Chapter 4

Comparison Sort Algorithms

Sorting algorithms are among the most important in computer science because they are so common in practice. To sort a sequence of values is to arrange the elements of the sequence into some order. If L is a list of n values $\langle l_1, l_2, \ldots, l_n \rangle$ then to sort the list is to reorder or permute the values into the sequence $\langle l'_1, l'_2, \ldots, l'_n \rangle$ such that $l'_1 \leq l'_2 \leq \ldots \leq l'_n$. The n values in the sequences could be integer or real numbers for which the $\leq$ operation is provided by the C++ language. Or, they could be values of any class that provides such a comparison. For example, the values could be objects with class `Rational` from the Appendix (page 491), because that class provides both the `<` and the `==` operators for rational number comparisons.

Most sort algorithms are comparison sorts, which are based on successive pair-wise comparisons of the n values. At each step in the algorithm, l_i is compared with l_j, and depending on whether $l_i \leq l_j$ the elements are rearranged one way or another. This chapter presents a unified description of five comparison sorts—merge sort, quick sort, insertion sort, selection sort, and heap sort. When the values to be sorted take a special form, however, it is possible to sort them more efficiently without comparing the values pair-wise. For example, the counting sort algorithm is more efficient than the sorts described in this chapter. But, it requires each of the n input elements to an integer in a fixed range. Comparison sort algorithms have no such restrictions.

4.1 The Merritt Sort Taxonomy

The idea behind the Merritt sort taxonomy is to sort a large list assuming you can recursively sort a smaller part of the list. Figure 4.1 shows the general approach. Suppose you have a list of elements, L. To sort the list, you split it into two sublists, $L1$ and $L2$. The sublists are each smaller than the original list, L. The recursive idea lets you assume that you have the solution to the problem of sorting the smaller lists. So, recursively sort $L1$, producing the sorted sublist $L1'$. Then, recursively sort $L2$, producing the sorted sublist $L2'$. The last step is to join the two sorted sublists, $L1'$ and $L2'$, into the final sorted list, L'.

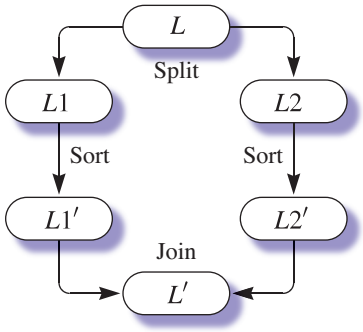


Figure 4.1 The general approach to sorting with the Merritt sort taxonomy.

Merge sort and quick sort

There are two basic families of sort algorithms, which differ in the methods they use to perform the split and the join. The two are the quick sort algorithm and the merge sort algorithm, shown in Figure 4.2. The classification of sort algorithms into these two families is known as the Merritt taxonomy after Susan Merritt, who proposed it in 1985.[1]

The figure shows the original unsorted list, L, as the eight values

7 3 1 6 2 8 5 4

for both algorithms. The final list for both is

1 2 3 4 5 6 7 8

which is the sorted list, L'.

The merge sort algorithm performs a simple split. It takes $L1$ as the first half of the list

7 3 1 6

and $L2$ as the second half of the list

2 8 5 4

It recursively sorts the sublists, producing the sorted sublist $L1'$ as

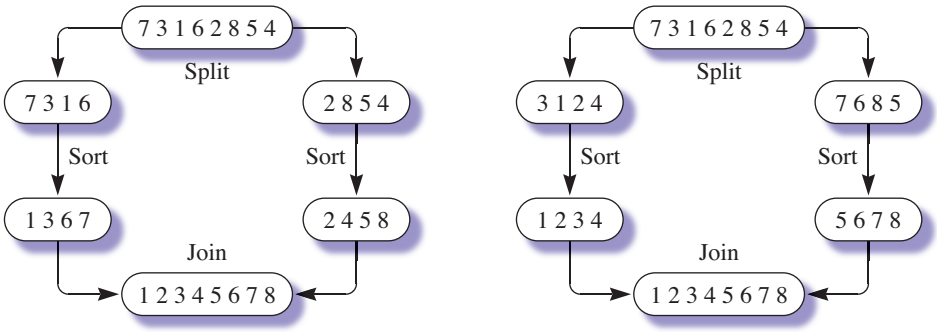
1 3 6 7

and the sorted sublist $L2'$ as

2 4 5 8

The last step is to merge these two sublists into a single sorted list, L'. You can see that the split of L into $L1$ and $L2$ is easy. You simply take the left half of L as $L1$ and the right half as $L2$. On the other hand, the join is hard. It requires a loop to cycle through the sublists, selecting the smallest number at each step to place in the merged list.

[1] S. M. Merritt and K. K. Lau, "A Logical Inverted Taxonomy of Sorting Algorithms," *Communications of the ACM*, Volume 28 Issue 1, Jan. 1985.



(a) The merge sort algorithm. **(b)** The quick sort algorithm.

Figure 4.2 The merge sort and quick sort algorithms with the Merritt sort taxonomy.

The quick sort algorithm splits the original list, L, such that every element in the sublist $L1$ is at most the median value, and every element in the sublist $L2$ is at least the median value. It follows that every element in $L1$ will be less than or equal to every element in $L2$. The sublist $L1$ is

3 1 2 4

and the sublist $L2$ is

7 6 8 5

The algorithm sorts $L1$ recursively into the list $L1'$

1 2 3 4

and $L2$ recursively into the list $L2'$

5 6 7 8

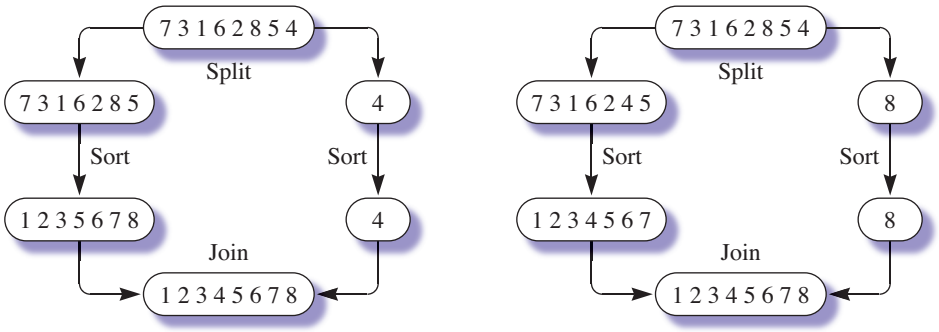
Then it joins $L1'$ and $L2'$ into the final sorted list, L'. You can see that the split of L into $L1$ and $L2$ is hard. It requires a loop that somehow compares the elements in the list with each other and moves the smaller elements to the left and the larger ones to the right. On the other hand, the join is easy. It does not require any further comparisons in a loop, the way the join in the merge sort does.

Insertion sort and selection sort

Figure 4.3 shows the special cases of the merge and quick sorts when the split of n elements subdivides the list such that $L1$ has $n-1$ elements and $L2$ has one element.

When $L2$ has a single element, the merge sort algorithm simply picks the right most element in the list during the split operation. In Figure 4.3(a), the right most element is 4 which is split off into $L2$. $L1$ is the sublist

7 3 1 6 2 8 5



(a) The insertion sort algorithm. **(b)** The selection sort algorithm.

Figure 4.3 The insertion sort and selection sort algorithms with the Merritt sort taxonomy.

The algorithm recursively sorts the sublist $L1$ into

1 2 3 5 6 7 8

but does not need to sort $L2$ because $L2$ has only one element. Then it joins $L1'$ and $L2'$ by inserting the single element from $L2'$ into $L1'$. The insertion process requires a simple loop to shift the lower elements down one slot to make room for the element from $L2'$. The merge sort with a split of one element is called the insertion sort. The insert operation is really a merge of two lists where one of the lists has a single element.

When $L2$ has a single element, the quick sort algorithm must select the largest value from L to put in $L2$. Figure 4.3(b) shows that the largest element from the original list is 8. After it is selected from the original list, $L1$ is left as

7 3 1 6 2 4 5

The selection process requires a simple loop to find the index of the largest value. After the index is computed, an exchange puts the largest value in $L2$. In this example, the largest value 8 is exchanged with 4. The algorithm sorts the sublist $L1$ into

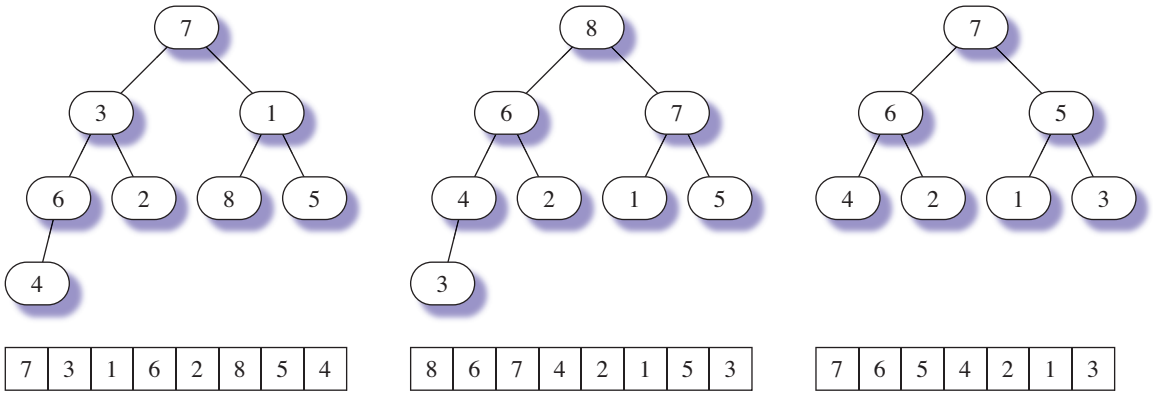
1 2 3 4 5 6 7

but it does not need to sort $L2$ because $L2$ has only one element. The quick sort with a split of one element is called the selection sort.

Heap sort

The heap sort is like the selection sort because it is a specialization of the quick sort with a split of one element. It differs from the selection sort in the way that it splits off its single element. Furthermore, the heap sort differs from the other comparison sorts because it requires a preprocessing step that the other sort algorithms do not require. The preprocessing step rearranges the original list of values into a special order that makes them satisfy what is known as the max-heap property. One characteristic of a max-heap is that its first value is the largest value in the list. That characteristic makes it easy to select the largest value, because it will always be the first one.

A max-heap is a binary tree with two properties:



(a) The initial list is not a heap. **(b)** List *L* after build heap. **(c)** List *L*1 after the split.

Figure 4.4 The equivalence between a list and a binary tree.

- It has a max-heap shape.
- It has a max-heap order.

Figure 4.4 shows three binary trees that all have a max-heap shape, but only two of which have a max-heap order.

The definition of a *max-heap shape* is based on the equivalence of a list and a binary tree. Given a list of values, write the first value as the root of the tree. Write the next two values as the children of the root. Write the next four values as the grandchildren of the root. Continue for each generation, writing the values from left to right and doubling the number of offspring at each level until you run out of values. Figure 4.4 shows the equivalence of three lists and their binary trees. If you delete the leaf 3 in Figure 4.4(c), the binary tree will still have a max-heap shape. However, if you delete the leaf 2 it will not. You cannot have any gaps in a row of leaves like the gap you would have between leaves 4 and 1 if you were to delete leaf 2. Not having gaps between leaves is equivalent to not having gaps in the list of values to which the binary tree corresponds. A tree with the max-heap shape is known as a *complete* binary tree. Notice that the list in Figure 4.4(a) is the initial unsorted list in Figures 4.2 and 4.3.

The definition of a *max-heap order* is that for every node other than the root, the value of the node is at most the value of its parent. This property cannot hold for the root, because the root has no parent. It follows from this definition that the root must contain the maximum of all the values in the tree. In Figure 4.4(a), nodes 3 and 1 satisfy the max-heap property because they are both less than their parent 7. But, node 6 does not because it is greater than its parent 3. Node 8 is another one that fails the max-heap order property because it is greater than its parent 1. Because there are some nodes that fail to satisfy the max-heap order property, the binary tree in Figure 4.4(a) is not a max-heap. In Figure 4.4(b), you can verify that all nodes in the tree other than the root satisfy the max-heap order property. For example, 6 now satisfies the max-heap order property because it is less than its parent 8. You can also verify that the tree in Figure 4.4(c) satisfies the max-heap order property.

