8):

1. ***Divide:*** If $S$ has at least two elements (nothing needs to be done if $S$ has zero or one element), select a specific element $x$ from $S$, which is called the ***pivot***. As is common practice, choose the pivot $x$ to be the last element in $S$. Remove all the elements from $S$ and put them into three sequences:
   - $L$, storing the elements in $S$ less than $x$
   - $E$, storing the elements in $S$ equal to $x$
   - $G$, storing the elements in $S$ greater than $x$.

   Of course, if the elements of $S$ are all distinct, then $E$ holds just one element—the pivot itself.
2. ***Recur:*** Recursively sort sequences $L$ and $G$.
3. ***Conquer:*** Put back the elements into $S$ in order by first inserting the elements of $L$, then those of $E$, and finally those of $G$.

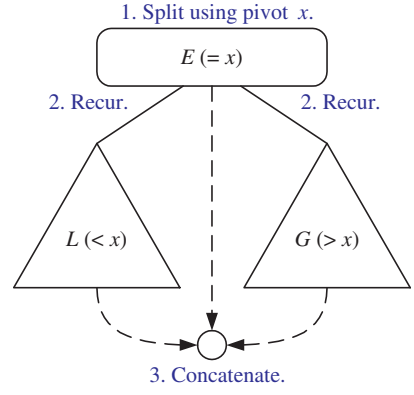


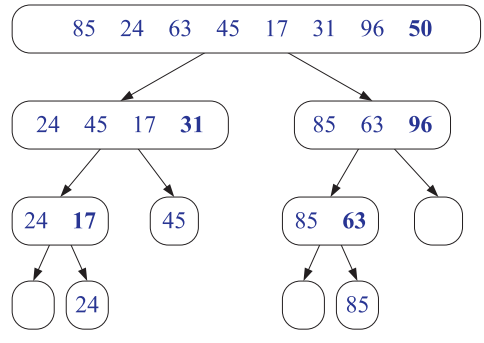

**Figure 11.8:** A visual schematic of the quick-sort algorithm.

Like merge-sort, the execution of quick-sort can be visualized by means of a binary recursion tree, called the ***quick-sort tree***. Figure 11.9 summarizes an execution of the quick-sort algorithm by showing the input and output sequences processed at each node of the quick-sort tree. The step-by-step evolution of the quick-sort tree is shown in Figures 11.10, 11.11, and 11.12.

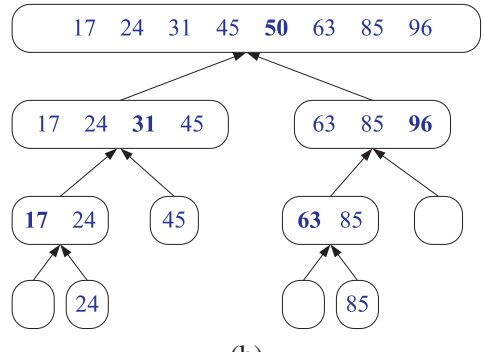
Unlike merge-sort, however, the height of the quick-sort tree associated with an execution of quick-sort is linear in the worst case. This happens, for example, if the sequence consists of $n$ distinct elements and is already sorted. Indeed, in this case, the standard choice of the pivot as the largest element yields a subsequence $L$ of size $n-1$, while subsequence $E$ has size 1 and subsequence $G$ has size 0. At each invocation of quick-sort on subsequence $L$, the size decreases by 1. Hence, the height of the quick-sort tree is $n-1$.

### Performing Quick-Sort on Arrays and Lists

In Code Fragment 11.5, we give a pseudo-code description of the quick-sort algorithm that is efficient for sequences implemented as arrays or linked lists. The algorithm follows the template for quick-sort given above, adding the detail of scanning the input sequence $S$ backwards to divide it into the lists $L$, $E$, and $G$ of elements that are respectively less than, equal to, and greater than the pivot. We perform this scan backwards, since removing the last element in a sequence is a constant-time operation independent of whether the sequence is implemented as an array or a linked list. We then recur on the $L$ and $G$ lists, and copy the sorted lists $L$, $E$, and $G$ back to $S$. We perform this latter set of copies in the forward direction, since inserting elements at the end of a sequence is a constant-time operation independent of whether the sequence is implemented as an array or a linked list.




(a)




(b)

**Figure 11.9:** Quick-sort tree $T$ for an execution of the quick-sort algorithm on a sequence with eight elements: (a) input sequences processed at each node of $T$; (b) output sequences generated at each node of $T$. The pivot used at each level of the recursion is shown in bold.

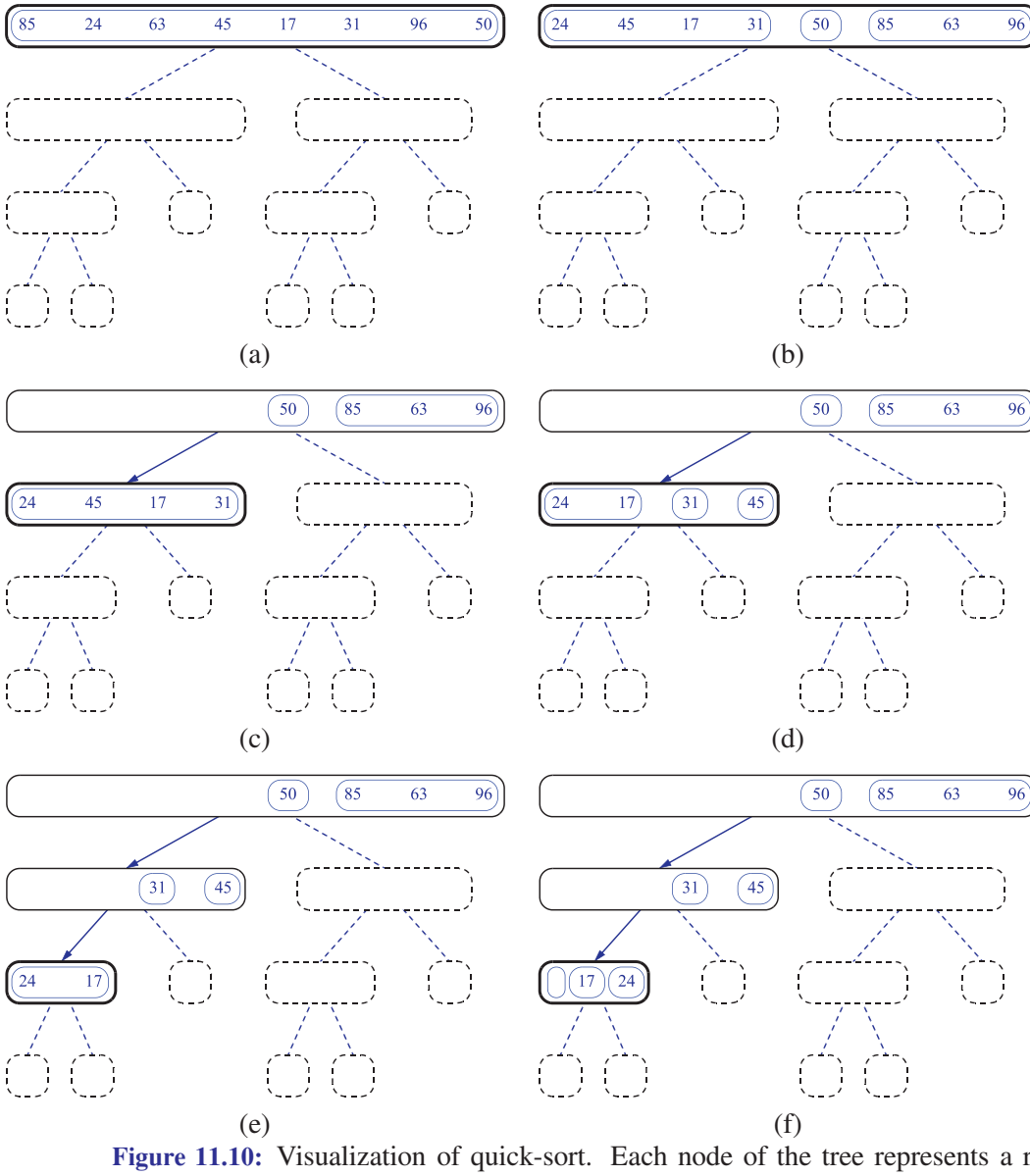



**Figure 11.10:** Visualization of quick-sort. Each node of the tree represents a recursive call. The nodes drawn with dashed lines represent calls that have not been made yet. The node drawn with thick lines represents the running invocation. The empty nodes drawn with thin lines represent terminated calls. The remaining nodes represent suspended calls (that is, active invocations that are waiting for a child invocation to return). Note the divide steps performed in (b), (d), and (f). (Continues in Figure 11.11.)

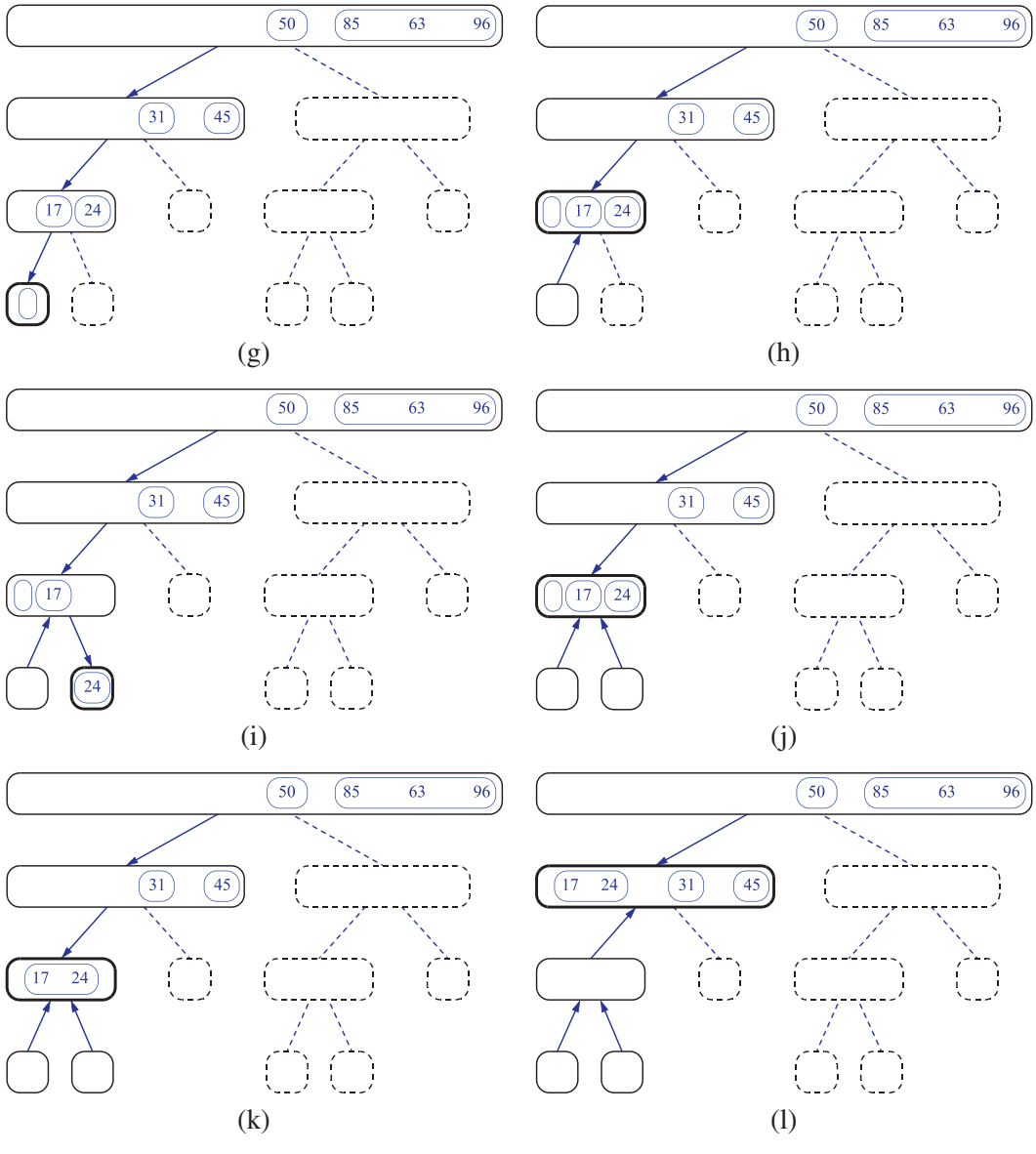



**Figure 11.11:** Visualization of an execution of quick-sort. Note the conquer step performed in (k). (Continues in Figure 11.12.)

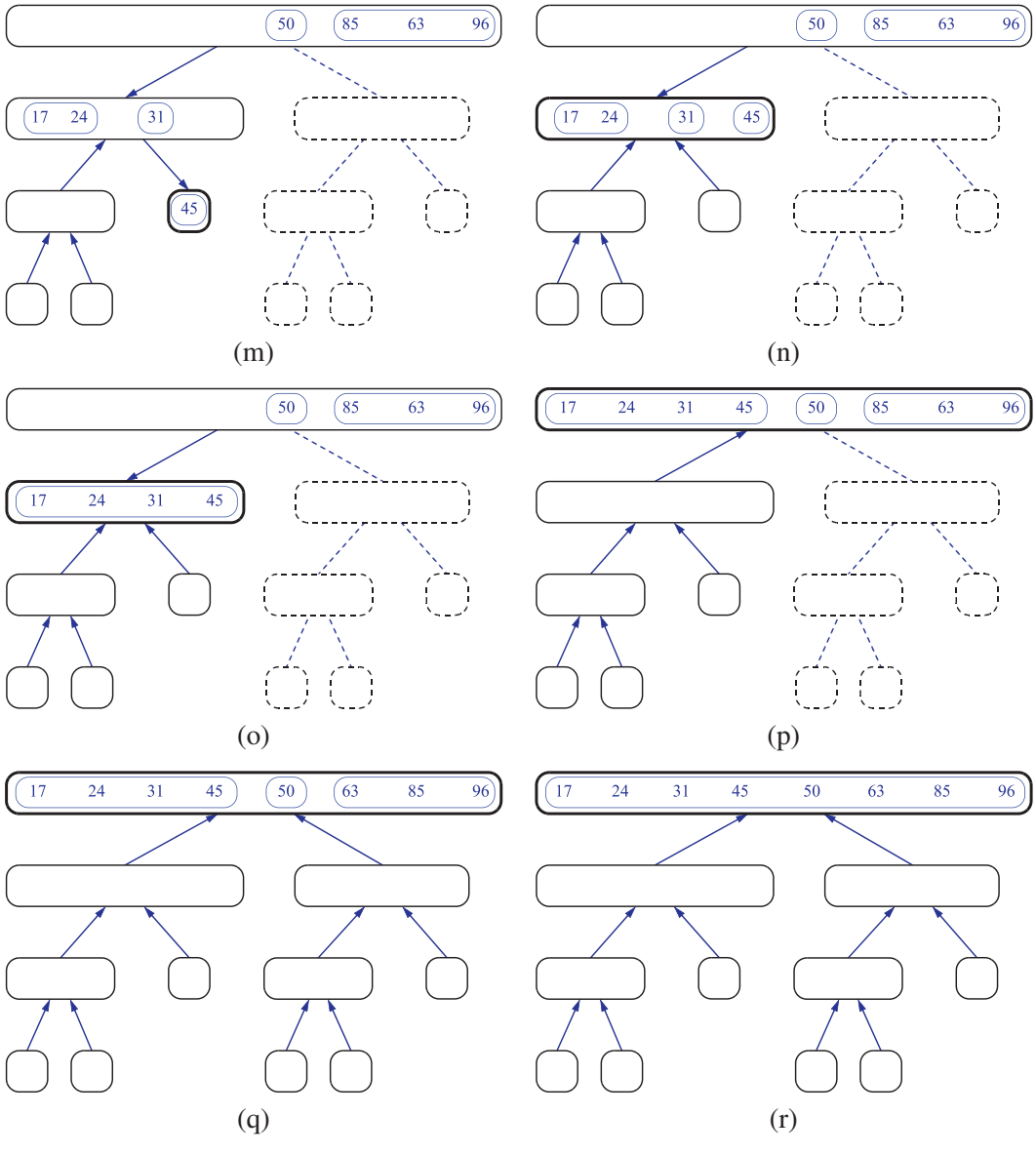



**Figure 11.12:** Visualization of an execution of quick-sort. Several invocations between (p) and (q) have been omitted. Note the conquer steps performed in (o) and (r). (Continued from Figure 11.11.)

```
Algorithm QuickSort(S):
    Input: A sequence S implemented as an array or linked list
    Output: The sequence S in sorted order
    if S.size() ≤ 1 then
        return          {S is already sorted in this case}
    p ← S.back().element()          {the pivot}
    Let L, E, and G be empty list-based sequences
    while !S.empty() do {scan S backwards, dividing it into L, E, and G}
        if S.back().element() < p then
            L.insertBack(S.eraseBack())
        else if S.back().element() = p then
            E.insertBack(S.eraseBack())
        else {the last element in S is greater than p}
            G.insertBack(S.eraseBack())
    QuickSort(L)          {Recur on the elements less than p}
    QuickSort(G)          {Recur on the elements greater than p}
    while !L.empty() do {copy back to S the sorted elements less than p}
        S.insertBack(L.eraseFront())
    while !E.empty() do {copy back to S the elements equal to p}
        S.insertBack(E.eraseFront())
    while !G.empty() do {copy back to S the sorted elements greater than p}
        S.insertBack(G.eraseFront())
    return          {S is now in sorted order}
```

**Code Fragment 11.5:** Quick-sort for an input sequence $S$ implemented with a linked list or an array.

### Running Time of Quick-Sort

We can analyze the running time of quick-sort with the same technique used for merge-sort in Section 11.1.3. Namely, we can identify the time spent at each node of the quick-sort tree $T$ and sum up the running times for all the nodes.

Examining Code Fragment 11.5, we see that the divide step and the conquer step of quick-sort can be implemented in linear time. Thus, the time spent at a node $v$ of $T$ is proportional to the ***input size*** $s(v)$ of $v$, defined as the size of the sequence handled by the invocation of quick-sort associated with node $v$. Since subsequence $E$ has at least one element (the pivot), the sum of the input sizes of the children of $v$ is at most $s(v)-1$.

Given a quick-sort tree $T$, let $s_i$ denote the sum of the input sizes of the nodes at depth $i$ in $T$. Clearly, $s_0 = n$, since the root $r$ of $T$ is associated with the entire sequence. Also, $s_1 \leq n-1$, since the pivot is not propagated to the children of $r$. Consider next $s_2$. If both children of $r$ have nonzero input size, then $s_2 = n-3$. Otherwise (one child of the root has zero size, the other has size $n-1$), $s_2 = n-2$. Thus, $s_2 \leq n-2$. Continuing this line of reasoning, we obtain that $s_i \leq n-i$. As observed in Section 11.2, the height of $T$ is $n-1$ in the worst case. Thus, the worst-case running time of quick-sort is $O\left(\sum_{i=0}^{n-1} s_i\right)$, which is $O\left(\sum_{i=0}^{n-1}(n-i)\right)$, that is, $O\left(\sum_{i=1}^{n} i\right)$ . By Proposition 4.3, $\sum_{i=1}^{n} i$ is $O(n^2)$. Thus, quick-sort runs in $O(n^2)$ worst-case time.

Given its name, we would expect quick-sort to run quickly. However, the quadratic bound above indicates that quick-sort is slow in the worst case. Paradoxically, this worst-case behavior occurs for problem instances when sorting should be easy—if the sequence is already sorted.

Going back to our analysis, note that the best case for quick-sort on a sequence of distinct elements occurs when subsequences $L$ and $G$ happen to have roughly the same size. That is, in the best case, we have

$$\begin{aligned} s_0 &= n \\ s_1 &= n-1 \\ s_2 &= n-(1+2) = n-3 \\ &\vdots \\ s_i &= n-(1+2+2^2+\cdots+2^{i-1}) = n-(2^i-1). \end{aligned}$$

Thus, in the best case, $T$ has height $O(\log n)$ and quick-sort runs in $O(n \log n)$ time. We leave the justification of this fact as an exercise (Exercise R-11.12).

The informal intuition behind the expected behavior of quick-sort is that at each invocation the pivot will probably divide the input sequence about equally. Thus, we expect the average running time quick-sort to be similar to the best-case running time, that is, $O(n \log n)$. In the next section, we see that introducing randomization makes quick-sort behave exactly in this way.

### 11.2.1 Randomized Quick-Sort

One common method for analyzing quick-sort is to assume that the pivot always divides the sequence almost equally. We feel such an assumption would presuppose knowledge about the input distribution that is typically not available, however. For example, we would have to assume that we will rarely be given "almost" sorted sequences to sort, which are actually common in many applications. Fortunately, this assumption is not needed in order for us to match our intuition to quick-sort's behavior.

In general, we desire some way of getting close to the best-case running time for quick-sort. The way to get close to the best-case running time, of course, is for the pivot to divide the input sequence $S$ almost equally. If this outcome were to occur, then it would result in a running time that is asymptotically the same as the best-case running time. That is, having pivots close to the "middle" of the set of elements leads to an $O(n \log n)$ running time for quick-sort.

#### Picking Pivots at Random

Since the goal of the partition step of the quick-sort method is to divide the sequence $S$ almost equally, let us introduce randomization into the algorithm and pick a ***random element*** of the input sequence as the pivot. That is, instead of picking the pivot as the last element of $S$, we pick an element of $S$ at random as the pivot, keeping the rest of the algorithm unchanged. This variation of quick-sort is called ***randomized quick-sort***. The following proposition shows that the expected running time of randomized quick-sort on a sequence with $n$ elements is $O(n \log n)$. This expectation is taken over all the possible random choices the algorithm makes, and is independent of any assumptions about the distribution of the possible input sequences the algorithm is likely to be given.

**Proposition 11.3:** *The expected running time of randomized quick-sort on a sequence $S$ of size $n$ is $O(n \log n)$.*

**Justification:** We assume two elements of $S$ can be compared in $O(1)$ time. Consider a single recursive call of randomized quick-sort, and let $n$ denote the size of the input for this call. Say that this call is "good" if the pivot chosen is such that subsequences $L$ and $G$ have size at least $n/4$ and at most $3n/4$ each; otherwise, a call is "bad."

Now, consider the implications of our choosing a pivot uniformly at random. Note that there are $n/2$ possible good choices for the pivot for any given call of size $n$ of the randomized quick-sort algorithm. Thus, the probability that any call is good is $1/2$. Note further that a good call will at least partition a list of size $n$ into two lists of size $3n/4$ and $n/4$, and a bad call could be as bad as producing a single call of size $n-1$.

Now consider a recursion trace for randomized quick-sort. This trace defines a binary tree, $T$, such that each node in $T$ corresponds to a different recursive call on a subproblem of sorting a portion of the original list.