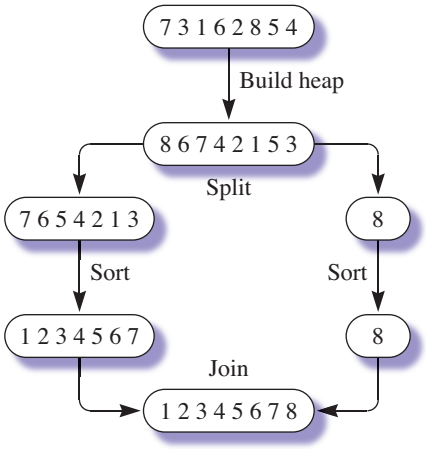


Figure 4.5 The heap sort algorithm with the Merritt sort taxonomy.

Figure 4.5 shows the heap sort algorithm with the Merritt sort taxonomy acting on the original list of numbers

7 3 1 6 2 8 5 4

The preprocessing step labeled "Build heap" rearranges these values to make the following max-heap

8 6 7 4 2 1 5 3

which, like all max-heaps has the largest value first. The algorithm splits the 8 from the first position of the list in such a way as to maintain $L1$ as a max-heap. In Figure 4.1, $L1$ is the list

7 6 5 4 2 1 3

which is also the list in Figure 4.4(c). The algorithm works by selecting the maximum value 8 from the root of the tree in Figure 4.4(b). Then, it deletes the last node 3 from the heap, which maintains the max-heap shape. It puts the 3 at the now-vacant root. But now the max-heap order property is violated. So, it does an adjustment called a *sift* to restore the max-heap order. The result is the max-heap of Figure 4.4(c).

There is no longer a need for the preprocessing step in this part of the algorithm. That is, when the algorithm recursively sorts $L1$ into $L1'$, it does not need to do a complete Build heap operation on $L1$, because $L1$ is already a max-heap. The next section describes the algorithms for building the initial heap and splitting off the root in such a way as to maintain the max-heap property of $L1$.

4.2 The Template Method Pattern

It would be possible to implement the sorting algorithms of this chapter in isolation. One goal of this book, however, is to teach object-oriented design patterns with classical

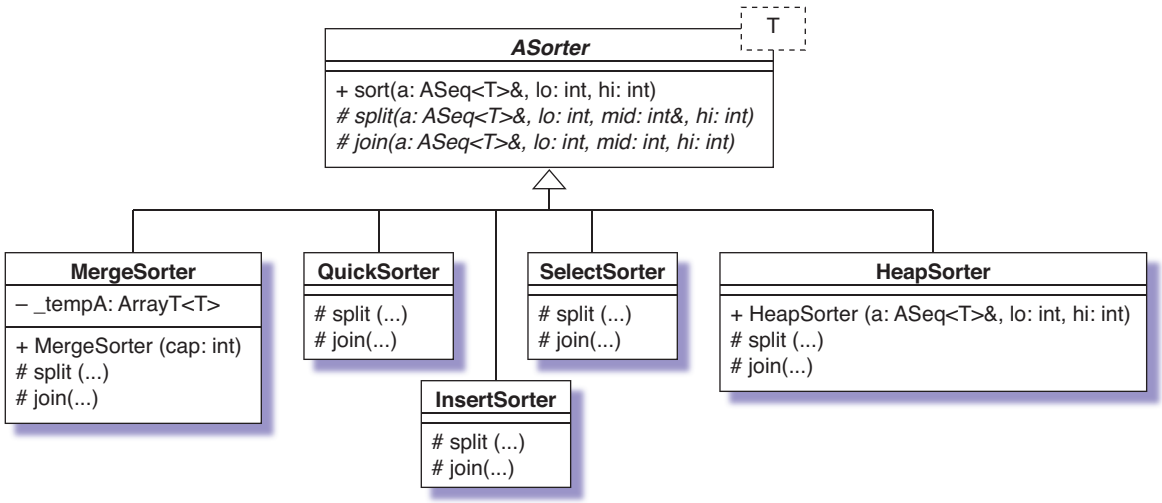


Figure 4.6 A UML class diagram of the Template Method pattern applied to the Merritt sort taxonomy.

data structures. Accordingly, this chapter presents the comparison sort algorithms in the context of the Template Method object-oriented pattern.

The Template Method pattern is used when a system must implement an algorithm with an invariant part, which always executes the same way, but which depends on variant parts that are implemented differently depending on the circumstance. There are two kinds of methods in the Template Method pattern:

- The template method
- Primitive operations

You implement the invariant part in the template method in the superclass. You implement the variant parts in the primitive operations in the subclass. The superclass defines the signatures of the primitive operations in abstract methods and calls the primitive operations in its implementation of the template method.

Abstract sorter class

With the Merritt sort taxonomy, the invariant part of the algorithm is Figure 4.1, which is the method of splitting a list, sorting the left sublist, sorting the right sublist, and joining the sublists to produce the final sorted list. All the comparative sort algorithms described in the previous section have that same invariant part as the template method. Each kind of sort implements the two primitive operations of split and join.

Figure 4.6 shows the UML diagram of the Template Method pattern applied to the Merritt sort taxonomy. The superclass is named `ASorter`, which stands for abstract sorter. It implements the public template method `sort()`, which is not abstract. It also defines the signatures of the protected primitive operations `split()` and `join()`. The UML notation for protected attributes is #.

Do not confuse the C++ template feature with the object-oriented design pattern known as the Template Method pattern. They are not the same. It could appear especially confusing in this application because the program happens to use the C++ template feature as well as the Template Method design pattern. The array to be sorted is of type `aSeq<T>`, where `<T>` is the C++ template type and `aSeq` is the abstract sequence template as described in Chapter 2. The Template Method pattern is the object-oriented technique of having the `ASorter` superclass implement the invariant part of the sort algorithm in the `sort()` method, and define the abstract methods of `split()` and `join()`, which are implemented by the various sorters in the subclasses.

Three of the concrete sorters, `QuickSorter`, `InsertSorter`, and `SelectSorter`, do nothing more than implement the split and join operations defined by their superclass. The ellipses (...) in their parameter lists in the UML diagram signify the same parameter lists as the corresponding ones in `ASorter`.

The class diagram for `MergeSorter` shows an additional private temporary array named `_tempA`. You cannot merge two sorted parts of a single array in place. You must either copy the two parts into a temporary array and then merge them back in order into the original array, or you must merge the two parts in order into a temporary array and then copy the temporary array back into the original array. `_tempA` is the temporary array for the merge operation. `MergeSorter` also has a constructor whose job is to allocate storage for `_tempA`.

Figure 4.6 shows that `HeapSorter` also has a constructor. Its job is to perform the Build heap operation shown in Figure 4.5. The constructor only executes one time when the sorter is created, which is precisely when the Build heap operation is required.

Implementation of ASorter

Figure 4.7 shows the implementation of the abstract sorter `ASorter` in the file named `ASorter.hpp`. `ASorter` implements the template method `sort()`. This is one case where it is appropriate to implement a method in a `.hpp` file. `sort()` is a void function that takes a safe template array `a` called by reference and two integer parameters `lo` and `hi` called by value.

For the sake of brevity in the following mathematical expressions, let a stand for array `a`, l stand for `lo`, m stand for `mid`, and h stand for `hi`. The precondition for `sort()` as shown in the documentation of `ASorter` in Figure 4.7 is

$$0 \leq l \wedge h < a.cap()$$

and the postcondition is the predicate

$$\text{sorted}(a[l..h])$$

which is an abbreviation for

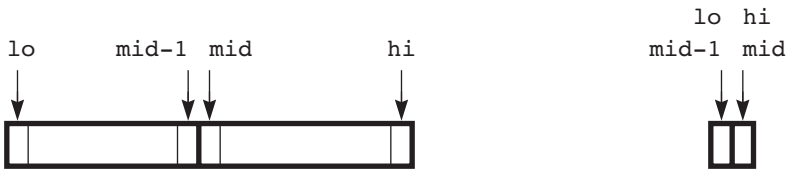
$$(\forall i \mid l \leq i < h : a[i] \leq a[i+1]).$$

Recall from Chapter 2 that `a.cap()` is a method that returns the capacity of array `a`. The postcondition states that the array is sorted in increasing order between `a[lo]` and `a[hi]`. So, `sort()` is able to sort an arbitrary segment of an array between `lo` and `hi` inclusive, leaving all the elements of the array outside the segment unchanged. The


```
template<class T>
class ASorter {

public:
    void sort(ASeq<T> &a, int lo, int hi);
    // Pre: 0 <= lo && hi < a.cap().
    // Post: sorted(a[lo..hi]).

    virtual ~ASorter() = default;
    // Virtual destructor necessary for subclassing.

protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) = 0;
    // Pre: lo < hi.
    // Post: lo < mid <= hi.

    virtual void join(ASeq<T> &a, int lo, int mid, int hi) = 0;
    // Pre: lo < mid <= hi.
    // Pre: sorted(a[lo..mid-1]) && sorted(a[mid..hi]).
    // Post: sorted(a[lo..hi]).
};
template<class T>
void ASorter<T>::sort(ASeq<T> &a, int lo, int hi) {
    if (lo < 0 || a.cap() <= hi) {
        cerr << "ASorter<T>::sort precondition failed." << endl;
        cerr << "lo == " << lo << " a.cap() == " << a.cap()
             << " hi == " << hi << endl;
        throw -1;
    }
    if (lo < hi) {
        int mid;
        split(a, lo, mid, hi);
        sort(a, lo, mid - 1);
        sort(a, mid, hi);
        join(a, lo, mid, hi);
    }
}
```

Figure 4.7 `ASorter.hpp`. Specification of the abstract sorter `ASorter`. Function `sort()` is implemented here and is not overridden by a subclass. Functions `split()` and `join()` are pure virtual and must be implemented by a specific subclass.

precondition guarantees that the values of `lo` and `hi` are reasonable. The case where `lo` and `hi` are equal corresponds to a segment of one element, which is already sorted. The case where `hi` is less than `lo` corresponds to an empty segment, which is also already sorted.



(a) The general array `a`.

(b) Array `a` with the smallest segments.

Figure 4.8 The `split()` primitive operation.

The primitive operation `split()` has precondition

$l < h$

and postcondition

$l < m \leq h$

It calls array `a` and integer `mid` by reference. It always sets `mid` and either modifies array `a` or not depending on the specific sorting algorithm. The precondition guarantees that `lo` and `hi` are not equal so that the segment to be split contains at least two elements. The postcondition guarantees that that `mid` is greater than `lo` which in turn guarantees that both the left segment `a[lo..mid-1]` and the right segment `a[mid..hi]` have length less than the original segment `a[lo..hi]`.

This latter guarantee is necessary to insure that the sort method terminates. Note that if `mid` were to equal `lo` then the left segment would be `a[lo..lo-1]`, *i.e.* empty, and the right segment would be `a[lo..hi]`, *i.e.* identical to the original segment. The algorithm would not make progress, and the recursive calls to `sort()` would be endless.

Figure 4.8(a) shows the two segments in the general case, and Figure 4.8(b) shows the case with the shortest possible segment having two elements before the split with $l = m-1 < m = h$.

The primitive operation `join()` has precondition

$l < m \leq h \wedge \text{sorted}(a[l..m-1]) \wedge \text{sorted}(a[m..h])$

and postcondition

$\text{sorted}(a[l..h])$

It calls array `a` by reference and integers `lo`, `mid`, and `hi` by value. It never modifies `mid` and either modifies array `a` or not depending on the specific sorting algorithm. The first conjunct of the precondition of join, $l < m \leq h$, corresponds to the postcondition of `split()`. Because `sort()` does not modify `lo`, `mid`, or `hi`, the first conjunct is guaranteed to hold when `join()` executes.

The code for `sort()` is a direct implementation of Figure 4.1, namely, first split the list, then sort the left sublist, then sort the right sublist, then join the two sorted lists. Following is a proof that `sort()` is correct provided that the preconditions and postconditions of `split()` and `join()` are correct.

Proof: The sort algorithm is recursive. Hence, the proof of its correctness is a proof by mathematical induction.

Base case: The base case occurs when the segment of the array to be sorted is empty or when it has one element. In that case, the segment is already sorted. The `if` statement in `sort()` prevents any processing when the segment is empty or when it has one element. Therefore, the base case is proved.