Say that a node $v$ in $T$ is in ***size group*** $i$ if the size of $v$'s subproblem is greater than $(3/4)^{i+1}n$ and at most $(3/4)^i n$. Let us analyze the expected time spent working on all the subproblems for nodes in size group $i$. By the linearity of expectation (Proposition A.19), the expected time for working on all these subproblems is the sum of the expected times for each one. Some of these nodes correspond to good calls and some correspond to bad calls. But note that, since a good call occurs with probability $1/2$, the expected number of consecutive calls we have to make before getting a good call is 2. Moreover, notice that as soon as we have a good call for a node in size group $i$, its children will be in size groups higher than $i$. Thus, for any element $x$ from the input list, the expected number of nodes in size group $i$ containing $x$ in their subproblems is 2. In other words, the expected total size of all the subproblems in size group $i$ is $2n$. Since the nonrecursive work we perform for any subproblem is proportional to its size, this implies that the total expected time spent processing subproblems for nodes in size group $i$ is $O(n)$.

The number of size groups is $\log_{4/3} n$, since repeatedly multiplying by $3/4$ is the same as repeatedly dividing by $4/3$. That is, the number of size groups is $O(\log n)$. Therefore, the total expected running time of randomized quick-sort is $O(n \log n)$. (See Figure 11.13.) ■

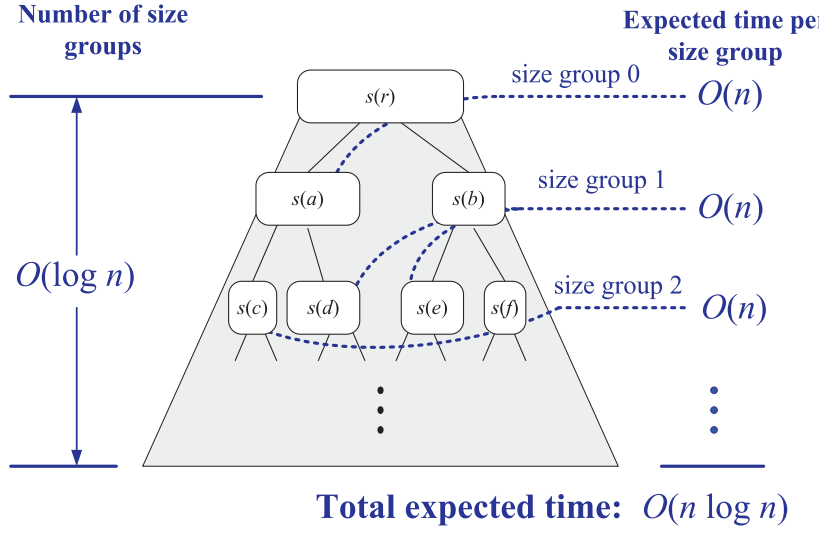



**Figure 11.13:** A visual time analysis of the quick-sort tree $T$. Each node is shown labeled with the size of its subproblem.

Actually, we can show that the running time of randomized quick-sort is $O(n \log n)$ with high probability.

### 11.2.2 C++ Implementations and Optimizations

Recall from Section 8.3.5 that a sorting algorithm is ***in-place*** if it uses only a small amount of memory in addition to that needed for the objects being sorted themselves. The merge-sort algorithm, as described above, does not use this optimization technique, and making it be in-place seems to be quite difficult. In-place sorting is not inherently difficult, however. For, as with heap-sort, quick-sort can be adapted to be in-place, and this is the version of quick-sort that is used in most deployed implementations.

Performing the quick-sort algorithm in-place requires a bit of ingenuity, however, for we must use the input sequence itself to store the subsequences for all the recursive calls. We show algorithm inPlaceQuickSort, which performs in-place quick-sort, in Code Fragment 11.6. Algorithm inPlaceQuickSort assumes that the input sequence, $S$, is given as an array of ***distinct*** elements. The reason for this restriction is explored in Exercise R-11.15. The extension to the general case is discussed in Exercise C-11.9.

**Algorithm** inPlaceQuickSort$(S,a,b)$:
***Input:*** An array $S$ of distinct elements; integers $a$ and $b$
***Output:*** Array $S$ with elements originally from indices from $a$ to $b$, inclusive, sorted in nondecreasing order from indices $a$ to $b$

```
if a ≥ b then return          {at most one element in subrange}
p ← S[b]          {the pivot}
l ← a          {will scan rightward}
r ← b − 1          {will scan leftward}
while l ≤ r do
  {find an element larger than the pivot}
  while l ≤ r and S[l] ≤ p do
    l ← l + 1
  {find an element smaller than the pivot}
  while r ≥ l and S[r] ≥ p do
    r ← r − 1
  if l < r then
    swap the elements at S[l] and S[r]
{put the pivot into its final place}
swap the elements at S[l] and S[b]
{recursive calls}
inPlaceQuickSort(S, a, l − 1)
inPlaceQuickSort(S, l + 1, b)
{we are done at this point, since the sorted subarrays are already consecutive}
```

**Code Fragment 11.6:** In-place quick-sort for an input array $S$.

In-place quick-sort modifies the input sequence using element swapping and does not explicitly create subsequences. Indeed, a subsequence of the input sequence is implicitly represented by a range of positions specified by a left-most index $l$ and a right-most index $r$. The divide step is performed by scanning the array simultaneously from $l$ forward and from $r$ backward, swapping pairs of elements that are in reverse order as shown in Figure 11.14. When these two indices "meet," subvectors $L$ and $G$ are on opposite sides of the meeting point. The algorithm completes by recurring on these two subvectors.

In-place quick-sort reduces the running time caused by the creation of new sequences and the movement of elements between them by a constant factor. It is so efficient that the STL's sorting algorithm is based in part on quick-sort.

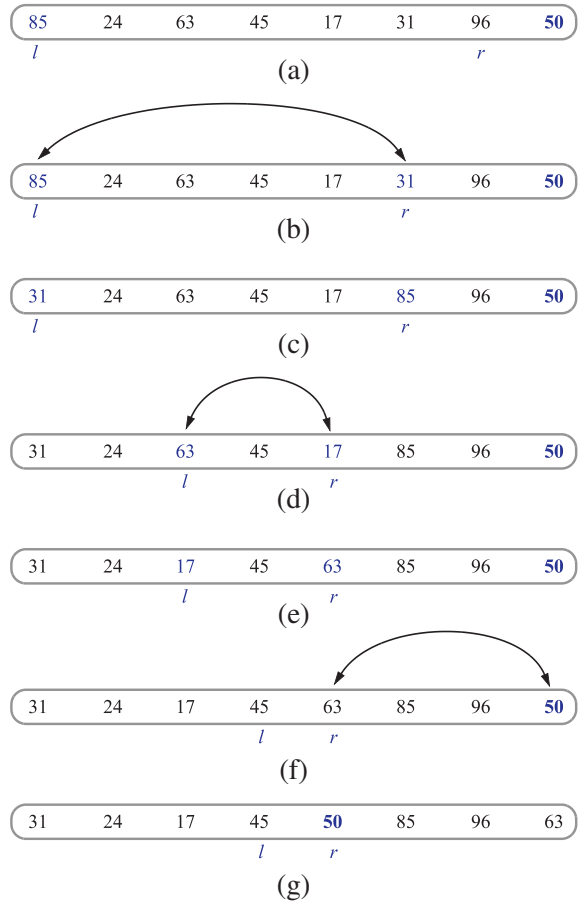



**Figure 11.14:** Divide step of in-place quick-sort. Index $l$ scans the sequence from left to right, and index $r$ scans the sequence from right to left. A swap is performed when $l$ is at an element larger than the pivot and $r$ is at an element smaller than the pivot. A final swap with the pivot completes the divide step.

We show a C++ version of in-place quick-sort in Code Fragment 11.7. The input to the sorting procedure is an STL vector of elements and a comparator object, which provides the less-than function. Our implementation is a straightforward adaptation of Code Fragment 11.6. The main procedure, quickSort, invokes the recursive procedure quickSortStep to do most of the work.

```
template <typename E, typename C>                    // quick-sort S
void quickSort(std::vector<E>& S, const C& less) {
  if (S.size() <= 1) return;                         // already sorted
  quickSortStep(S, 0, S.size()-1, less);             // call sort utility
}
template <typename E, typename C>
void quickSortStep(std::vector<E>& S, int a, int b, const C& less) {
  if (a >= b) return;                                // 0 or 1 left? done
  E pivot = S[b];                                    // select last as pivot
  int l = a;                                         // left edge
  int r = b - 1;                                     // right edge
  while (l <= r) {
    while (l <= r && !less(pivot, S[l])) l++;        // scan right till larger
    while (r >= l && !less(S[r], pivot)) r--;        // scan left till smaller
    if (l < r)                                       // both elements found
      std::swap(S[l], S[r]);
  }                                                  // until indices cross
  std::swap(S[l], S[b]);                             // store pivot at l
  quickSortStep(S, a, l-1, less);                    // recur on both sides
  quickSortStep(S, l+1, b, less);
}
```

**Code Fragment 11.7:** A coding of in-place quick-sort, assuming distinct elements.

The function quickSortStep is given indices $a$ and $b$, which indicate the bounds of the subvector to be sorted. The pivot element is chosen to be the last element of the vector. The indices $l$ and $r$ mark the left and right ends of the subvectors being processed in the partitioning function. They are initialized to $a$ and $b-1$, respectively. During each round, elements that are on the wrong side of the pivot are swapped with each other, until these markers bump into each other.

Much of the efficiency of quick-sort depends on how the pivot is chosen. As we have seen, quick-sort is most efficient if the pivot is near the middle of the subvector being sorted. Our choice of setting the pivot to the last element of the subvector relies on the assumption that the last element is reflective of the median key valye. A better choice, if the subvector is moderately sized, is to select the pivot as the median of three values, taken respectively from the front, middle, and tail of the array. This is referred to as the ***median-of-three*** heuristic. It tends to perform well in practice, and is faster than selecting a random pivot through the use of a random-number generator.

## 11.3 Studying Sorting through an Algorithmic Lens

Recapping our discussions on sorting to this point, we have described several methods with either a worst-case or expected running time of $O(n \log n)$ on an input sequence of size $n$. These methods include merge-sort and quick-sort, described in this chapter, as well as heap-sort (Section 8.3.5). In this section, we study sorting as an algorithmic problem, addressing general issues about sorting algorithms.

### 11.3.1 A Lower Bound for Sorting

A natural first question to ask is whether we can sort any faster than $O(n \log n)$ time. Interestingly, if the computational primitive used by a sorting algorithm is the comparison of two elements, then this is, in fact, the best we can do—comparison-based sorting has an $\Omega(n \log n)$ worst-case lower bound on its running time. (Recall the notation $\Omega(\cdot)$ from Section 4.2.3.) To focus on the main cost of comparison-based sorting, let us only count comparisons, for the sake of a lower bound.

Suppose we are given a sequence $S = (x_0, x_1, \ldots, x_{n-1})$ that we wish to sort, and assume that all the elements of $S$ are distinct (this is not really a restriction since we are deriving a lower bound). We do not care if $S$ is implemented as an array or a linked list, for the sake of our lower bound, since we are only counting comparisons. Each time a sorting algorithm compares two elements $x_i$ and $x_j$, that is, it asks, "is $x_i < x_j$?", there are two outcomes: "yes" or "no." Based on the result of this comparison, the sorting algorithm may perform some internal calculations (which we are not counting here) and eventually performs another comparison between two other elements of $S$, which again has two outcomes. Therefore, we can represent a comparison-based sorting algorithm with a decision tree $T$ (recall Example 7.8). That is, each internal node $v$ in $T$ corresponds to a comparison and the edges from node $v'$ to its children correspond to the computations resulting from either a "yes" or "no" answer. It is important to note that the hypothetical sorting algorithm in question probably has no explicit knowledge of the tree $T$. $T$ simply represents all the possible sequences of comparisons that a sorting algorithm might make, starting from the first comparison (associated with the root) and ending with the last comparison (associated with the parent of an external node).

Each possible initial ordering, or ***permutation***, of the elements in $S$ causes our hypothetical sorting algorithm to execute a series of comparisons, traversing a path in $T$ from the root to some external node. Let us associate with each external node $v$ in $T$, then, the set of permutations of $S$ that cause our sorting algorithm to end up in $v$. The most important observation in our lower-bound argument is that each external node $v$ in $T$ can represent the sequence of comparisons for at most one permutation of $S$. The justification for this claim is simple: if two different

permutations $P_1$ and $P_2$ of $S$ are associated with the same external node, then there are at least two objects $x_i$ and $x_j$, such that $x_i$ is before $x_j$ in $P_1$ but $x_i$ is after $x_j$ in $P_2$. At the same time, the output associated with $v$ must be a specific reordering of $S$, with either $x_i$ or $x_j$ appearing before the other. But if $P_1$ and $P_2$ both cause the sorting algorithm to output the elements of $S$ in this order, then that implies there is a way to trick the algorithm into outputting $x_i$ and $x_j$ in the wrong order. Since this cannot be allowed by a correct sorting algorithm, each external node of $T$ must be associated with exactly one permutation of $S$. We use this property of the decision tree associated with a sorting algorithm to prove the following result.

**Proposition 11.4:** *The running time of any comparison-based algorithm for sorting an $n$-element sequence is $\Omega(n \log n)$ in the worst case.*

**Justification:** The running time of a comparison-based sorting algorithm must be greater than or equal to the height of the decision tree $T$ associated with this algorithm, as described above. (See Figure 11.15.) By the argument above, each external node in $T$ must be associated with one permutation of $S$. Moreover, each permutation of $S$ must result in a different external node of $T$. The number of permutations of $n$ objects is $n! = n(n-1)(n-2)\cdots 2 \cdot 1$. Thus, $T$ must have at least $n!$ external nodes. By Proposition 7.10, the height of $T$ is at least $\log(n!)$. This immediately justifies the proposition, because there are at least $n/2$ terms that are greater than or equal to $n/2$ in the product $n!$; hence

$$\log(n!) \geq \log\left(\frac{n}{2}\right)^{\frac{n}{2}} = \frac{n}{2}\log\frac{n}{2},$$

which is $\Omega(n \log n)$. ■

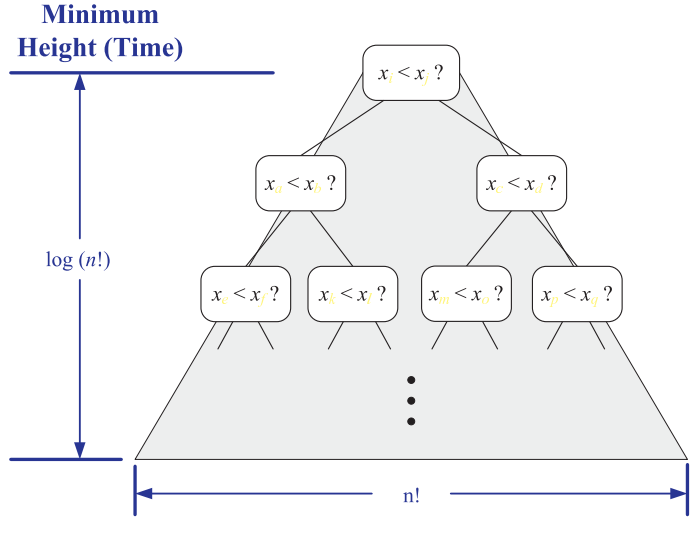



**Figure 11.15:** Visualizing the lower bound for comparison-based sorting.

## 11.3.2 Linear-Time Sorting: Bucket-Sort and Radix-Sort

In the previous section, we showed that $\Omega(n \log n)$ time is necessary, in the worst case, to sort an $n$-element sequence with a comparison-based sorting algorithm. A natural question to ask, then, is whether there are other kinds of sorting algorithms that can be designed to run asymptotically faster than $O(n \log n)$ time. Interestingly, such algorithms exist, but they require special assumptions about the input sequence to be sorted. Even so, such scenarios often arise in practice, so discussing them is worthwhile. In this section, we consider the problem of sorting a sequence of entries, each a key-value pair, where the keys have a restricted type.

### Bucket-Sort

Consider a sequence $S$ of $n$ entries whose keys are integers in the range $[0, N-1]$, for some integer $N \geq 2$, and suppose that $S$ should be sorted according to the keys of the entries. In this case, it is possible to sort $S$ in $O(n+N)$ time. It might seem surprising, but this implies, for example, that if $N$ is $O(n)$, then we can sort $S$ in $O(n)$ time. Of course, the crucial point is that, because of the restrictive assumption about the format of the elements, we can avoid using comparisons.

The main idea is to use an algorithm called ***bucket-sort***, which is not based on comparisons, but on using keys as indices into a bucket array $B$ that has cells indexed from 0 to $N-1$. An entry with key $k$ is placed in the "bucket" $B[k]$, which itself is a sequence (of entries with key $k$). After inserting each entry of the input sequence $S$ into its bucket, we can put the entries back into $S$ in sorted order by enumerating the contents of the buckets $B[0], B[1], \ldots, B[N-1]$ in order. We describe the bucket-sort algorithm in Code Fragment 11.8.

**Algorithm** bucketSort($S$):
*Input:* Sequence $S$ of entries with integer keys in the range $[0, N-1]$
*Output:* Sequence $S$ sorted in nondecreasing order of the keys

```
let B be an array of N sequences, each of which is initially empty
for each entry e in S do
    k ← e.key()
    remove e from S and insert it at the end bucket (sequence) B[k]
for i ← 0 to N − 1 do
    for each entry e in sequence B[i] do
        remove e from B[i] and insert it at the end of S
```

**Code Fragment 11.8:** Bucket-sort.

It is easy to see that bucket-sort runs in $O(n+N)$ time and uses $O(n+N)$ space. Hence, bucket-sort is efficient when the range $N$ of values for the keys is small compared to the sequence size $n$, say $N = O(n)$ or $N = O(n \log n)$. Still, its performance deteriorates as $N$ grows compared to $n$.

An important property of the bucket-sort algorithm is that it works correctly even if there are many different elements with the same key. Indeed, we described it in a way that anticipates such occurrences.

### Stable Sorting

When sorting key-value pairs, an important issue is how equal keys are handled. Let $S = ((k_0,x_0),\ldots,(k_{n-1},x_{n-1}))$ be a sequence of such entries. We say that a sorting algorithm is ***stable*** if, for any two entries $(k_i,x_i)$ and $(k_j,x_j)$ of $S$, such that $k_i = k_j$ and $(k_i,x_i)$ precedes $(k_j,x_j)$ in $S$ before sorting (that is, $i < j$), entry $(k_i,x_i)$ also precedes entry $(k_j,x_j)$ after sorting. Stability is important for a sorting algorithm because applications may want to preserve the initial ordering of elements with the same key.

Our informal description of bucket-sort in Code Fragment 11.8 does not guarantee stability. This is not inherent in the bucket-sort method itself, however, for we can easily modify our description to make bucket-sort stable, while still preserving its $O(n+N)$ running time. Indeed, we can obtain a stable bucket-sort algorithm by always removing the ***first*** element from sequence $S$ and from the sequences $B[i]$ during the execution of the algorithm.

### Radix-Sort

One of the reasons that stable sorting is so important is that it allows the bucket-sort approach to be applied to more general contexts than to sort integers. Suppose, for example, that we want to sort entries with keys that are pairs $(k,l)$, where $k$ and $l$ are integers in the range $[0,N-1]$, for some integer $N \geq 2$. In a context such as this, it is natural to define an ordering on these keys using the ***lexicographical*** (dictionary) convention, where $(k_1,l_1) < (k_2,l_2)$ if $k_1 < k_2$ or if $k_1 = k_2$ and $l_1 < l_2$ (Section 8.1.2). This is a pair-wise version of the lexicographic comparison function, usually applied to equal-length character strings (and it easily generalizes to tuples of $d$ numbers for $d > 2$).

The ***radix-sort*** algorithm sorts a sequence $S$ of entries with keys that are pairs, by applying a stable bucket-sort on the sequence twice; first using one component of the pair as the ordering key and then using the second component. But which order is correct? Should we first sort on the $k$'s (the first component) and then on the $l$'s (the second component), or should it be the other way around?

Before we answer this question, we consider the following example.

**Example 11.5:** *Consider the following sequence $S$ (we show only the keys):*

$$S = ((3,3),(1,5),(2,5),(1,2),(2,3),(1,7),(3,2),(2,2)).$$

*If we sort $S$ stably on the first component, then we get the sequence*

$$S_1 = ((1,5),(1,2),(1,7),(2,5),(2,3),(2,2),(3,3),(3,2)).$$

*If we then stably sort this sequence $S_1$ using the second component, then we get the sequence*

$$S_{1,2} = ((1,2),(2,2),(3,2),(2,3),(3,3),(1,5),(2,5),(1,7)),$$

*which is not exactly a sorted sequence. On the other hand, if we first stably sort $S$ using the second component, then we get the sequence*

$$S_2 = ((1,2),(3,2),(2,2),(3,3),(2,3),(1,5),(2,5),(1,7)).$$

*If we then stably sort sequence $S_2$ using the first component, then we get the sequence*

$$S_{2,1} = ((1,2),(1,5),(1,7),(2,2),(2,3),(2,5),(3,2),(3,3)),$$

*which is indeed sequence $S$ lexicographically ordered.*

So, from this example, we are led to believe that we should first sort using the second component and then again using the first component. This intuition is exactly right. By first stably sorting by the second component and then again by the first component, we guarantee that if two entries are equal in the second sort (by the first component), then their relative order in the starting sequence (which is sorted by the second component) is preserved. Thus, the resulting sequence is guaranteed to be sorted lexicographically every time. We leave the determination of how this approach can be extended to triples and other $d$-tuples of numbers as a simple exercise (Exercise R-11.20). We can summarize this section as follows:

**Proposition 11.6:** *Let $S$ be a sequence of $n$ key-value pairs, each of which has a key $(k_1,k_2,\ldots,k_d)$, where $k_i$ is an integer in the range $[0,N-1]$ for some integer $N \geq 2$. We can sort $S$ lexicographically in time $O(d(n+N))$ using radix-sort.*

As important as it is, sorting is not the only interesting problem dealing with a total order relation on a set of elements. There are some applications, for example, that do not require an ordered listing of an entire set, but nevertheless call for some amount of ordering information about the set. Before we study such a problem (called "selection"), let us step back and briefly compare all of the sorting algorithms we have studied so far.

### 11.3.3 Comparing Sorting Algorithms

At this point, it might be useful for us to take a breath and consider all the algorithms we have studied in this book to sort an $n$-element vector, node list, or general sequence.

#### Considering Running Time and Other Factors

We have studied several methods, such as insertion-sort and selection-sort, that have $O(n^2)$-time behavior in the average and worst case. We have also studied several methods with $O(n \log n)$-time behavior, including heap-sort, merge-sort, and quick-sort. Finally, we have studied a special class of sorting algorithms, namely, the bucket-sort and radix-sort methods, that run in linear time for certain types of keys. Certainly, the selection-sort algorithm is a poor choice in any application, since it runs in $O(n^2)$ time even in the best case. But, of the remaining sorting algorithms, which is the best?

As with many things in life, there is no clear "best" sorting algorithm from the remaining candidates. The sorting algorithm best suited for a particular application depends on several properties of that application. We can offer some guidance and observations, therefore, based on the known properties of the "good" sorting algorithms.