Induction case: Show that the sort algorithm will sort correctly for a large segment assuming that it does sort two smaller subsegments as the inductive hypothesis. But that follows by the pre- and postconditions of `split()`, `sort()`, and `join()`. First, the `if` statement of `sort()` guarantees the precondition of `split()`, $l < h$. Then, the postcondition of `split()`, $l < m \leq h$, guarantees that each of the two segments are smaller than the original segment. So, by the inductive hypothesis, the postcondition of the first recursive call, $sorted(a[l..m-1])$, and the postcondition of the second call, $sorted(a[m..h])$, are guaranteed. These two postconditions, together with the postcondition from `split()`, are the preconditions of `join()`. The postcondition of `join()` is identical to the postcondition of `sort()`. Therefore, the induction case is proved. ∎

Any comparative sort algorithm need only implement a `split()` and a corresponding `join()` that work together to satisfy the pre- and postconditions of `split()` and `join()` as specified in `ASorter`. The sort template method executes the same for all comparative sorts. Therefore, the proof of correctness of the sort template method applies to each comparative sort that implements its `split()` and `join()`. In the same way that the code for `sort()` is reused for all the sort algorithms, the proof of its correctness applies to all the sort algorithms. Not only does the Template Method pattern provide code reuse, it also provides proof-of-correctness reuse.

Implementation of a sort application

The dp4ds distribution has a project named `SortInt` that presents the user with a choice of sort algorithms to use to sort an array of integers. To use a specific sorter requires the instantiation of a particular sorter.

For example, the C++ statement to declare an abstract sorter is

```
shared_ptr<ASorter<int>> sorter;
```

The above statement declares `sorter` to be a pointer to an abstract sorter. The abstract sorter class is a template class, which requires the programmer to supply the type of the element to be sorted, in this case `int`. Then, the statement to instantiate a sorter using the quick sort algorithm is

```
sorter = make_shared<QuickSorter<int>>();
```

An abstract sorter is designed to take a safe array of type `ASeq<T>` defined in the dp4ds distribution library. To use the above sorter would require the declaration of a safe array as follows.

```
ArrayT<int> array(promptIntGE("Enter array capacity", 1));
```

Recall that `promptIntGE()` is a library routine that prompts the user for an input value, which in this case must be greater than or equal to one. After reading the values into `array`, they are sorted with the call


```
template<class T>
class MergeSorter : public ASorter<T> {
private:
    ArrayT<T> _tempA;

public:
    MergeSorter(int cap);

protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T> &a, int lo, int mid, int hi) override;
};

template<class T>
MergeSorter<T>::MergeSorter(int cap): _tempA(cap) {}

template<class T>
void MergeSorter<T>::split(ASeq<T> &, int lo, int &mid, int hi) {
// Post: mid ==(lo + hi + 1) / 2
    mid = (lo + hi + 1) / 2;
}
template<class T>
void MergeSorter<T>::join(ASeq<T> &a, int lo, int mid, int hi) {
    cerr << "MergeSorter<T>::join: Exercise for the student." << endl;
    throw -1;
}
```

Figure 4.9 `MergeSorter.hpp`. Implementation of `MergeSorter`.

```
sorter->sort(array, 0, length - 1);
```

where `length` is the number of elements in the array.

Implementation of MergeSorter

Figure 4.9 shows the implementation of `MergeSorter`, and Figure 4.10 shows a trace of its top-level calls with a ten-element array.

Merge sort is easy to split and hard to join. The primitive operation `split()` is a single line of code:

```
mid = (lo + hi + 1)/2;
```

where `/` executes as integer division, that is, truncation. Figure 4.10(b) shows the computation of `mid` as the vertical bar to the left of the cell with index 5, the value of `mid`. Following is a proof that `split()` is correct.

Proof: To prove the `split()` postcondition $l < m \leq h$ prove each conjunct, $l < m$ and $m \leq h$. Assume the `split()` precondition, $l < h$, from the previous proof of `sort()`.

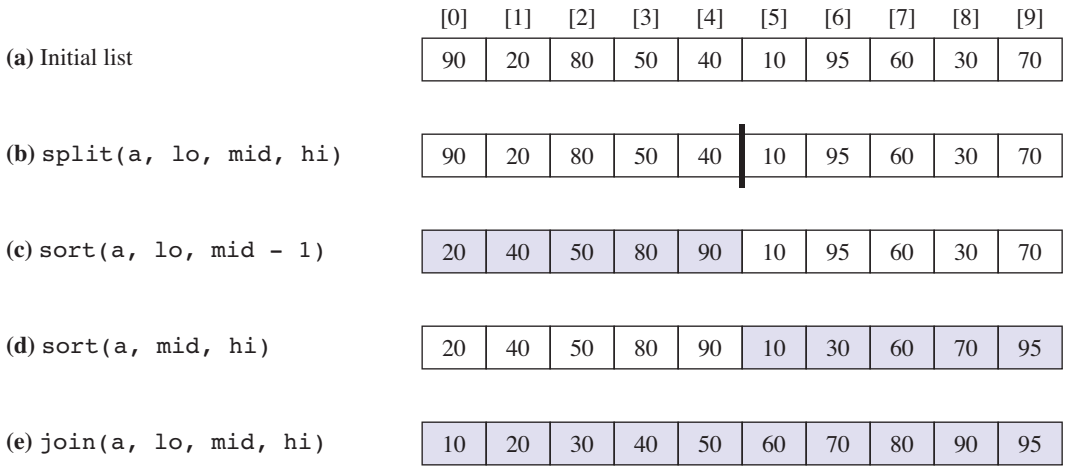


Figure 4.10 A trace of the top-level calls that `sort()` makes with `MergeSorter`. The figure shows a call to `sort()` when `lo` is 0, `hi` is 9, and `mid` is computed as 5.

First conjunct: To prove the first conjunct, $l < m$, compute the smallest possible value of `mid` and show that it is greater than `lo`.

$$
\begin{array}{ll}
 & m \\
= & \langle \text{Assignment statement in split} \rangle \\
 & (l+h+1) \operatorname{div} 2 \\
= & \langle l < h \Rightarrow m \text{ is closest to } l \text{ when } h = l+1 \rangle \\
 & (l+(l+1)+1) \operatorname{div} 2 \\
= & \langle \text{Math} \rangle \\
 & l+1
\end{array}
$$

which is greater than `lo`.

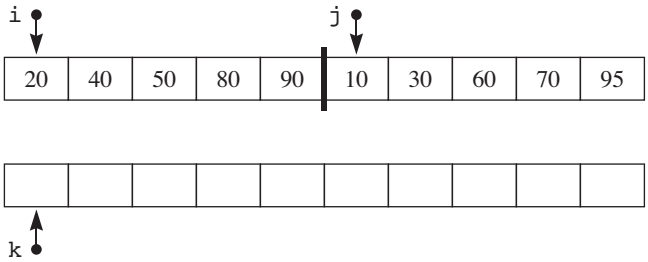
Second conjunct: To prove the second conjunct, $m \leq h$, compute the largest possible value of `mid` and show that it is at most `hi`.

$$
\begin{array}{ll}
 & m \\
= & \langle \text{Assignment statement in split} \rangle \\
 & (l+h+1) \operatorname{div} 2 \\
= & \langle l < h \Rightarrow m \text{ is closest to } l \text{ when } l = h-1 \rangle \\
 & ((h-1)+h+1) \operatorname{div} 2 \\
= & \langle \text{Math} \rangle \\
 & h
\end{array}
$$

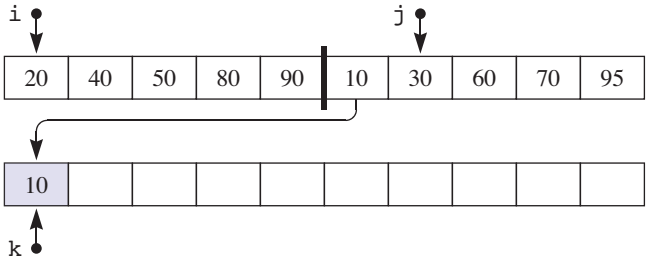
which is at most `hi`. Both proofs depend on the fact that `mid` is closest to both `lo` and `hi` when the segment is as small as possible, which happens when `hi` is `1+lo`. ∎

As you see in Figure 4.9, the implementation of `join()` for `MergeSorter` is left as an exercise for the student. Figure 4.11 shows an outline of how to do the merge.

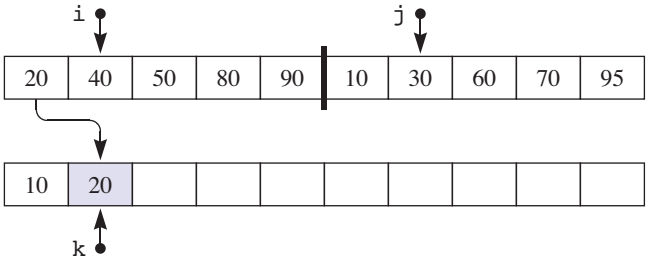
(a) Initialize `i` to the first element of the left segment, `j` to the first element of the right segment, and `k` to the first element of the segment in the temporary array.



(b) Compare 20 and 10. Because 10 is less than 20, copy `a[j]` to `_tempA[k]` and increment `j`.

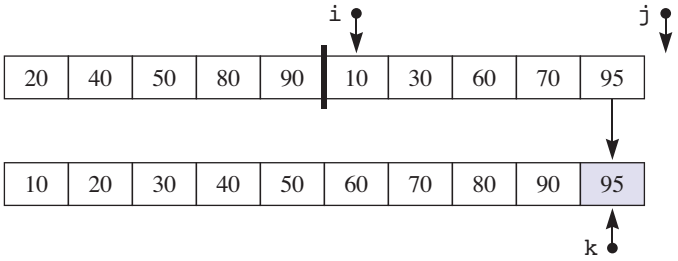


(c) Compare 20 and 30. Because 20 is less than 30, copy `a[i]` to `_tempA[k]` and increment `i`.



.
.
.

(d) Because `i` has increased past the last element of the left segment, do not compare two elements. Copy `a[j]` to `_tempA[k]` and increment `j`.



(e) Copy the segement from `_tempA` back into array `a`.

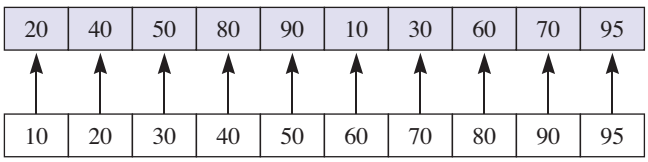


Figure 4.11 A low-level trace of the `join()` method of Figure 4.10(e) of `MergeSorter`. Seven steps are not shown between parts (c) and (d) of this figure.

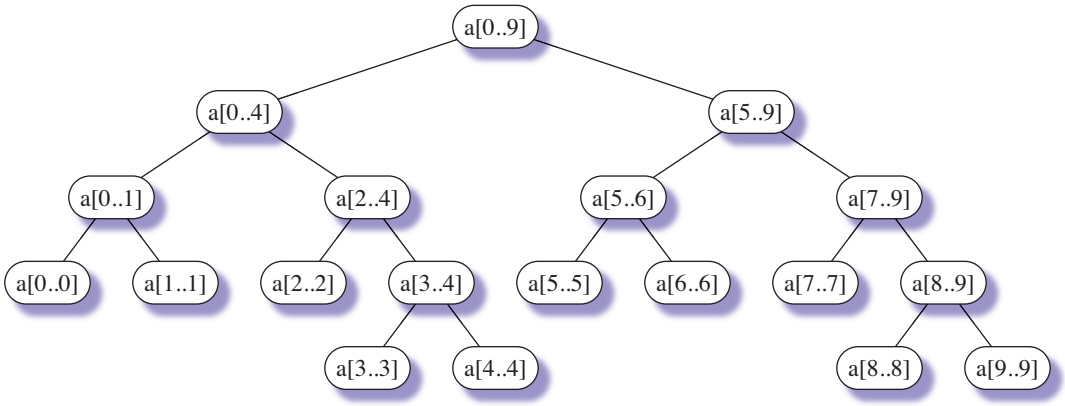


Figure 4.12 The call tree for `MergeSorter` in Figure 4.10. a[0..4] represents the call to `sort(a, lo, mid - 1)` in Figure 4.10(b) when `lo` is 0 and `mid` is 5. a[5..9] represents the call to `sort(a, mid, hi)` in Figure 4.10(c) when `mid` is 5 and `hi` is 9.