#### Insertion-Sort

If implemented well, the running time of ***insertion-sort*** is $O(n+m)$, where $m$ is the number of ***inversions*** (that is, the number of pairs of elements out of order). Thus, insertion-sort is an excellent algorithm for sorting small sequences (say, less than 50 elements), because insertion-sort is simple to program, and small sequences necessarily have few inversions. Also, insertion-sort is quite effective for sorting sequences that are already "almost" sorted. By "almost," we mean that the number of inversions is small. But the $O(n^2)$-time performance of insertion-sort makes it a poor choice outside of these special contexts.

#### Merge-Sort

***Merge-sort***, on the other hand, runs in $O(n \log n)$ time in the worst case, which is optimal for comparison-based sorting methods. Still, experimental studies have shown that, since it is difficult to make merge-sort run in-place, the overheads needed to implement merge-sort make it less attractive than the in-place implementations of heap-sort and quick-sort for sequences that can fit entirely in a computer's main memory area. Even so, merge-sort is an excellent algorithm for situations

where the input cannot all fit into main memory, but must be stored in blocks on an external memory device, such as a disk. In these contexts, the way that merge-sort processes runs of data in long merge streams makes the best use of all the data brought into main memory in a block from disk. Thus, for external memory sorting, the merge-sort algorithm tends to minimize the total number of disk reads and writes needed, which makes the merge-sort algorithm superior in such contexts.

### Quick-Sort

Experimental studies have shown that if an input sequence can fit entirely in main memory, then the in-place versions of quick-sort and heap-sort run faster than merge-sort. The extra overhead needed for copying nodes or entries puts merge-sort at a disadvantage to quick-sort and heap-sort in these applications. In fact, quick-sort tends, on average, to beat heap-sort in these tests. So, ***quick-sort*** is an excellent choice as a general-purpose, in-memory sorting algorithm. Indeed, it is included in the qsort sorting utility provided in C language libraries. Still, its $O(n^2)$ time worst-case performance makes quick-sort a poor choice in real-time applications where we must make guarantees on the time needed to complete a sorting operation.

### Heap-Sort

In real-time scenarios where we have a fixed amount of time to perform a sorting operation and the input data can fit into main memory, the ***heap-sort*** algorithm is probably the best choice. It runs in $O(n \log n)$ worst-case time and can easily be made to execute in-place.

### Bucket-Sort and Radix-Sort

Finally, if our application involves sorting entries with small integer keys or $d$-tuples of small integer keys, then ***bucket-sort*** or ***radix-sort*** is an excellent choice, because it runs in $O(d(n+N))$ time, where $[0, N-1]$ is the range of integer keys (and $d = 1$ for bucket sort). Thus, if $d(n+N)$ is significantly "below" the $n \log n$ function, then this sorting method should run faster than even quick-sort or heap-sort.

Thus, our study of all these different sorting algorithms provides us with a versatile collection of sorting methods in our algorithm engineering "toolbox."

# 11.4 Sets and Union/Find Structures

In this section, we study sets, including operations that define them and operations that can be applied to entire sets.

## 11.4.1 The Set ADT

A ***set*** is a collection of distinct objects. That is, there are no duplicate elements in a set, and there is no explicit notion of keys or even an order. Even so, if the elements in a set are comparable, then we can maintain sets to be ordered. The fundamental functions of the set ADT for a set $S$ are the following:

insert($e$): Insert the element $e$ into $S$ and return an iterator referring to its location; if the element already exists the operation is ignored.

find($e$): If $S$ contains $e$, return an iterator $p$ referring to this entry, else return end.

erase($e$): Remove the element $e$ from $S$.

begin(): Return an iterator to the beginning of $S$.

end(): Return an iterator to an imaginary position just beyond the end of $S$.

The C++ Standard Template Library provides a class set that contains all of these functions. It actually implements an ordered set, and supports the following additional operations as well.

lower_bound($e$): Return an iterator to the largest element less than or equal to $e$.

upper_bound($e$): Return an iterator to the smallest element greater than or equal to $e$.

equal_range($e$): Return an iterator range of elements that are equal to $e$.

The STL set is templated with the element type. As with the other STL classes we have seen so far, the set is an example of a container, and hence supports access by iterators. In order to declare an object of type set, it is necessary to first include the definition file called "set." The set is part of the std namespace, and hence it is necessary either to use "std::set" or to provide an appropriate "**using**" statement.

The STL set is implemented by adapting the STL ordered map (which is based on a red-black tree). Each entry has the property that the key and element are both equal to $e$. That is, each entry is of the form $(e, e)$.

## 11.4.2 Mergable Sets and the Template Method Pattern

Let us explore a further extension of the ordered set ADT that allows for operations between pairs of sets. This also serves to motivate a software engineering design pattern known as the ***template method***.

First, we recall the mathematical definitions of the ***union***, ***intersection***, and ***subtraction*** of two sets $A$ and $B$:

$$\begin{aligned} A \cup B &= \{x\text{: } x \text{ is in } A \text{ or } x \text{ is in } B\}, \\ A \cap B &= \{x\text{: } x \text{ is in } A \text{ and } x \text{ is in } B\}, \\ A - B &= \{x\text{: } x \text{ is in } A \text{ and } x \text{ is not in } B\}. \end{aligned}$$

**Example 11.7:** *Most Internet search engines store, for each word $x$ in their dictionary database, a set, $W(x)$, of Web pages that contain $x$, where each Web page is identified by a unique Internet address. When presented with a query for a word $x$, such a search engine need only return the Web pages in the set $W(x)$, sorted according to some proprietary priority ranking of page "importance." But when presented with a two-word query for words $x$ and $y$, such a search engine must first compute the intersection $W(x) \cap W(y)$, and then return the Web pages in the resulting set sorted by priority. Several search engines use the set intersection algorithm described in this section for this computation.*

### Fundamental Methods of the Mergable Set ADT

The fundamental functions of the mergable set ADT, acting on a set $A$, are as follows:

union($B$): Replace $A$ with the union of $A$ and $B$, that is, execute $A \leftarrow A \cup B$.

intersect($B$): Replace $A$ with the intersection of $A$ and $B$, that is, execute $A \leftarrow A \cap B$.

subtract($B$): Replace $A$ with the difference of $A$ and $B$, that is, execute $A \leftarrow A - B$.

### A Simple Mergable Set Implementation

One of the simplest ways of implementing a set is to store its elements in an ordered sequence. This implementation is included in several software libraries for generic data structures, for example. Therefore, let us consider implementing the set ADT with an ordered sequence (we consider other implementations in several exercises). Any consistent total order relation among the elements of the set can be used, provided the same order is used for all the sets.

We implement each of the three fundamental set operations using a generic version of the merge algorithm that takes, as input, two sorted sequences representing the input sets, and constructs a sequence representing the output set, be it the union, intersection, or subtraction of the input sets. Incidentally, we have defined these operations so that they modify the contents of the set $A$ involved. Alternatively, we could have defined these functions so that they do not modify $A$ but return a new set instead.

The generic merge algorithm iteratively examines and compares the current elements $a$ and $b$ of the input sequence $A$ and $B$, respectively, and finds out whether $a < b$, $a = b$, or $a > b$. Then, based on the outcome of this comparison, it determines whether it should copy one of the elements $a$ and $b$ to the end of the output sequence $C$. This determination is made based on the particular operation we are performing, be it a union, intersection, or subtraction. For example, in a union operation, we proceed as follows:

- If $a < b$, we copy $a$ to the end of $C$ and advance to the next element of $A$
- If $a = b$, we copy $a$ to the end of $C$ and advance to the next elements of $A$ and $B$
- If $a > b$, we copy $b$ to the end of $C$ and advance to the next element of $B$

### Performance of Generic Merging

Let us analyze the running time of generic merging. At each iteration, we compare two elements of the input sequences $A$ and $B$, possibly copy one element to the output sequence, and advance the current element of $A$, $B$, or both. Assuming that comparing and copying elements takes $O(1)$ time, the total running time is $O(n_A + n_B)$, where $n_A$ is the size of $A$ and $n_B$ is the size of $B$; that is, generic merging takes time proportional to the number of elements. Thus, we have the following:

**Proposition 11.8:** *The set ADT can be implemented with an ordered sequence and a generic merge scheme that supports operations* union, intersect, *and* subtract *in $O(n)$ time, where $n$ denotes the sum of sizes of the sets involved.*

### Generic Merging as a Template Method Pattern

The generic merge algorithm is based on the ***template method pattern*** (see Section 7.3.7). The template method pattern is a software engineering design pattern describing a generic computation mechanism that can be specialized by redefining certain steps. In this case, we describe a method that merges two sequences into one and can be specialized by the behavior of three abstract methods.

Code Fragments 11.9 and 11.10 show the class Merge providing a C++ implementation of the generic merge algorithm. This class has no data members. It defines a public function merge, which merges the two lists $A$ and $B$, and stores the

result in $C$. It provides three virtual functions, fromA, fromB, and fromBoth. These are pure virtual functions (that is, they are not defined here), but are overridden in subclasses of Merge, to achieve a desired effect. The function fromA specifies the action to be taken when the next element to be selected in the merger is from $A$. Similarly, fromB specifies the action when the next element to be selected is from $B$. Finally, fromBoth is the action to be taken when the two elements of $A$ and $B$ are equal, and hence both are to be selected.

```
template <typename E>
class Merge {                                   // generic Merge
public:                                         // global types
  typedef std::list<E> List;                    // list type
  void merge(List& A, List& B, List& C);        // generic merge function
protected:                                      // local types
  typedef typename List::iterator Itor;         // iterator type
                                                // overridden functions
  virtual void fromA(const E& a, List& C) = 0;
  virtual void fromBoth(const E& a, const E& b, List& C) = 0;
  virtual void fromB(const E& b, List& C) = 0;
};
```

**Code Fragment 11.9:** Definition of the class Merge for generic merging.

The function merge, which is presented in Code Fragment 11.10 performs the actual merger. It is structurally similar to the list-based merge procedure given in Code Fragment 11.3. Rather than simply taking an element from list $A$ or list $B$, it invokes one of the virtual functions to perform the appropriate specialized task. The final result is stored in the list $C$.

```
template <typename E>                           // generic merge
void Merge<E>::merge(List& A, List& B, List& C) {
  Itor pa = A.begin();                          // A's elements
  Itor pb = B.begin();                          // B's elements
  while (pa != A.end() && pb != B.end()) {      // main merging loop
    if (*pa < *pb)
      fromA(*pa++, C);                          // take from A
    else if (*pa == *pb)
      fromBoth(*pa++, *pb++, C);                // take from both
    else
      fromB(*pb++, C);                          // take from B
  }
  while (pa != A.end()) { fromA(*pa++, C); }    // take rest from A
  while (pb != B.end()) { fromB(*pb++, C); }    // take rest from B
}
```

**Code Fragment 11.10:** Member function merge which implements generic merging for class Merge.

To convert Merge into a useful class, we provide definitions for the three auxiliary functions, fromA, fromBoth, and fromB. See Code 11.11.