It consists of a single `for` loop to merge the two halves of the list segment into the temporary array `_tempA` followed by another `for` loop to copy the merged segment back into the original array `a`.

Figure 4.12 is the call tree for the values in Figure 4.10(a). For a left child, the values in brackets are the values of parameters `lo` and `mid-1`. For a right child the values in brackets are the values of parameters `mid` and `hi`. The leaf nodes have the two parameters equal, indicating that a subarray of one element needs to be sorted. The algorithm does no processing in those cases, but just returns to the calling procedure.

To determine the performance of the merge sort algorithm, consider the following code of `sort()`.

```
sort(a, lo, hi)
    if (lo < hi)
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

For merge sort, the split is $\Theta(1)$. Each of the two recursive sorts are for $n/2$ elements and hence take time $2T(n/2)$. The join requires a loop to merge the n elements, and hence is $\Theta(n)$. The recurrence for merge sort is, therefore,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

The solution to this recurrence is $T(n) = \Theta(n \lg n)$, which is a tight bound for the performance of merge sort.

Implementation of QuickSorter

Figure 4.13 is a trace of the top-level calls that `sort` makes with `QuickSorter`. In contrast to merge sort, quick sort is hard to split and easy to join. Figure 4.13(b) shows the effect of the split operation, which illustrates a famous dilemma with the design of the quick sort algorithm. Figure 4.10(b) shows the corresponding split for the merge sort algorithm, which computes `mid` to be 5. A value of 5 for `mid` splits the list exactly in half. On the other hand, Figure 4.13(b) shows the value of `mid` to be 6 for quick sort. Why the discrepancy? It would be better for the performance of quick sort to have the list split exactly in half.

The reason for the discrepancy becomes apparent when you analyze the requirement for the split operation. The split operation rearranges the values in array a to make every value in $a[l..m-1]$ less than or equal to every value in $a[m..h]$. To decide whether to place a value in a on the left or the right, the algorithm must compare that value with some other key value.

The *median* of a list of numbers is that number such that there are just as many values below the median as there are above the median. It necessarily follows that in order to spilt the list exactly in half, each value in array $a[l..h]$ must be compared to the median of the list. If the value is less than the median it is moved to the left half, and if it is greater than the median it is moved to the right half.

Here is the crux of the dilemma. The only way to compute the median of a list of numbers is to sort them and then pick the middle value as the median. But, sorting a list of numbers is the task of the quick sort algorithm in the first place! You cannot have an algorithm whose subproblem requires the solution of the main problem.

The solution to this dilemma is to estimate the median by sampling just a few values in $a[l..h]$. Using such an estimate in the algorithm for `split()` implies that the array a will not be split exactly in half each time, and therefore that the performance may suffer somewhat. Here are three common strategies for estimating the median.

- Estimate the median of the elements in $a[l..h]$ as $a[h]$.
- Estimate the median of the elements in $a[l..h]$ as $a[r]$, where r is an integer index chosen at random from the range $l..h$.
- Estimate the median of the elements in $a[l..h]$ as the median of the three values $a[r]$, $a[s]$, and $a[t]$ where r, s, and t are integer indices chosen at random from the range $l..h$.

Figure 4.14 is a low-level trace of the split operation. Part (a) of the figure is the initial list and corresponds to Figure 4.13(a). Part (k) of the figure is the list after the split and corresponds to Figure 4.13(b). The trace assumes either the second or third option above for estimating the median. It is impossible to know which value the algorithm will choose for the estimate, because the algorithm is based on a random sample. If you run a program that implements this algorithm several times with the same input, its performance will vary slightly from run to run. Furthermore, if you trace the runs you will find that the splits do not occur in the same places from run to run. However, the final list will always be sorted.

Figure 4.14(b) shows the first step of the algorithm, which assumes that 60 is chosen as the key value. Such a choice could happen, for example, if the values 80, 40, and 60 were chosen at random, because 60 is the median of these three values. This step of the

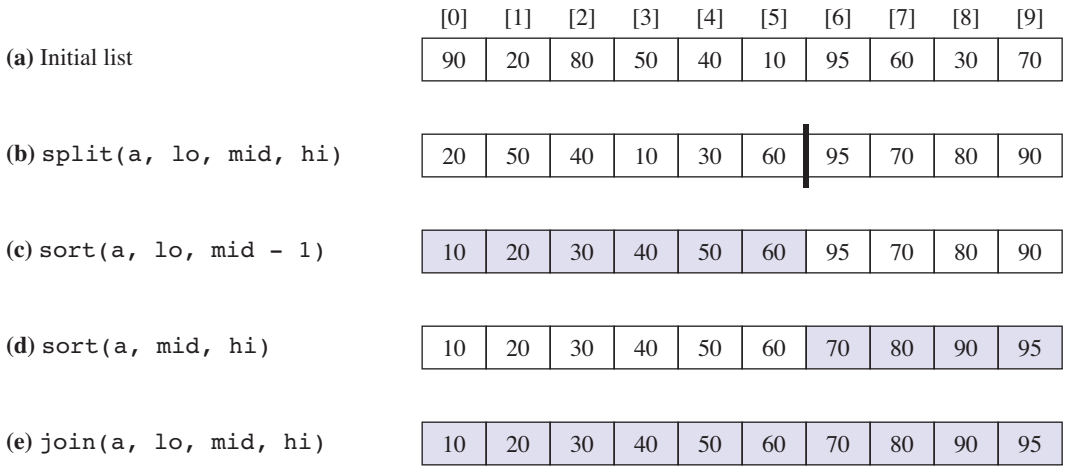


Figure 4.13 A trace of the top-level calls that `sort()` makes with `QuickSorter`. The figure shows a call to `sort()` when `lo` is 0, `hi` is 9, and `mid` is computed as 6.

algorithm swaps 60 with `a[hi]`. There are many different versions of the quick sort algorithm, and most of them use `a[hi]` to store the key value.

Figure 4.14(c) shows integer variables `mid` and `j` initialized to `lo`.

Parts (d) – (k) of Figure 4.14 is a trace of the loop that performs the split. Every execution of the loop causes `j` to increase by one. Every time through the loop, the algorithm compares the value of `a[j]` with the key value. If `a[j]` is less than or equal to `key`, the algorithm swaps `a[j]` and `a[mid]` and then increments `mid`. Otherwise, no swap occurs, and the value of `mid` is unchanged.

Figure 4.14(k) shows the last step of the loop, in which 60, the key value, is swapped with 90, which is `a[mid]`. Then, `mid` is incremented. The algorithm always swaps the key value with `a[mid]` in the last step of the loop, because that test is always between `a[j]` and itself. Any number is always less than or equal to itself.

The algorithm for quick sort requires that three values be selected at random from $a[l..h]$. The problem is that all computations in a computer are deterministic. It is impossible to compute a truly random value. So, instead of computing random numbers, computers instead compute *pseudo-random* numbers, which are not really random but appear so to the casual observer. Most random number generators compute the next random value from the previously computed one. The starting value for the pseudo-random sequence is called the *seed*.

Figure 4.15 shows two additional attributes for `QuickSorter` that are not in the abstract sorter class. These two private attributes, `rdev` and and `engine`, are for generating the random integers in the estimate of the median. Their types are defined in library `random`, which must be included in the header file, and are part of the `std` name space.

The object `rdev` provides the seed for the random number generator and is tied to some physical device inside the processor. An example of a random physical device is the system clock, which keeps track of the date and time. Each time you run the

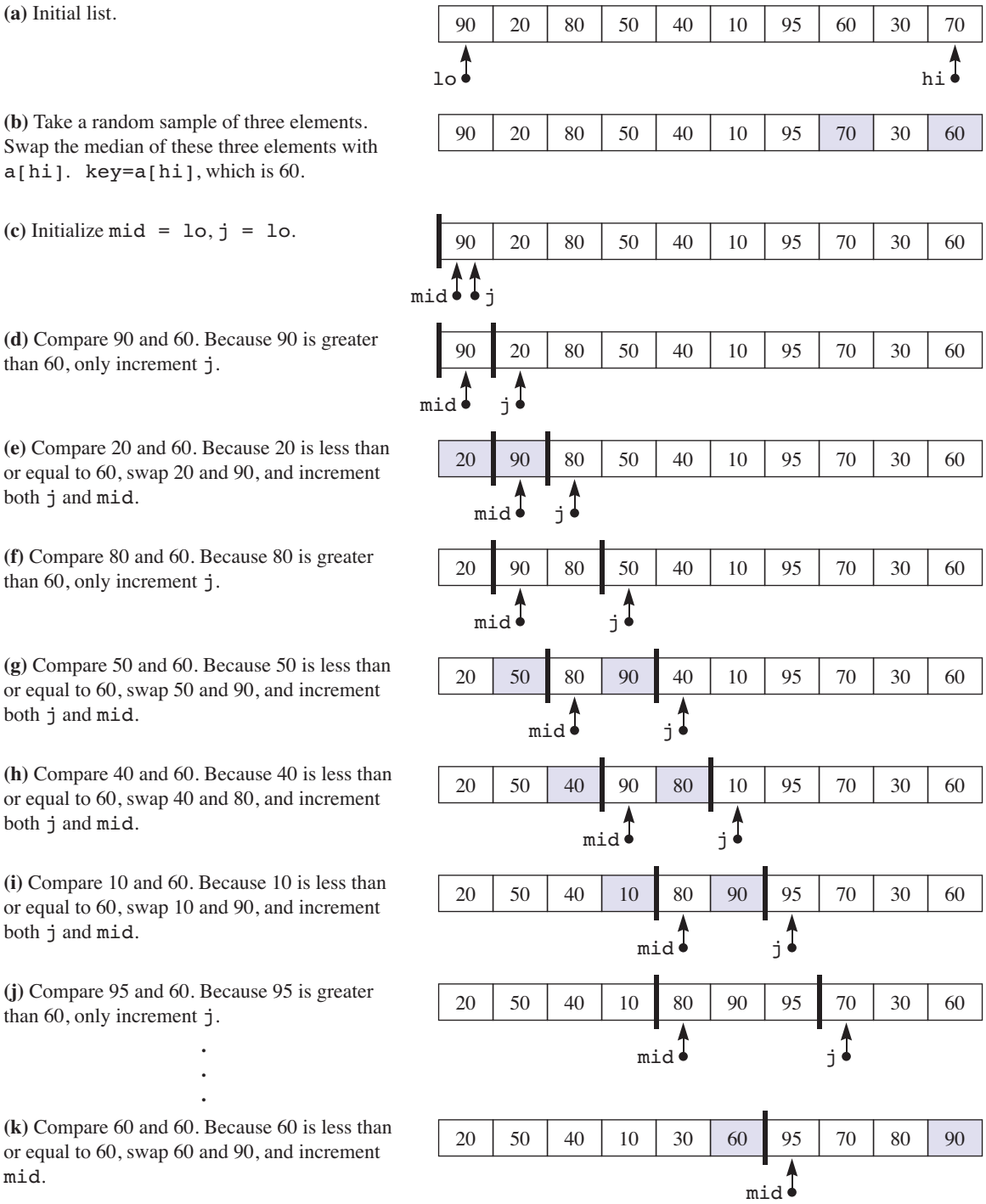


Figure 4.14 A low-level trace of the `split()` method of Figure 4.13(b) of `QuickSorter`. Two steps are not shown between parts (j) and (k) of this figure.


```
#include <random>
#include "ASorter.hpp"

template<class T>
class QuickSorter : public ASorter<T> {
protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T> &a, int lo, int mid, int hi) override;

private:
    random_device rdev{};
    default_random_engine engine{rdev()};
};
```

Figure 4.15 `QuickSorter.hpp`. Implementation of the quick sort algorithm. The program listing continues in the next figure.

program it will execute at a different time, and the seed value returned by `rdev` will be different. Some processors have a sensor that measures a signal from some random physical event. Whatever device `rdev` is tied to, it is impossible to predict what seed value it will provide.

Once you set the seed to an initial value, subsequent random numbers are determined precisely by the generating algorithm and could theoretically be predicted. Because the algorithm is usually complex, it is difficult in practice to make such predictions even if you know the initial seed value. If you run `QuickSorter` twice, the second run will have a different seed, the random number generator will produce a different sequence of random numbers, and the splits will not be identical to those of the first run.

If you are trying to debug a program that initializes the seed using this technique and you want the sequence of random numbers to be the same from one run to the other, you can change the initialization to

```
default_random_engine engine{1};
```

Because this statement initializes the seed to 1, the same sequence of pseudo-random numbers will be generated with every run.

Figure 4.16 shows the `split()` method for `QuickSorter`. The line

```
uniform_int_distribution<int> distr(lo, hi);
```

declares local object `distr`, which calls its constructor with parameters `lo` and `hi`. Distributions operate in conjunction with engines. An engine provides random values to a distribution, and the distribution filters the values to provide the required distribution of random values. As the name of its class indicates, `distr` provides a uniform distribution of integer values between the values of `lo` and `hi` inclusively. The line

```
int mdn1 = distr(engine);
```

calls the overloaded function `distr()` which requires an engine for its parameter and returns a random integer with the required distribution.


```
template<class T>
void QuickSorter<T>::split(ASeq<T> &a, int lo, int &mid, int hi) {
// Post: a[lo..mid-1] <= a[mid..hi]
    T temp;
    if (hi - lo > 4) {
        int mdn; // Index of the estimate of the median value.
        uniform_int_distribution<int> distr(lo, hi);
        // Find the estimate of the median.
        int mdn1 = distr(engine);
        int mdn2 = distr(engine);
        int mdn3 = distr(engine);
        if ((a[mdn2] <= a[mdn1] && a[mdn1] <= a[mdn3])
                || (a[mdn3] <= a[mdn1] && a[mdn1] <= a[mdn2])) {
            mdn = mdn1; // a[mdn1] is the median
        } else if ((a[mdn1] <= a[mdn2] && a[mdn2] <= a[mdn3])
                || (a[mdn3] <= a[mdn2] && a[mdn2] <= a[mdn1])) {
            mdn = mdn2; // a[mdn2] is the median
        } else {
            mdn = mdn3; // a[mdn3] is the median
        }
        // Swap the estimate of the median with a[hi].
        temp = a[mdn];
        a[mdn] = a[hi];
        a[hi] = temp;
    }
    // Now do the split.
    T key = a[hi];
    mid = lo;
    for (int j = lo; j <= hi; j++) {
        if (a[j] <= key) {
            temp = a[mid];
            a[mid] = a[j];
            a[j] = temp;
            mid++;
        }
    }
    mid = hi < mid ? hi : mid; // If a[hi] contains the maximum element.
}

template<class T>
void QuickSorter<T>::join(ASeq<T>&, int lo, int mid, int hi) {
}
```

Figure 4.16 `QuickSorter.hpp` (continued). Implementation of the quick sort algorithm. This concludes the program listing.

The separation of initializers, generators, and distributions from each other in the C++ `random` library is a major advance over earlier C++ libraries and libraries for other programming languages. Good pseudo-random number generators are difficult to design and many other libraries provide notoriously poor services. The code in `QuickSorter` uses default methods for all three parts of the process and is good for casual use. For the specialist, the `random` library provides the ability to specify in detail the initializer, the generator, and the distribution. Generators include algorithms with such esoteric names as the `mt19937` Merssene twister and the `ranlux24` discard block engine. Distributions in the library include uniform, Bernoulli, Poisson, and normal.

The code in Figure 4.16 that computes the estimate of the median is within the `if` statement

```
if (hi - lo > 4)
```

Just to compute the estimate of the median requires several array element comparisons. Experimental data show that *always* using a random sample of three to estimate the median gives better performance than *never* using a random sample of three, that is, by taking $a[h]$ as the estimate. However, the algorithm gives even better performance by *sometimes* using a random sample of three. The above `if` statement does not use the random sample of three if there are five or fewer elements in the range $h..l$. When the range is small, the number of comparisons to compute the estimate with the random sample predominates over the number of comparisons to do the sort itself. In that case, it does not pay to estimate the median with a random sample of three elements.

There is one final statement in `split()` shown in Figure 4.16 that does not appear in Figure 4.14.

```
mid = hi < mid ? hi : mid;
```

Consider what would happen if by chance, the estimate of the median were the greatest element of the array. It would be placed at `a[hi]`, and would be the key value. The test

```
if (a[j] <= key)
```

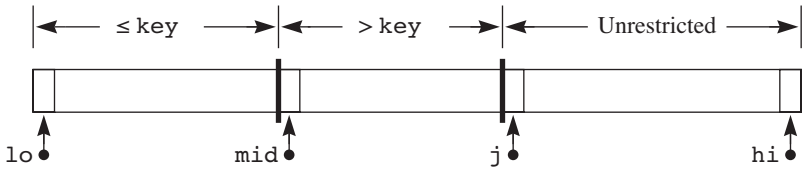
would succeed every time, `mid` would increment every time, and the final value of `mid` would be one plus `hi`. But that would violate the postcondition for `split()`, which requires `mid` to be less than or equal to `hi`. The final adjustment to `mid` in that case sets it to `hi` to satisfy the postcondition.

Figure 4.16 also shows the `join()` method for `QuickSorter`. Even though `join()` does nothing, the Template Method pattern requires it to exist because `sort()` calls both `split()` and `join()` for all sort algorithms.

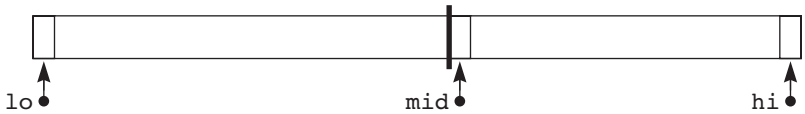
The proof that `split()` is correct depends first on identifying the relevant loop invariant. Figure 4.14 shows the progress of `mid` and `j` as the loop executes. In each part of the figure during execution of the loop, is a vertical bar to the left of `mid` and another to the left of `j`. You can see in the figures that all the values to the left of `mid` are less than or equal to all the values at or to the right of `mid` and to the left of `j`. Any values at or beyond `j` have not been processed.

Figure 4.17(a) is a general depiction of the three regions of array `a` subdivided by `mid` and `j`. Using single-letter abbreviations for the variables, the loop invariant is

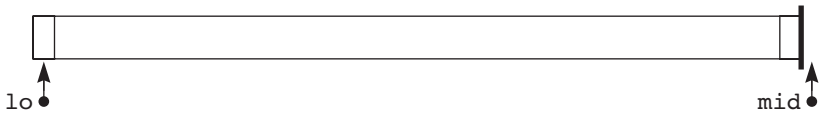
$$(\forall i \mid l \leq i < m : a[i] \leq k) \wedge (\forall i \mid m \leq i < j : a[i] > k)$$



(a) The three regions during execution of the loop.



(b) The two regions after normal termination of the loop.



(c) After termination of the loop if `a[hi]` contains the maximum element.

(d) After adjustment of `mid` if `a[hi]` contains the maximum element.

Figure 4.17 Regions of the array during the `split()` primitive operation of quick sort.

which states that the elements in $a[l..m-1]$ are all less than or equal to k, and the elements in $a[m..j-1]$ are all greater than k. The elements in $a[j..h]$ have not been processed, and hence have no restrictions on their values. Figure 4.17(b) shows the regions after normal termination of the loop. Because the final value of j is $h+1$, the unrestricted region is empty. Part (c) shows the final value of m when $a[h]$ contains the maximum element. For the recursion to terminate, each of the two subproblems must be smaller than the original problem. Part (d) shows the adjustment of m, which guarantees the general postcondition $l < m \leq h$.

To prove that `split()` is correct is a four-step process.

- Prove that the invariant is true at the beginning of the loop.
- Prove that the invariant is maintained with each execution of the loop.
- Prove that the loop terminates.
- Prove that the postcondition holds at the end of the loop.

Figure 4.16 shows a second postcondition for `split()`, namely $a[l..m-1] \leq a[m..h]$, which means that every element in $a[l..m-1]$ is less than or equal to every element in $a[m..h]$. This postcondition is in addition to the general postcondition for `split()` in Figure 4.7, which is $l < m \leq h$.

Here are the four parts of the proof.

The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

$$
\begin{array}{ll}
 & (\forall i \mid l \le i < m : a[i] \le k) \wedge (\forall i \mid m \le i < j : a[i] > k) \\
= & \langle \text{Assignment statements } \texttt{mid = lo} \text{ and } \texttt{j = lo} \text{ in } \texttt{sort()} \rangle \\
 & (\forall i \mid l \le i < l : a[i] \le k) \wedge (\forall i \mid l \le i < l : a[i] > k) \\
= & \langle \text{Math} \rangle \\
 & (\forall i \mid false : a[i] \le k) \wedge (\forall i \mid false : a[i] > k) \\
= & \langle \text{Empty range rule} \rangle \\
 & true \wedge true \\
= & \langle \text{Idempotency of } \wedge \rangle \\
 & true \quad \blacksquare
\end{array}
$$

The invariant is maintained with each execution of the loop.

Proof: There are two cases for the nested `if` statement inside the loop.

Case 1: $a[j] > k$. The only code that executes is `j++`. Only the second conjunct of the invariant is affected, $(\forall i \mid m \le i < j : a[i] > k)$. Increasing j by one excludes the value of $a[j]$ from the unrestricted region and includes it in the $> k$ region. The conjunct of the invariant is maintained because of the `if` guard, $a[j] > k$.

Case 2: $a[j] \le k$. First, $a[m]$ is swapped with $a[j]$, then `m++` and `j++` execute. Only the first conjunct of the invariant is affected, $(\forall i \mid l \le i < m : a[i] \le k)$. Swapping $a[m]$ with $a[j]$ temporarily moves $a[m]$ to the left-most position of the unrestricted region and $a[j]$ to the left-most position of the $> k$ region. Increasing m then includes the value that was at location j before the swap, into the $\le k$ region. The first conjunct of the invariant is maintained because of the `if` guard $a[j] \le k$. Increasing j then includes the value that was at location m before the swap, into the $> k$ region. But it was already in the $> k$ region before the loop executed, so the second conjunct of the invariant is not affected. $\blacksquare$

The loop terminates.

Proof: The loop is controlled by the `for` statement

```
for (int j = lo; j <= hi; j++)
```

So, every time through the loop j increases by one. Furthermore, no statement in the algorithm changes h. Therefore, eventually j will be greater than h, and the loop will terminate. $\blacksquare$

The postcondition holds at the end of the loop.

Proof: Starting with the invariant,

$$
\begin{array}{ll}
 & (\forall i \mid l \le i < m : a[i] \le k) \wedge (\forall i \mid m \le i < j : a[i] > k) \\
= & \langle \text{The final value of } j \text{ is } h+1 \rangle \\
 & (\forall i \mid l \le i < m : a[i] \le k) \wedge (\forall i \mid m \le i < h+1 : a[i] > k) \\
= & \langle \text{Math} \rangle \\
 & (\forall i \mid l \le i < m : a[i] \le k) \wedge (\forall i \mid m \le i \le h : a[i] > k) \\
\Rightarrow & \langle \text{Transitivity of the } < \text{ operator and logic} \rangle \\
 & a[l..m-1] \le a[m..h]
\end{array}
$$

which is the second postcondition of `sort()`.

The first postcondition is $l < m \leq h$, which is guaranteed by the final adjustment of m shown in Figure 4.17(d). This adjustment does not invalidate the second postcondition, because it moves $a[h]$ from the $\leq k$ region to the $> k$ region, and only happens when $a[h] = k$. The $> k$ region thereby becomes the $\geq k$ region, but the second postcondition still holds as it does not require a strict inequality. ∎

To determine the performance of the quick sort algorithm, consider the code of `sort()`. For quick sort, the split

```
split(a, lo, mid, hi)
```

requires a loop to split the n elements and hence is $\Theta(n)$. Assuming the best case, each of the two recursive sorts

```
sort(a, lo, mid - 1)
sort(a, mid, hi)
```

are for $n/2$ elements and hence take time $2T(n/2)$. The join