- In class UnionMerge, merge copies every element from $A$ and $B$ into $C$, but does not duplicate any element.
- In class IntersectMerge, merge copies every element that is in both $A$ and $B$ into $C$, but "throws away" elements in one set but not in the other.
- In class SubtractMerge, merge copies every element that is in $A$ and not in $B$ into $C$.

```
template <typename E>                                   // set union
class UnionMerge : public Merge<E> {
protected:
  typedef typename Merge<E>::List List;
  virtual void fromA(const E& a, List& C)
    { C.push_back(a); }                                 // add a
  virtual void fromBoth(const E& a, const E& b, List& C)
    { C.push_back(a); }                                 // add a only
  virtual void fromB(const E& b, List& C)
    { C.push_back(b); }                                 // add b
};

template <typename E>                                   // set intersection
class IntersectMerge : public Merge<E> {
protected:
  typedef typename Merge<E>::List List;
  virtual void fromA(const E& a, List& C)
    { }                                                 // ignore
  virtual void fromBoth(const E& a, const E& b, List& C)
    { C.push_back(a); }                                 // add a only
  virtual void fromB(const E& b, List& C)
    { }                                                 // ignore
};

template <typename E>                                   // set subtraction
class SubtractMerge : public Merge<E> {
protected:
  typedef typename Merge<E>::List List;
  virtual void fromA(const E& a, List& C)
    { C.push_back(a); }                                 // add a
  virtual void fromBoth(const E& a, const E& b, List& C)
    { }                                                 // ignore
  virtual void fromB(const E& b, List& C)
    { }                                                 // ignore
};
```

**Code Fragment 11.11:** Classes extending the Merge class by specializing the auxiliary functions to perform set union, intersection, and subtraction, respectively.

### 11.4.3 Partitions with Union-Find Operations

A ***partition*** is a collection of disjoint sets. We define the functions of the partition ADT using position objects (Section 6.2.1), each of which stores an element $x$. The partition ADT supports the following functions.

makeSet($x$): Create a singleton set containing the element $x$ and return the position storing $x$ in this set.

union($A,B$): Return the set $A \cup B$, destroying the old $A$ and $B$.

find($p$): Return the set containing the element in position $p$.

A simple implementation of a partition with a total of $n$ elements is using a collection of sequences, one for each set, where the sequence for a set $A$ stores set positions as its elements. Each position object stores a variable, *element*, which references its associated element $x$ and allows the execution of the element() function in $O(1)$ time. In addition, we also store a variable, *set*, in each position, which references the sequence storing $p$, since this sequence is representing the set containing $p$'s element. (See Figure 11.16.) Thus, we can perform operation find($p$) in $O(1)$ time, by following the *set* reference for $p$. Likewise, makeSet also takes $O(1)$ time. Operation union($A,B$) requires that we join two sequences into one and update the *set* references of the positions in one of the two. We choose to implement this operation by removing all the positions from the sequence with smaller size, and inserting them in the sequence with larger size. Each time we take a position $p$ from the smaller set $s$ and insert it into the larger set $t$, we update the *set* reference for $p$ to now point to $t$. Hence, the operation union($A,B$) takes time $O(\min(|A|,|B|))$, which is $O(n)$, because, in the worst case, $|A| = |B| = n/2$. Nevertheless, as shown below, an amortized analysis shows this implementation to be much better than appears from this worst-case analysis.

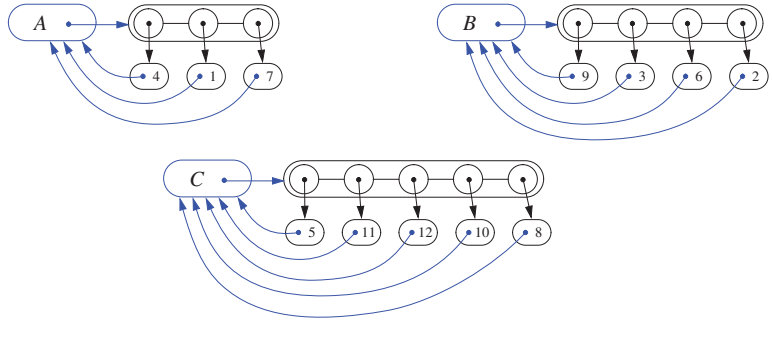



**Figure 11.16:** Sequence-based implementation of a partition consisting of three sets: $A = \{1,4,7\}$, $B = \{2,3,6,9\}$, and $C = \{5,8,10,11,12\}$.

### Performance of the Sequence Implementation

The sequence implementation above is simple, but it is also efficient, as the following theorem shows.

**Proposition 11.9:** *Performing a series of $n$* makeSet, union, *and* find *operations, using the sequence-based implementation above, starting from an initially empty partition takes $O(n \log n)$ time.*

**Justification:** We use the accounting method and assume that one cyber-dollar can pay for the time to perform a find operation, a makeSet operation, or the movement of a position object from one sequence to another in a union operation. In the case of a find or makeSet operation, we charge the operation itself 1 cyber-dollar. In the case of a union operation, however, we charge 1 cyber-dollar to each position that we move from one set to another. Note that we charge nothing to the union operations themselves. Clearly, the total charges to find and makeSet operations add up to $O(n)$.

Consider, then, the number of charges made to positions on behalf of union operations. The important observation is that each time we move a position from one set to another, the size of the new set at least doubles. Thus, each position is moved from one set to another at most $\log n$ times; hence, each position can be charged at most $O(\log n)$ times. Since we assume that the partition is initially empty, there are $O(n)$ different elements referenced in the given series of operations, which implies that the total time for all the union operations is $O(n \log n)$. ■

The amortized running time of an operation in a series of makeSet, union, and find operations, is the total time taken for the series divided by the number of operations. We conclude from the proposition above that, for a partition implemented using sequences, the amortized running time of each operation is $O(\log n)$. Thus, we can summarize the performance of our simple sequence-based partition implementation as follows.

**Proposition 11.10:** *Using a sequence-based implementation of a partition, in a series of $n$* makeSet, union, *and* find *operations starting from an initially empty partition, the amortized running time of each operation is $O(\log n)$.*

Note that in this sequence-based implementation of a partition, each find operation takes worst-case $O(1)$ time. It is the running time of the union operations that is the computational bottleneck.

In the next section, we describe a tree-based implementation of a partition that does not guarantee constant-time find operations, but has amortized time much better than $O(\log n)$ per union operation.

### A Tree-Based Partition Implementation ★

An alternative data structure uses a collection of trees to store the $n$ elements in sets, where each tree is associated with a different set. (See Figure 11.17.) In particular, we implement each tree with a linked data structure whose nodes are themselves the set position objects. We still view each position $p$ as being a node having a variable, *element*, referring to its element $x$, and a variable, *set*, referring to a set containing $x$, as before. But now we also view each position $p$ as being of the "*set*" data type. Thus, the *set* reference of each position $p$ can point to a position, which could even be $p$ itself. Moreover, we implement this approach so that all the positions and their respective *set* references together define a collection of trees.

We associate each tree with a set. For any position $p$, if $p$'s *set* reference points back to $p$, then $p$ is the ***root*** of its tree, and the name of the set containing $p$ is "$p$" (that is, we use position names as set names in this case). Otherwise, the *set* reference for $p$ points to $p$'s parent in its tree. In either case, the set containing $p$ is the one associated with the root of the tree containing $p$.

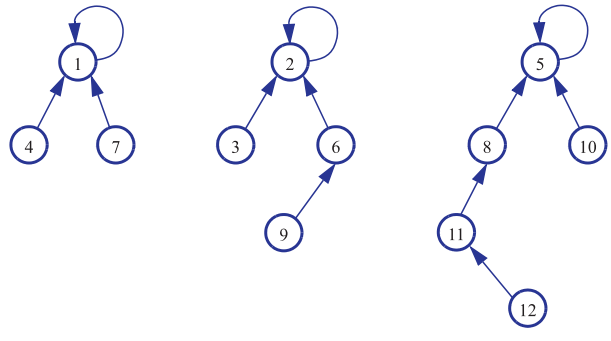



**Figure 11.17:** Tree-based implementation of a partition consisting of three disjoint sets: $A = \{1,4,7\}$, $B = \{2,3,6,9\}$, and $C = \{5,8,10,11,12\}$.

With this partition data structure, operation union$(A,B)$ is called with position arguments $p$ and $q$ that respectively represent the sets $A$ and $B$ (that is, $A = p$ and $B = q$). We perform this operation by making one of the trees a subtree of the other (Figure 11.18b), which can be done in $O(1)$ time by setting the *set* reference of the root of one tree to point to the root of the other tree. Operation find for a position $p$ is performed by walking up to the root of the tree containing the position $p$ (Figure 11.18a), which takes $O(n)$ time in the worst case.

At first, this implementation may seem to be no better than the sequence-based data structure, but we add the following two simple heuristics to make it run faster.

**Union-by-Size:** Store, with each position node $p$, the size of the subtree rooted at $p$. In a union operation, make the tree of the smaller set become a subtree of the other tree, and update the size field of the root of the resulting tree.

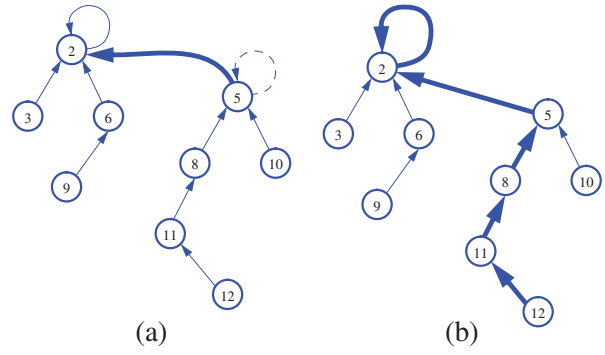



**Figure 11.18:** Tree-based implementation of a partition: (a) operation union$(A,B)$; (b) operation find$(p)$, where $p$ denotes the position object for element 12.

**Path Compression:** In a find operation, for each node $v$ that the find visits, reset the parent pointer from $v$ to point to the root. (See Figure 11.19.)

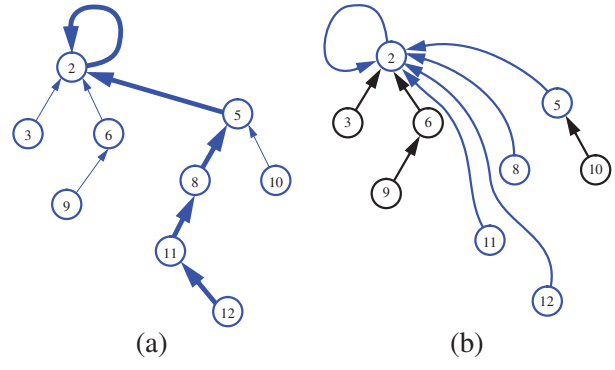



**Figure 11.19:** Path-compression heuristic: (a) path traversed by operation find on element 12; (b) restructured tree.

A surprising property of this data structure, when implemented using the union-by-size and path-compression heuristics, is that performing a series of $n$ union and find operations takes $O(n \log^* n)$ time, where $\log^* n$ is the ***log-star*** function, which is the inverse of the ***tower-of-twos*** function. Intuitively, $\log^* n$ is the number of times that one can iteratively take the logarithm (base 2) of a number before getting a number smaller than 2. Table 11.1 shows a few sample values.

| ***Minimum*** $n$ | 2 | $2^2 = 4$ | $2^{2^2} = 16$ | $2^{2^{2^2}} = 65,536$ | $2^{2^{2^{2^2}}} = 2^{65,536}$ |
|---|---|---|---|---|---|
| $\log^* n$ | 1 | 2 | 3 | 4 | 5 |

**Table 11.1:** Some values of $\log^* n$ and critical values for its inverse.

## 11.5 Selection

There are a number of applications in which we are interested in identifying a single element in terms of its rank relative to an ordering of the entire set. Examples include identifying the minimum and maximum elements, but we may also be interested in, say, identifying the ***median*** element, that is, the element such that half of the other elements are smaller and the remaining half are larger. In general, queries that ask for an element with a given rank are called ***order statistics***.

### Defining the Selection Problem

In this section, we discuss the general order-statistic problem of selecting the $k$th smallest element from an unsorted collection of $n$ comparable elements. This is known as the ***selection*** problem. Of course, we can solve this problem by sorting the collection and then indexing into the sorted sequence at index $k-1$. Using the best comparison-based sorting algorithms, this approach would take $O(n \log n)$ time, which is obviously an overkill for the cases where $k = 1$ or $k = n$ (or even $k = 2$, $k = 3$, $k = n-1$, or $k = n-5$), because we can easily solve the selection problem for these values of $k$ in $O(n)$ time. Thus, a natural question to ask is whether we can achieve an $O(n)$ running time for all values of $k$ (including the interesting case of finding the median, where $k = \lfloor n/2 \rfloor$).