```
join(a, lo, mid, hi)
```

does no work and hence is $\Theta(1)$. The recurrence for best case quick sort is, therefore,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

which is identical to the recurrence for merge sort. The only difference is that for merge sort the $\Theta(1)$ term comes from the split and the $\Theta(n)$ term comes from the join, while for quick sort the $\Theta(1)$ term comes from the join and the $\Theta(n)$ term comes from the split. The solution is the same, namely $T(n) = \Theta(n \lg n)$ for the best case.

For the worst case, the estimate of the median is always the maximum element, which is placed in $a[h]$ before the split. The split is still $\Theta(n)$, because the loop must compare all n elements with the key value. `mid` is always set to `hi` as Figure 4.17(d) shows. The first sort

```
sort(a, lo, mid - 1)
```

is a sort of $a[l..m-1]$ which is $a[l..h-1]$ when $m = h$. Therefore, it takes time $T(n-1)$. The second sort

```
sort(a, mid, hi)
```

is a sort of one element, because $a[m..h]$ is $a[h..h]$ when $m = h$. Therefore, it takes time $\Theta(1)$. The join takes time $\Theta(1)$ as in the best case. The recurrence for worst case quick sort is, therefore,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

The solution to this recurrence is $T(n) = \Theta(n^2)$.

So in the best case, quick sort has $T(n) = \Theta(n \lg n)$ which is asymptotically better than its worst case $T(n) = \Theta(n^2)$. You might think that in the average case, quick sort would have performance somewhere between these two extremes. The remarkable thing


```
template<class T>
class SelectSorter : public ASorter<T> {
protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T> &a, int lo, int mid, int hi) override;
};

template<class T>
void SelectSorter<T>::split(ASeq<T> &a, int lo, int &mid, int hi) {
// Post: a[hi] == max(a[lo..hi]).
// Post: mid == hi.
    int indexOfMax = lo;
    for (int i = lo + 1; i <= hi; i++) {
        if (a[indexOfMax] < a[i]) {
            indexOfMax = i;
        }
    }
    T temp = a[hi];
    a[hi] = a[indexOfMax];
    a[indexOfMax] = temp;
    mid = hi;
}

template<class T>
void SelectSorter<T>::join(ASeq<T>&, int lo, int mid, int hi) {
}
```

Figure 4.18 `SelectSorter.hpp`. Implementation of the selection sort algorithm.

about the quick sort algorithm is that both in practice and in theory its average execution time is $T(n) = \Theta(n \lg n)$. The analysis of its average performance is beyond the scope of this book. Quick sort therefore competes favorably with merge sort, especially when you consider that merge sort cannot be done in place, and requires an extra temporary array and copy back operation shown in Figure 4.11(e).

Implementation of SelectSorter and InsertSorter

Figure 4.3(b) shows that selection sort is quick sort when list $L2$ on the right has only one element. The split exchanges the largest element 8 with the last element 4. Figure 4.18 shows the implementation of `SelectSorter`. Because selection sort is a member of the quick sort family, it is hard to split and easy to join. The figure shows that, as with quick sort, `join()` does no work at all.

The purpose of `split()` is to exchange the largest element in $a[l..h]$ with $a[h]$. The `for` loop determines the index of the largest element, after which the largest is exchanged with the last element. Figure 3.1 (page 77) shows a trace of function `largest`


```
template<class T>
class InsertSorter : public ASorter<T> {
protected:
    virtual void split(ASeq<T>&, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T>&, int lo, int mid, int hi) override;
};

template<class T>
void InsertSorter<T>::split(ASeq<T> &, int, int &mid, int hi) {
// Post: mid == hi.
    mid = hi;
}

template<class T>
void InsertSorter<T>::join(ASeq<T> &a, int lo, int mid, int hi) {
// Pre: mid == hi && sorted(a[lo..hi - 1]).
// Post: sorted(a[lo..hi]).
    cerr << "InsertSorter<T>::join: Exercise for the student." << endl;
    throw -1;
}
```

Figure 4.19 `InsertSorter.hpp`. Implementation of the insertion sort algorithm.

`Last()`, which does the same thing but with an array starting at index 0 with `len` elements. Section 3.4 proves the correctness of `largestLast()` and can be modified slightly to prove the correctness of `split()`.

To determine the asymptotic bounds of `SelectSorter` consider the worst-case analysis of quick sort. The worst case happens when *by chance* the estimate of the median is the largest element and is swapped with the last element of the array segment. The same thing happens in selection sort *by design*. The split operation always finds the largest element, swaps it with the last element, and sets `mid` to `hi`. The recurrence is identical to that of worst-case quick sort, and the conclusion is the same. Selection sort has asymptotic tight bound $T(n) = \Theta(n^2)$.

Figure 4.3(a) shows that insertion sort is merge sort when list $L2$ on the right has only one element. The split isolates the last element 4 in list $L2$ on the right. Figure 4.19 shows the implementation of `InsertSorter`. Because insert sort is a member of the merge sort family, it is easy to split and hard to join. The figure shows that, as with merge sort, `split()` consists of a single assignment statement setting the value of `mid`. For merge sort the assignment is

```
mid = (lo + hi + 1) / 2;
```

setting `mid` to the to the midpoint between `lo` and `hi`, while for insert sort the assignment is

```
mid = hi;
```


isolating the last element as a single value in list *L2* on the right. Implementation of `join()` for `InsertSorter` is an exercise for the student. Figure 4.20 is a trace of the processing that must be done.

To determine the performance of the insert sort algorithm, consider the code of `sort()`. For insert sort, the split

```
split(a, lo, mid, hi)
```

is a single assignment and hence is $\Theta(1)$. The first recursive sort

```
sort(a, lo, mid - 1)
```

is a sort of $n-1$ elements and hence takes time $T(n-1)$. The second recursive sort

```
sort(a, mid, hi)
```

does no work and hence is $\Theta(1)$. The join

```
join(a, lo, mid, hi)
```

as shown in Figure 4.20 requires a loop that executes n times in the worst case and $n/2$ times on average. Hence, its asymptotic bound is $\Theta(n)$. The recurrence for insert sort is, therefore,

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1, \\ T(n-1) + \Theta(n) & \text{if } n > 1. \end{cases}$$

which is identical to the recurrence for selection sort. The only difference is that for insert sort the $\Theta(1)$ term comes from the split and the $\Theta(n)$ term comes from the join, while for selection sort the $\Theta(1)$ term comes from the join and the $\Theta(n)$ term comes from the split. The solution is the same, namely $T(n) = \Theta(n^2)$.

Implementation of HeapSorter

The heap sort algorithm is a member of the quick sort family because it is hard to split and easy to join. Figure 4.5 shows that the algorithm has a preprocessing step not required by the other comparison sort algorithms. The preprocessing step is labeled Build heap in the figure and is implemented by the constructor for `HeapSorter` shown in Figure 4.21. The documentation for the constructor has

```
// Post: maxHeap(a[lo..hi]).
```

which indicates that the array segment $a[l..h]$ is a max-heap. That is, for every node other than the root, the value of the node is at most the value of its parent. Therefore, the maximum value in the tree is at the root.

The preprocessing computation makes repeated calls to `siftDown()`, which is also called by `split()`. The constructor for `HeapSorter`, `split()`, and `join()` are all methods of `HeapSorter`, but `siftDown()` is a void function in its own library header file because it is needed by another project (the priority queue). Figure 4.22 shows the implementation of `siftDown()`, which takes array `a` and indices `lo`, `i`, and `hi` as parameters. Parameter i is the index of a node that is required by the precondition to be in the range $l \leq i \leq h$. Another precondition is that the array in the subrange $a[i+1..h]$

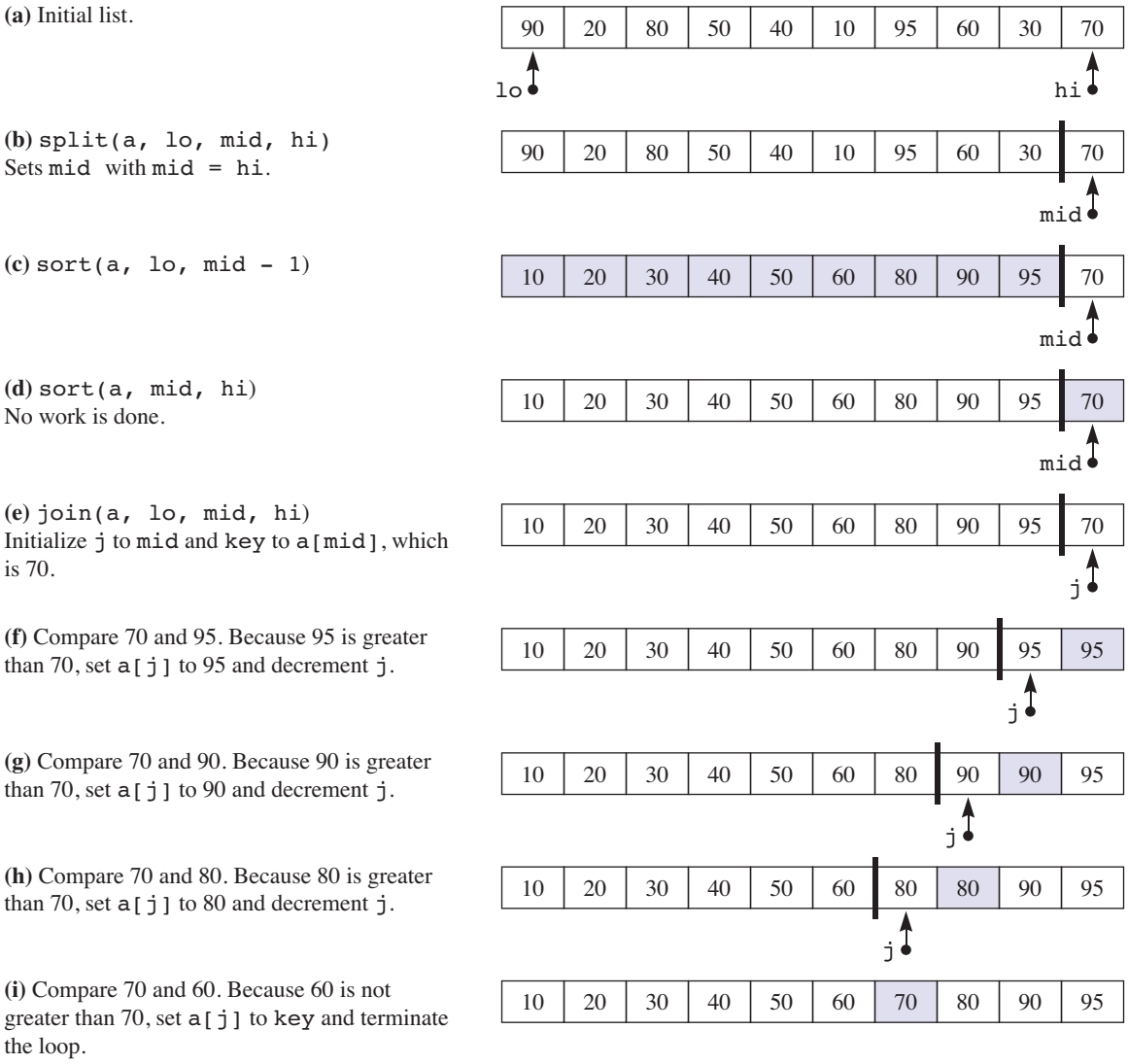


Figure 4.20 A trace of the calls that `sort()` makes with `InsertSorter`. Parts (e) – (i) are a low-level trace of `join()`.

is a max-heap. The postcondition is that the array in the slightly larger subrange $a[i..h]$ is a max-heap.

The operation of `siftDown()` requires the index of the left child of the node at index i. The heap library also contains a function `siftUp()` that requires the index of the parent of the node at index i. The processing to build the initial heap requires the largest index of a node with at least one child. These three relationships are as follows.


```
template<class T>
class HeapSorter : public ASorter<T> {
public:
    HeapSorter(ASeq<T> &a, int lo, int hi);
    // Constructor initializes a to a heap.
    // Post: maxHeap(a[lo..hi]).

protected:
    virtual void split(ASeq<T> &a, int lo, int &mid, int hi) override;
    virtual void join(ASeq<T> &a, int lo, int mid, int hi) override;
};

template<class T>
HeapSorter<T>::HeapSorter(ASeq<T> &a, int lo, int hi) {
// Post: maxHeap(a[lo..hi]).
    for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
        siftDown(a, lo, i, hi);
    }
}

template<class T>
void HeapSorter<T>::split(ASeq<T> &a, int lo, int &mid, int hi) {
// Pre: maxHeap(a[lo..hi]).
// Post: maxHeap(a[lo..hi - 1]).
// Post: a[hi] == old a[lo] && mid == hi.
    T temp = a[hi];
    a[hi] = a[lo];
    a[lo] = temp;
    siftDown(a, lo, lo, hi - 1);
    mid = hi;
}

template<class T>
void HeapSorter<T>::join(ASeq<T>&, int lo, int mid, int hi) {
}
```

Figure 4.21 `HeapSorter.hpp`. Implementation of the heap sort algorithm.

- The parent of the node at index i is at index $\lfloor (i+l-1)/2 \rfloor$.
- The left child of the node at index i is at index $2 \cdot i - l + 1$.
- The largest index of a node with at least one child is $\lfloor (l+h-1)/2 \rfloor$.

The notation $\lfloor \ \rfloor$ is the mathematical symbol for the *floor* operation, which corresponds to truncation of the fractional part of a numeric value. C++ accomplishes this operation automatically when it does integer division with the `/` operator. The right child of the node at index i is always one greater than the index of the left child. Figure 4.23 shows an


```
// ========= siftUp =========
template<class T>
void siftUp(ASeq<T> &a, int lo, int i) {
// Pre: maxHeap(a[lo..i - 1]).
// Post: maxHeap(a[lo..i]).
    T temp = a[i];
    int parent = (i + lo - 1) / 2;
    while (lo < i && a[parent] < temp) {
        cerr << "siftUp: Exercise for the student." << endl;
        throw -1;
    }
    a[i] = temp;
}

// ========= siftDown =========
template<class T>
void siftDown(ASeq<T> &a, int lo, int i, int hi) {
// Pre: maxHeap(a[i + 1..hi]).
// Pre: lo <= i <= hi.
// Post: maxHeap(a[i..hi]).
    int child = 2 * i - lo + 1; // Index of left child.
    if (child <= hi) {
        if (child < hi && a[child] < a[child + 1]) {
            child++;
        } // child is the index of the larger of the two children.
        if (a[i] < a[child]) {
            T temp = a[i];
            a[i] = a[child];
            a[child] = temp;
            siftDown(a, lo, child, hi);
        }
    }
}
```

Figure 4.22 `Heapifier.hpp`. Implementation of `siftUp()` and `siftDown()`.

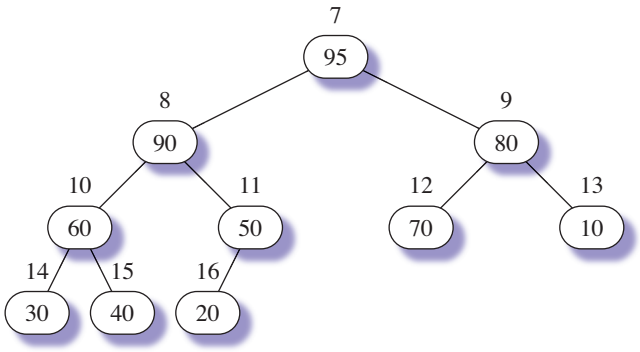
example with $l = 7$ and $h = 16$. The node at index 10 has parent at $\lfloor (10+7-1)/2 \rfloor = 8$ and left child at $2 \cdot 10 - 7 + 1 = 14$. The largest index of a node with at least one child is $\lfloor (7+16-1)/2 \rfloor = 11$.

Using the formula for the index of the parent node, the formal definition of the max-heap order is the predicate

$$\text{maxHeap}(a[l..h]) \equiv (\forall i \mid l+1 \le i \le h : a[\lfloor (i+l-1)/2 \rfloor \ge a[i]).$$

That is, for all nodes except the root, which is at $a[l]$, the value of a node is at most the value of its parent.

Figures 4.24 and 4.25 are a trace of `siftDown()`. Part (a) shows the initial tree



[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]
95	90	80	60	50	70	10	30	40	20

Figure 4.23 The relation between the index of an element in an array segment and its position in a max-heap.

with the index i at the left subchild of the root. The precondition of `siftDown()` is that every node with index greater than i satisfies the max-heap property. The figure shows that initially the node at i does not satisfy the max-heap property because 30 is not greater than or equal to either 90 or 80. On the other hand, both 90 and 80 are the roots of max-heaps. The purpose of `siftDown()` is to adjust the tree rooted at i to make it a max-heap assuming that its left and right subtrees are already max-heaps.

The algorithm computes the index of the left child of the node at i. If that index is greater than h, the node at i is a leaf, it automatically satisfies the max-heap property, and the algorithm terminates. Otherwise the algorithm checks if there is a right child and if so compares its value with the value of its left child. The test

```
if (child < hi && a[child] < a[child + 1])
```

depends on short-circuit evaluation of the `if` guard. The first conjunct checks for the existence of the right child and the second conjunct compares the values only if the right child exists.

The algorithm determines the maximum of the three values 30, 90, and 80. Because 90 is the largest, it swaps 30 with 90 and makes a recursive call to the node that now contains 30. Part (b) shows the result of the swap and recursive call.

The left child of 85 is now closer to being a max-heap, because the only node to violate the max-heap property is now the left child of 90. There can be no violation of the max-heap property with the right child of 90. The right child of 90 cannot be greater than 90, because 90 was chosen as the maximum for the swap.

Parts (c) and (d) show the same process with 30 sifting down to eventually become a leaf at the right child of 40. Because a leaf automatically satisfies the max-heap property, the left child of 85 is now a max-heap.

It does not necessarily happen that the node at i will sift down all the way down to a leaf. If the initial tree in part (a) were the same except with a value of 50 at i instead of 30, then in part (c) the 50 would be where the 30 is. Because 50 is greater than both 20 and 40, no swap would occur and the algorithm would terminate.

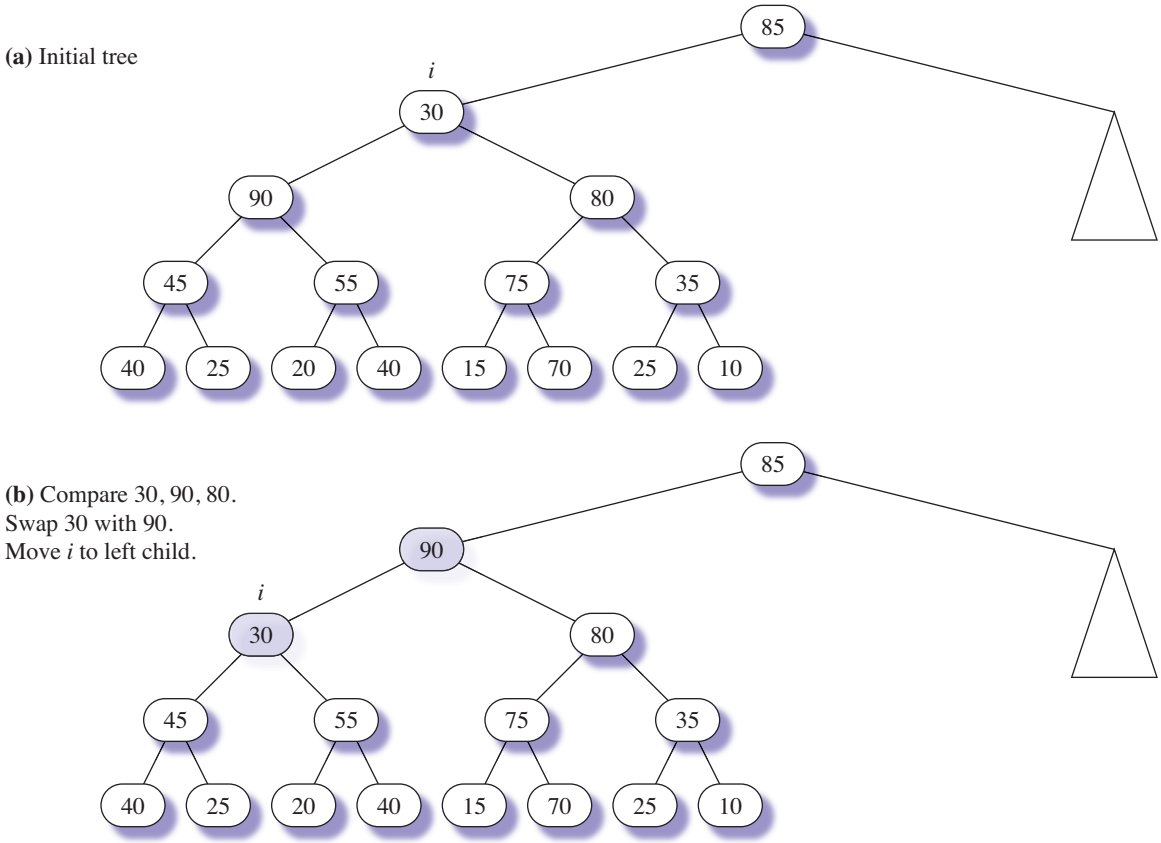


Figure 4.24 A trace of `siftDown()`. The triangle represents the entire right subtree of the root.

The postcondition is guaranteed if the algorithm terminates before $a[i]$ sifts all the way down to a leaf. The algorithm can only stop when $a[i]$ is at least the values of its left and right subtrees. Hence, the max-heap property is satisfied at $a[i]$. By the precondition, the max-heap property is already satisfied at the left and right subtrees. Therefore, the tree rooted at the original value of i is a max-heap.

The execution time of `siftDown()` can be as short as $\Theta(1)$ in the best case. If i is originally at a leaf, or if the initial value of $a[i]$ is greater than the values at its left and right subtrees there will be no recursive calls. In the worst case, i is equal to l at the root and $a[i]$ will sift all the way to a leaf. Therefore, the number of recursive calls is on the order of the height of the tree. Because the height of a complete binary tree with n elements is on the order of $\lg n$, the worst case asymptotic bound on `siftDown()` is $T(n) = \Theta(\lg n)$.

Figure 4.26 is a trace of the constructor of `heapSorter` to build the initial heap. Part (a) of the figure corresponds to Figure 4.4(a) and part (e) of the figure corresponds to Figure 4.4(b). The algorithm consists of the single loop

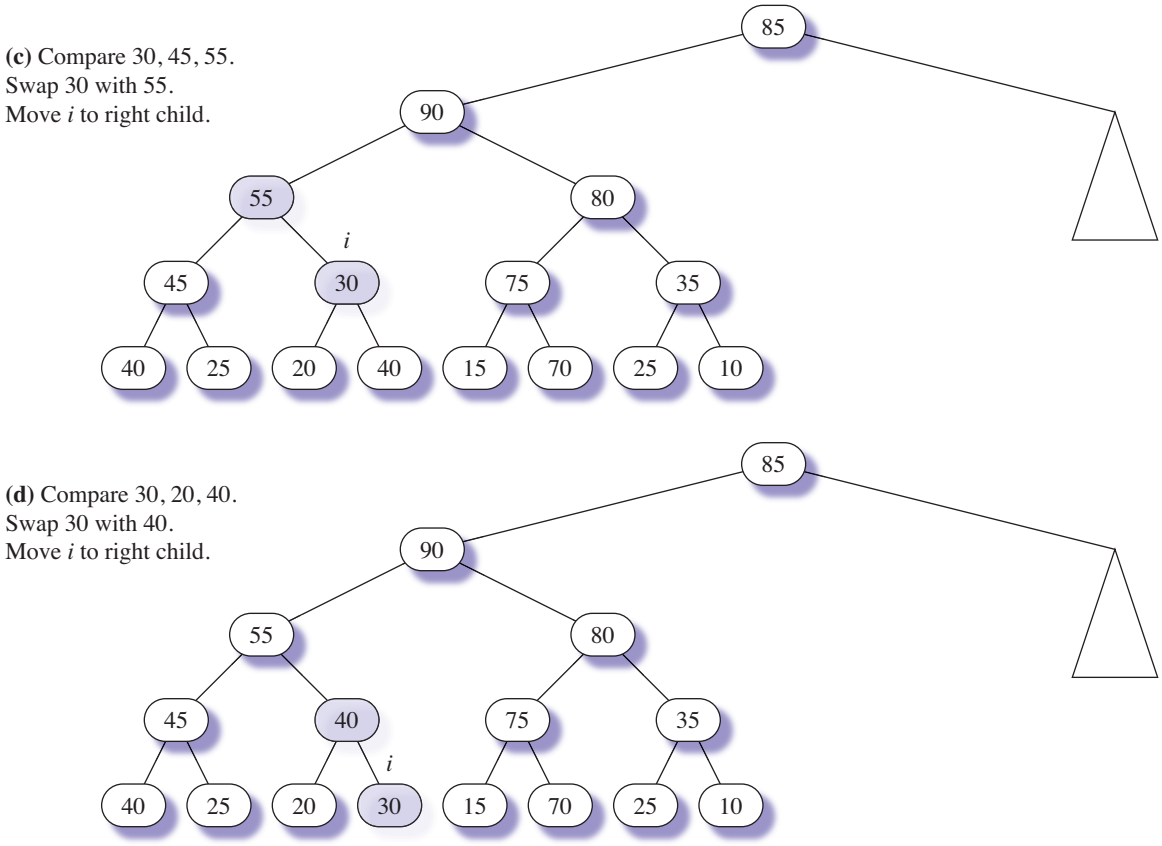
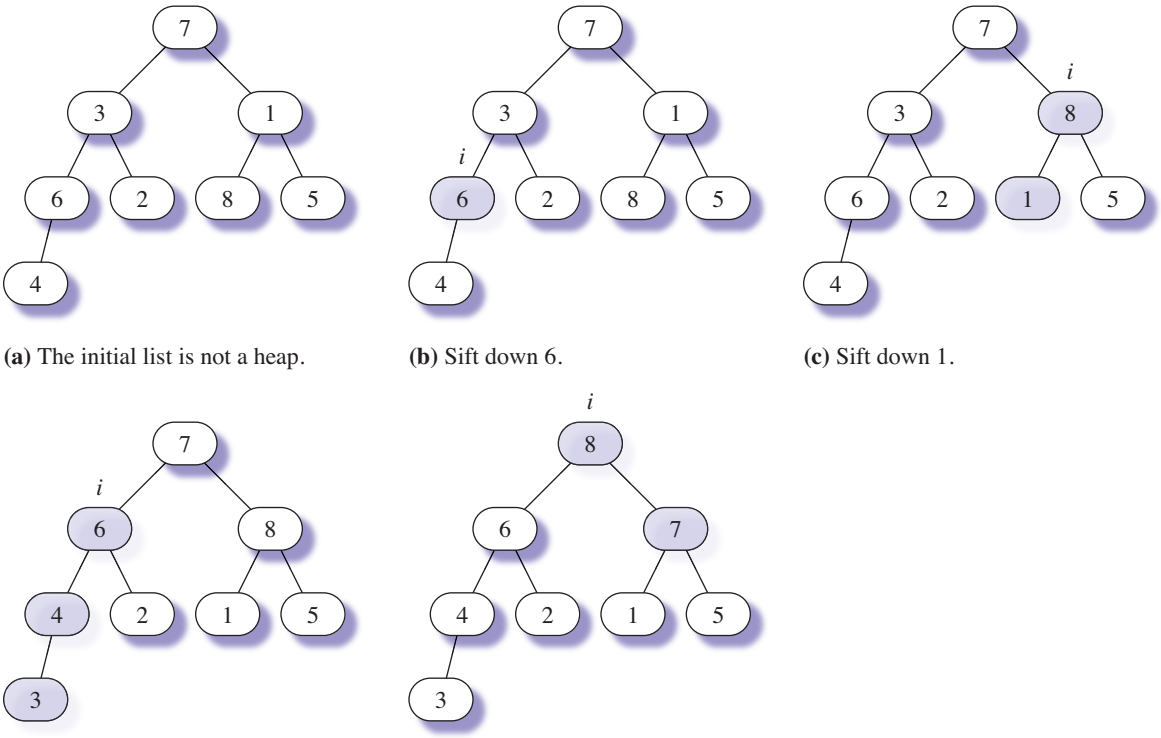


Figure 4.25 A trace of `siftDown()` (continued).