### 11.5.1 Prune-and-Search

This may come as a small surprise, but we can indeed solve the selection problem in $O(n)$ time for any value of $k$. Moreover, the technique we use to achieve this result involves an interesting algorithmic design pattern. This design pattern is known as ***prune-and-search*** or ***decrease-and-conquer***. In applying this design pattern, we solve a given problem that is defined on a collection of $n$ objects by pruning away a fraction of the $n$ objects and recursively solving the smaller problem. When we have finally reduced the problem to one defined on a constant-sized collection of objects, then we solve the problem using some brute-force method. Returning back from all the recursive calls completes the construction. In some cases, we can avoid using recursion, in which case we simply iterate the prune-and-search reduction step until we can apply a brute-force method and stop. Incidentally, the binary search method described in Section 9.3.1 is an example of the prune-and-search design pattern.

### 11.5.2 Randomized Quick-Select

In applying the prune-and-search pattern to the selection problem, we can design a simple and practical method, called ***randomized quick-select***, for finding the $k$th smallest element in an unordered sequence of $n$ elements on which a total order relation is defined. Randomized quick-select runs in $O(n)$ ***expected*** time, taken over all possible random choices made by the algorithm. This expectation does not depend whatsoever on any randomness assumptions about the input distribution. We note though that randomized quick-select runs in $O(n^2)$ time in the ***worst case***. The justification of this is left as an exercise (Exercise R-11.26). We also provide an exercise (Exercise C-11.32) for modifying randomized quick-select to get a ***deterministic*** selection algorithm that runs in $O(n)$ ***worst-case*** time. The existence of this deterministic algorithm is mostly of theoretical interest, however, since the constant factor hidden by the big-Oh notation is relatively large in this case.

Suppose we are given an unsorted sequence $S$ of $n$ comparable elements together with an integer $k \in [1,n]$. At a high level, the quick-select algorithm for finding the $k$th smallest element in $S$ is similar in structure to the randomized quick-sort algorithm described in Section 11.2.1. We pick an element $x$ from $S$ at random and use this as a "pivot" to subdivide $S$ into three subsequences $L$, $E$, and $G$, storing the elements of $S$ less than $x$, equal to $x$, and greater than $x$, respectively. This is the prune step. Then, based on the value of $k$, we determine which of these sets to recur on. Randomized quick-select is described in Code Fragment 11.12.

**Algorithm** quickSelect$(S,k)$:
*Input:* Sequence $S$ of $n$ comparable elements, and an integer $k \in [1,n]$
*Output:* The $k$th smallest element of $S$

**if** $n = 1$ **then**
  **return** the (first) element of $S$.
pick a random (pivot) element $x$ of $S$ and divide $S$ into three sequences:
  - $L$, storing the elements in $S$ less than $x$
  - $E$, storing the elements in $S$ equal to $x$
  - $G$, storing the elements in $S$ greater than $x$.

**if** $k \leq |L|$ **then**
  quickSelect$(L,k)$
**else if** $k \leq |L| + |E|$ **then**
  **return** $x$  {each element in $E$ is equal to $x$}
**else**
  quickSelect$(G, k - |L| - |E|)$  {note the new selection parameter}

**Code Fragment 11.12:** Randomized quick-select algorithm.

### 11.5.3 Analyzing Randomized Quick-Select

Showing that randomized quick-select runs in $O(n)$ time requires a simple probabilistic argument. The argument is based on the ***linearity of expectation***, which states that if $X$ and $Y$ are random variables and $c$ is a number, then

$$E(X+Y) = E(X) + E(Y) \qquad \text{and} \qquad E(cX) = cE(X),$$

where we use $E(\mathcal{Z})$ to denote the expected value of the expression $\mathcal{Z}$.

Let $t(n)$ be the running time of randomized quick-select on a sequence of size $n$. Since this algorithm depends on random events, its running time, $t(n)$, is a random variable. We want to bound $E(t(n))$, the expected value of $t(n)$. Say that a recursive invocation of our algorithm is "good" if it partitions $S$ so that the size of $L$ and $G$ is at most $3n/4$. Clearly, a recursive call is good with probability $1/2$. Let $g(n)$ denote the number of consecutive recursive calls we make, including the present one, before we get a good one. Then we can characterize $t(n)$ using the following ***recurrence equation***

$$t(n) \leq bn \cdot g(n) + t(3n/4),$$

where $b \geq 1$ is a constant. Applying the linearity of expectation for $n > 1$, we get

$$E\left(t(n)\right) \leq E\left(bn \cdot g(n) + t(3n/4)\right) = bn \cdot E\left(g(n)\right) + E\left(t(3n/4)\right).$$

Since a recursive call is good with probability $1/2$, and whether a recursive call is good or not is independent on its parent call being good, the expected value of $g(n)$ is the same as the expected number of times we must flip a fair coin before it comes up "heads." That is, $E(g(n)) = 2$. Thus, if we let $T(n)$ be shorthand for $E(t(n))$, then we can write the case for $n > 1$ as

$$T(n) \leq T(3n/4) + 2bn.$$

To convert this relation into a closed form, let us iteratively apply this inequality assuming $n$ is large. So, for example, after two applications,

$$T(n) \leq T((3/4)^2 n) + 2b(3/4)n + 2bn.$$

At this point, we should see that the general case is

$$T(n) \leq 2bn \cdot \sum_{i=0}^{\lceil \log_{4/3} n \rceil} (3/4)^i.$$

In other words, the expected running time is at most $2bn$ times a geometric sum whose base is a positive number less than 1. Thus, by Proposition 4.5, $T(n)$ is $O(n)$.

**Proposition 11.11:** *The expected running time of randomized quick-select on a sequence $S$ of size $n$ is $O(n)$, assuming two elements of $S$ can be compared in $O(1)$ time.*

## 11.6 Exercises

For help with exercises, please visit the web site, www.wiley.com/college/goodrich.

### Reinforcement

R-11.1 What is the best algorithm for sorting each of the following: general comparable objects, long character strings, double-precision floating point numbers, 32-bit integers, and bytes? Justify your answer.

R-11.2 Suppose $S$ is a list of $n$ bits, that is, $n$ 0's and 1's. How long will it take to sort $S$ with the merge-sort algorithm? What about quick-sort?

R-11.3 Suppose $S$ is a list of $n$ bits, that is, $n$ 0's and 1's. How long will it take to sort $S$ stably with the bucket-sort algorithm?

R-11.4 Give a complete justification of Proposition 11.1.

R-11.5 In the merge-sort tree shown in Figures 11.2 through 11.4, some edges are drawn as arrows. What is the meaning of a downward arrow? How about an upward arrow?

R-11.6 Give a complete pseudo-code description of the recursive merge-sort algorithm that takes an array as its input and output.

R-11.7 Show that the running time of the merge-sort algorithm on an $n$-element sequence is $O(n \log n)$, even when $n$ is not a power of 2.

R-11.8 Suppose we are given two $n$-element sorted sequences $A$ and $B$ that should not be viewed as sets (that is, $A$ and $B$ may contain duplicate entries). Describe an $O(n)$-time method for computing a sequence representing the set $A \cup B$ (with no duplicates).

R-11.9 Show that $(X - A) \cup (X - B) = X - (A \cap B)$, for any three sets $X$, $A$, and $B$.

R-11.10 Suppose we modify the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an $n$-element sequence as the pivot, we choose the element at index $\lfloor n/2 \rfloor$. What is the running time of this version of quick-sort on a sequence that is already sorted?

R-11.11 Consider a modification of the deterministic version of the quick-sort algorithm where we choose the element at index $\lfloor n/2 \rfloor$ as our pivot. Describe the kind of sequence that would cause this version of quick-sort to run in $\Omega(n^2)$ time.

R-11.12 Show that the best-case running time of quick-sort on a sequence of size $n$ with distinct elements is $O(n \log n)$.

R-11.13 Describe a randomized version of in-place quick-sort in pseudo-code.

R-11.14 Show that the probability that any given input element $x$ belongs to more than $2\log n$ subproblems in size group $i$, for randomized quick-sort, is at most $1/n^2$.

R-11.15 Suppose algorithm inPlaceQuickSort (Code Fragment 11.6) is executed on a sequence with duplicate elements. Show that the algorithm still correctly sorts the input sequence, but the result of the divide step may differ from the high-level description given in Section 11.2, and may result in inefficiencies. In particular, what happens in the partition step when there are elements equal to the pivot? Is the sequence $E$ (storing the elements equal to the pivot) actually computed? Does the algorithm recur on the subsequences $L$ and $G$, or on some other subsequences? What is the running time of the algorithm if all the input elements are equal?

R-11.16 Of the $n!$ possible inputs to a given comparison-based sorting algorithm, what is the absolute maximum number of inputs that could be sorted with just $n$ comparisons?

R-11.17 Bella has a comparison-based sorting algorithm that sorts the first $k$ elements in sequence of size $n$ in $O(n)$ time. Give a big-Oh characterization of the biggest that $k$ can be?

R-11.18 Is the merge-sort algorithm in Section 11.1 stable? Why or why not?

R-11.19 An algorithm that sorts key-value entries by key is said to be ***straggling*** if, any time two entries $e_i$ and $e_j$ have equal keys, but $e_i$ appears before $e_j$ in the input, then the algorithm places $e_i$ after $e_j$ in the output. Describe a change to the merge-sort algorithm in Section 11.1 to make it straggling.

R-11.20 Describe a radix-sort method for lexicographically sorting a sequence $S$ of triplets $(k,l,m)$, where $k$, $l$, and $m$ are integers in the range $[0,N-1]$, for some $N \geq 2$. How could this scheme be extended to sequences of $d$-tuples $(k_1,k_2,\ldots,k_d)$, where each $k_i$ is an integer in the range $[0,N-1]$?

R-11.21 Is the bucket-sort algorithm in-place? Why or why not?

R-11.22 Give an example input list that requires merge-sort and heap-sort to take $O(n\log n)$ time to sort, but insertion-sort runs in $O(n)$ time. What if you reverse this list?

R-11.23 Describe, in pseudo-code, how to perform path compression on a path of length $h$ in $O(h)$ time in a tree-based partition union/find structure.

R-11.24 Edward claims he has a fast way to do path compression in a partition structure, starting at a node $v$. He puts $v$ into a list $L$, and starts following parent pointers. Each time he encounters a new node, $u$, he adds $u$ to $L$ and updates the parent pointer of each node in $L$ to point to $u$'s parent. Show that Edward's algorithm runs in $\Omega(h^2)$ time on a path of length $h$.

R-11.25 Describe an in-place version of the quick-select algorithm in pseudo-code.

R-11.26 Show that the worst-case running time of quick-select on an $n$-element sequence is $\Omega(n^2)$.

## Creativity

C-11.1 Describe an efficient algorithm for converting a dictionary, $D$, implemented with a linked list, into a map, $M$, implemented with a linked list, so that each key in $D$ has an entry in $M$, and the relative order of entries in $M$ is the same as their relative order in $D$.

C-11.2 Linda claims to have an algorithm that takes an input sequence $S$ and produces an output sequence $T$ that is a sorting of the $n$ elements in $S$.

a. Give an algorithm, isSorted, for testing in $O(n)$ time if $T$ is sorted.
b. Explain why the algorithm isSorted is not sufficient to prove a particular output $T$ of Linda's algorithm is a sorting of $S$.
c. Describe what additional information Linda's algorithm could output so that her algorithm's correctness could be established on any given $S$ and $T$ in $O(n)$ time.

C-11.3 Given two sets $A$ and $B$ represented as sorted sequences, describe an efficient algorithm for computing $A \oplus B$, which is the set of elements that are in $A$ or $B$, but not in both.

C-11.4 Suppose that we represent sets with balanced search trees. Describe and analyze algorithms for each of the functions in the set ADT, assuming that one of the two sets is much smaller than the other.

C-11.5 Describe and analyze an efficient function for removing all duplicates from a collection $A$ of $n$ elements.

C-11.6 Consider sets whose elements are integers in the range $[0, N-1]$. A popular scheme for representing a set $A$ of this type is by means of a Boolean array, $B$, where we say that $x$ is in $A$ if and only if $B[x] =$ **true**. Since each cell of $B$ can be represented with a single bit, $B$ is sometimes referred to as a ***bit vector***. Describe and analyze efficient algorithms for performing the functions of the set ADT assuming this representation.

C-11.7 Consider a version of deterministic quick-sort where we pick the median of the $d$ last elements in the input sequence of $n$ elements as our pivot, for a fixed, constant odd number $d \geq 3$. What is the asymptotic worst-case running time of quick-sort in this case?

C-11.8 Another way to analyze randomized quick-sort is to use a ***recurrence equation***. In this case, we let $T(n)$ denote the expected running time of randomized quick-sort, and we observe that, because of the worst-case partitions for good and bad splits, we can write

$$T(n) \leq \frac{1}{2}(T(3n/4) + T(n/4)) + \frac{1}{2}(T(n-1)) + bn,$$

where $bn$ is the time needed to partition a list for a given pivot and concatenate the result sublists after the recursive calls return. Show, by induction, that $T(n)$ is $O(n \log n)$.

C-11.9 Modify inPlaceQuickSort (Code Fragment 11.6) to handle the general case efficiently when the input sequence, $S$, may have duplicate keys.

C-11.10 Describe a nonrecursive, in-place version of the quick-sort algorithm. The algorithm should still be based on the same divide-and-conquer approach, but use an explicit stack to process subproblems.

C-11.11 An ***inverted file*** is a critical data structure for implementing a search engine or the index of a book. Given a document $D$, which can be viewed as an unordered, numbered list of words, an inverted file is an ordered list of words, $L$, such that, for each $w$ word in $L$, we store the indices of the places in $D$ where $w$ appears. Design an efficient algorithm for constructing $L$ from $D$.

C-11.12 Given an array $A$ of $n$ entries with keys equal to 0 or 1, describe an in-place function for ordering $A$ so that all the 0's are before every 1.

C-11.13 Suppose we are given an $n$-element sequence $S$ such that each element in $S$ represents a different vote for president, where each vote is given as an integer representing a particular candidate. Design an $O(n \log n)$-time algorithm to see who wins the election $S$ represents, assuming the candidate with the most votes wins (even if there are $O(n)$ candidates).

C-11.14 Consider the voting problem from Exercise C-11.13, but now suppose that we know the number $k < n$ of candidates running. Describe an $O(n \log k)$-time algorithm for determining who wins the election.

C-11.15 Consider the voting problem from Exercise C-11.13, but now suppose a candidate wins only if he or she gets a majority of the votes cast. Design and analyze a fast algorithm for determining the winner if there is one.

C-11.16 Show that any comparison-based sorting algorithm can be made to be stable without affecting its asymptotic running time.

C-11.17 Suppose we are given two sequences $A$ and $B$ of $n$ elements, possibly containing duplicates, on which a total order relation is defined. Describe an efficient algorithm for determining if $A$ and $B$ contain the same set of elements. What is the running time of this method?

C-11.18 Given an array $A$ of $n$ integers in the range $[0, n^2 - 1]$, describe a simple function for sorting $A$ in $O(n)$ time.

C-11.19 Let $S_1, S_2, \ldots, S_k$ be $k$ different sequences whose elements have integer keys in the range $[0, N-1]$, for some parameter $N \geq 2$. Describe an algorithm running in $O(n+N)$ time for sorting all the sequences (not as a union), where $n$ denotes the total size of all the sequences.

C-11.20 Given a sequence $S$ of $n$ elements, on which a total order relation is defined, describe an efficient function for determining whether there are two equal elements in $S$. What is the running time of your function?

C-11.21 Let $S$ be a sequence of $n$ elements on which a total order relation is defined. Recall that an ***inversion*** in $S$ is a pair of elements $x$ and $y$ such that $x$ appears before $y$ in $S$ but $x > y$. Describe an algorithm running in $O(n \log n)$ time for determining the ***number*** of inversions in $S$.

C-11.22 Let $S$ be a random permutation of $n$ distinct integers. Argue that the expected running time of insertion-sort on $S$ is $\Omega(n^2)$.
(Hint: Note that half of the elements ranked in the top half of a sorted version of $S$ are expected to be in the first half of $S$.)

C-11.23 Let $A$ and $B$ be two sequences of $n$ integers each. Given an integer $m$, describe an $O(n \log n)$-time algorithm for determining if there is an integer $a$ in $A$ and an integer $b$ in $B$ such that $m = a + b$.

C-11.24 Given a set of $n$ integers, describe and analyze a fast method for finding the $\lceil \log n \rceil$ integers closest to the median.

C-11.25 James has a set $A$ of $n$ nuts and a set $B$ of $n$ bolts, such that each nut in $A$ has a unique matching bolt in $B$. Unfortunately, the nuts in $A$ all look the same, and the bolts in $B$ all look the same as well. The only kind of a comparison that Bob can make is to take a nut-bolt pair $(a,b)$, such that $a$ is in $A$ and $b$ is in $B$, and test it to see if the threads of $a$ are larger, smaller, or a perfect match with the threads of $b$. Describe and analyze an efficient algorithm for Bob to match up all of his nuts and bolts.

C-11.26 Show how to use a deterministic $O(n)$-time selection algorithm to sort a sequence of $n$ elements in $O(n \log n)$ ***worst-case*** time.

C-11.27 Given an unsorted sequence $S$ of $n$ comparable elements, and an integer $k$, give an $O(n \log k)$ expected-time algorithm for finding the $O(k)$ elements that have rank $\lceil n/k \rceil$, $2\lceil n/k \rceil$, $3\lceil n/k \rceil$, and so on.

C-11.28 Let $S$ be a sequence of $n$ insert and removeMin operations, where all the keys involved are integers in the range $[0, n-1]$. Describe an algorithm running in $O(n \log^* n)$ for determining the answer to each removeMin.

C-11.29 Space aliens have given us a program, alienSplit, that can take a sequence $S$ of $n$ integers and partition $S$ in $O(n)$ time into sequences $S_1, S_2, \ldots, S_k$ of size at most $\lceil n/k \rceil$ each, such that the elements in $S_i$ are less than or equal to every element in $S_{i+1}$, for $i = 1, 2, \ldots, k-1$, for a fixed number, $k < n$. Show how to use alienSplit to sort $S$ in $O(n \log n / \log k)$ time.

C-11.30 Karen has a new way to do path compression in a tree-based union/find partition data structure starting at a node $v$. She puts all the nodes that are on the path from $v$ to the root in a set $S$. Then she scans through $S$ and sets the parent pointer of each node in $S$ to its parent's parent pointer (recall

that the parent pointer of the root points to itself). If this pass changed the value of any node's parent pointer, then she repeats this process, and goes on repeating this process until she makes a scan through $S$ that does not change any node's parent value. Show that Karen's algorithm is correct and analyze its running time for a path of length $h$.

C-11.31 Let $S$ be a sequence of $n$ integers. Describe a method for printing out all the pairs of inversions in $S$ in $O(n+k)$ time, where $k$ is the number of such inversions.

C-11.32 This problem deals with modification of the quick-select algorithm to make it deterministic yet still run in $O(n)$ time on an $n$-element sequence. The idea is to modify the way we choose the pivot so that it is chosen deterministically, not randomly, as follows:

> Partition the set $S$ into $\lceil n/5 \rceil$ groups of size 5 each (except possibly for one group). Sort each little set and identify the median element in this set. From this set of $\lceil n/5 \rceil$ "baby" medians, apply the selection algorithm recursively to find the median of the baby medians. Use this element as the pivot and proceed as in the quick-select algorithm.

Show that this deterministic method runs in $O(n)$ time by answering the following questions (please ignore floor and ceiling functions if that simplifies the mathematics, for the asymptotics are the same either way):

a. How many baby medians are less than or equal to the chosen pivot? How many are greater than or equal to the pivot?

b. For each baby median less than or equal to the pivot, how many other elements are less than or equal to the pivot? Is the same true for those greater than or equal to the pivot?

c. Argue why the method for finding the deterministic pivot and using it to partition $S$ takes $O(n)$ time.

d. Based on these estimates, write a recurrence equation to bound the worst-case running time $t(n)$ for this selection algorithm (note that in the worst case there are two recursive calls—one to find the median of the baby medians and one to recur on the larger of $L$ and $G$).

e. Using this recurrence equation, show by induction that $t(n)$ is $O(n)$.

## Projects

P-11.1 Design and implement two versions of the bucket-sort algorithm in C++, one for sorting an array of **char** values and one for sorting an array of **short** values. Experimentally compare the performance of your implementations with the sorting algorithm of the Standard Template Library.

P-11.2 Experimentally compare the performance of in-place quick-sort and a version of quick-sort that is not in-place.

P-11.3 Design and implement a version of the bucket-sort algorithm for sorting a linked list of $n$ entries (for instance, a list of type std::list<int>) with integer keys taken from the range $[0, N-1]$, for $N \geq 2$. The algorithm should run in $O(n+N)$ time.

P-11.4 Implement merge-sort and deterministic quick-sort and perform a series of benchmarking tests to see which one is faster. Your tests should include sequences that are "random" as well as "almost" sorted.

P-11.5 Implement deterministic and randomized versions of the quick-sort algorithm and perform a series of benchmarking tests to see which one is faster. Your tests should include sequences that are very "random" looking as well as ones that are "almost" sorted.

P-11.6 Implement an in-place version of insertion-sort and an in-place version of quick-sort. Perform benchmarking tests to determine the range of values of $n$ where quick-sort is on average better than insertion-sort.

P-11.7 Design and implement an animation for one of the sorting algorithms described in this chapter. Your animation should illustrate the key properties of this algorithm in an intuitive manner.

P-11.8 Implement the randomized quick-sort and quick-select algorithms, and design a series of experiments to test their relative speeds.

P-11.9 Implement an extended set ADT that includes the functions union($B$), intersect($B$), subtract($B$), size(), empty(), plus the functions equals($B$), contains($e$), insert($e$), and remove($e$) with obvious meaning.

P-11.10 Implement the tree-based union/find partition data structure with both the union-by-size and path-compression heuristics.

## Chapter Notes

Knuth's classic text on *Sorting and Searching* [60] contains an extensive history of the sorting problem and algorithms for solving it. Huang and Langston [48] show how to merge two sorted lists in-place in linear time. Our set ADT is derived from that of Aho, Hopcroft, and Ullman [5]. The standard quick-sort algorithm is due to Hoare [45]. More information about randomization, including Chernoff bounds, can be found in the appendix and the book by Motwani and Raghavan [80]. The quick-sort analysis given in this chapter is a combination of the analysis given in a previous edition of this book and the analysis of Kleinberg and Tardos [55]. Exercise C-11.8 is due to Littman. Gonnet and Baeza-Yates [37] analyze and experimentally compare several sorting algorithms. The term "prune-and-search" comes originally from the computational geometry literature (such as in the work of Clarkson [21] and Megiddo [71, 72]). The term "decrease-and-conquer" is from Levitin [65].