```
for (int i = (lo + hi - 1) / 2; i >= lo; i--) {
    siftDown(a, lo, i, hi);
}
```

Figure 4.26(b) shows integer i initialized to $\lfloor (l+h-1)/2 \rfloor$ the maximum index of a node with a child. All the nodes with an index greater than i have the max-heap property, because they are leaves. As it turns out, 6 does not move initially because it is greater than 4. Part (c) shows i decremented by one, and then $a[i]$ sifted down. Part (d) shows i decremented by one again. The precondition for `siftDown()` is met here because the left child of $a[i]$, even though it is not a leaf, was processed by `siftDown()` earlier in part (b). As i works its way up the tree, the precondition for `siftDown()` is always met because the children of $a[i]$ are either leaves or have been previously processed by `siftDown()`.

What is the asymptotic execution time for the build-heap algorithm? The computation for the initial value of i is $\lfloor (l+h-1)/2 \rfloor$, which is on the order of $n/2$. That is, about half of the nodes in the array segment are processed by `siftDown()`. Each time a node is processed, the time for `siftDown()` is $\Theta(\lg n)$ in the worst case. So, the time



(a) The initial list is not a heap. **(b)** Sift down 6. **(c)** Sift down 1.

(d) Sift down 3. **(e)** Sift down 7.

Figure 4.26 A trace of the constructor of `heapSorter` to build the initial heap.

to build the heap is $\Theta(n/2) \cdot \Theta(\lg n)$. Because $1/2$ is a constant, $\Theta(n/2) = \Theta(n)$, and the worst case time to build the heap has bound $T(n) = \Theta(n \lg n)$.

The time to build the heap is actually better than this, because most calls to `siftDown()` are for values of i that are close to the bottom of the tree. It can be proved that the actual time to build the heap is $T(n) = \Theta(n)$, although the proof is beyond the scope of this book.

The template method for `sort()` is

```
sort(a, lo, hi)
    if (lo < hi)
        int mid;
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
        sort(a, mid, hi)
        join(a, lo, mid, hi)
```

Figure 4.21 shows the implementation of the `split()` operation for the heap sort. Here is a pseudocode description.


```
Swap a[hi] and a[lo]
siftDown(a, lo, lo, hi - 1)
mid = hi
```

Because `split()` sets `mid` to `hi`, the second recursive call in `sort()` does no work. Figure 4.21 also shows that `join()` does no work. Effectively, then, the template method for `sort()` does the following processing.

```
sort(a, lo, hi)
    if (lo < hi)
        int mid;
        split(a, lo, mid, hi)
        sort(a, lo, mid - 1)
```

The algorithm swaps `a[lo]` and `a[hi]`, sifts down `a[lo]` but not including `a[hi]` in the heap, and then repeats with the smaller heap not including `a[i]`.

Figures 4.27 and 4.28 show a trace of heap sort after the initial build-heap operation. Figure 4.27(a) is identical to Figure 4.4(b), which shows the heap after the preprocessing. Figure 4.27(c) is identical to Figure 4.4(c), which shows the heap after the first split operation.

Figure 4.27(b) shows the first swap of `a[lo]` and `a[hi]`. The link between 8 and its parent is missing, because the sift shown in part (c) comes from the call

```
siftDown(a, lo, lo, hi - 1)
```

The last parameter `hi-1` makes the sift not include the last element of the heap. Then, after `split()` sets `mid` to `hi`, the recursive call to sort

```
sort(a, lo, mid - 1)
```

repeats the process without the last element in the heap.

The effect of the call to `split()` is to move the largest element in the array segment to the last position in the segment. Both heap sort and selection sort place the largest last in their split operations. The crucial difference is the structure of the remaining unsorted part of the segment. Figure 4.3(b) shows that the selection sort has

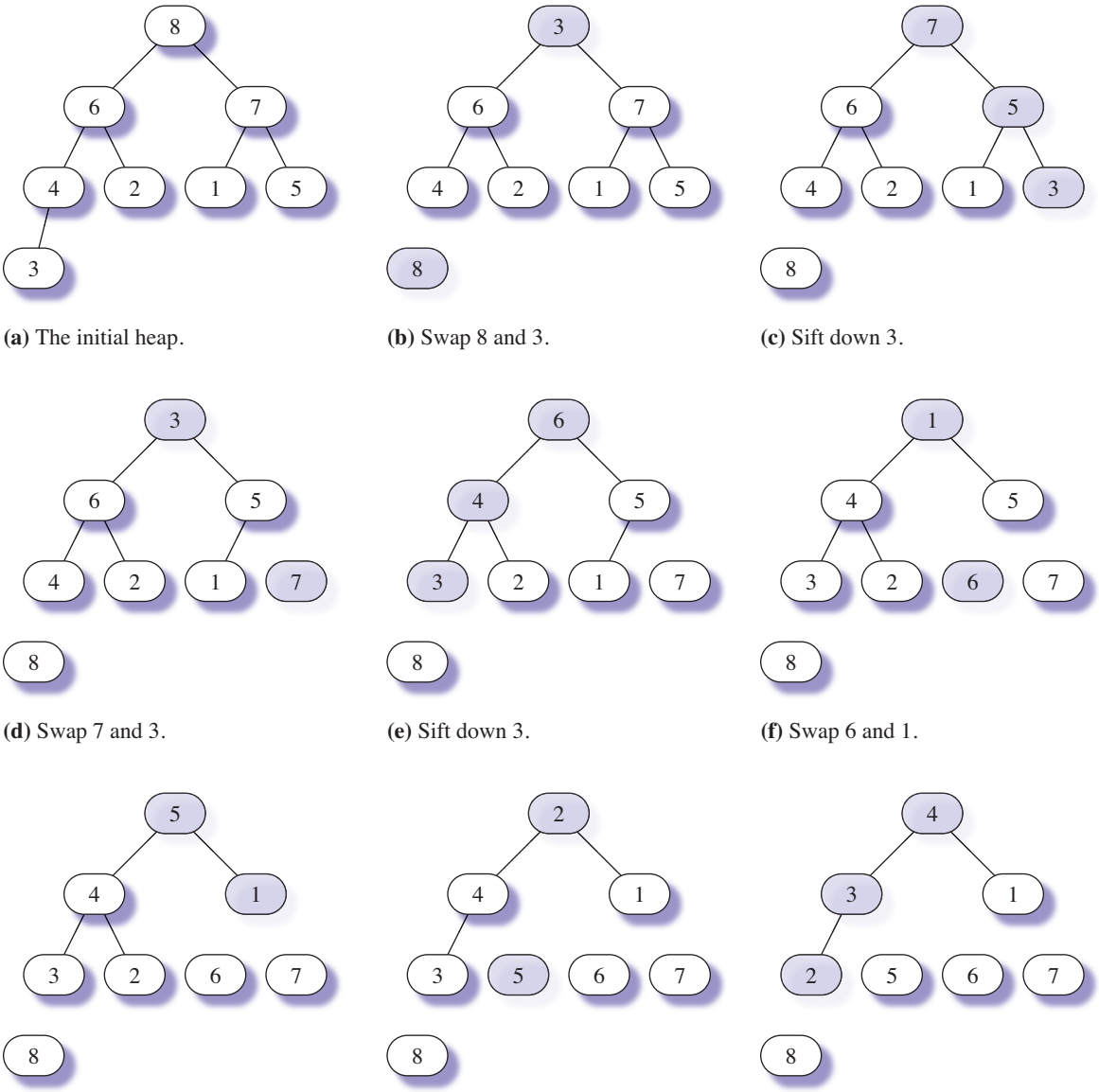
7 3 1 6 2 4 5

for the unsorted segment after its split of 8, while Figure 4.27(c) shows that heap sort has

7 6 5 4 2 1 3

after its split of 8. This seemingly innocuous difference, however, has ramifications for the relative performance of the algorithms. The fact that the second unsorted segment is a heap, and therefore always has its maximum value at `a[lo]`, makes the split operation more efficient.

Figure 4.27 parts (d) and (e) show the swap and sift down for the second call to `split()`. Before the swap, the root of the max-heap contains the maximum element 7. After the swap, the 7 is in its final resting place. After the sift down of 3, the maximum of the smaller heap, which is 6, is at the root.



(a) The initial heap. **(b)** Swap 8 and 3. **(c)** Sift down 3.

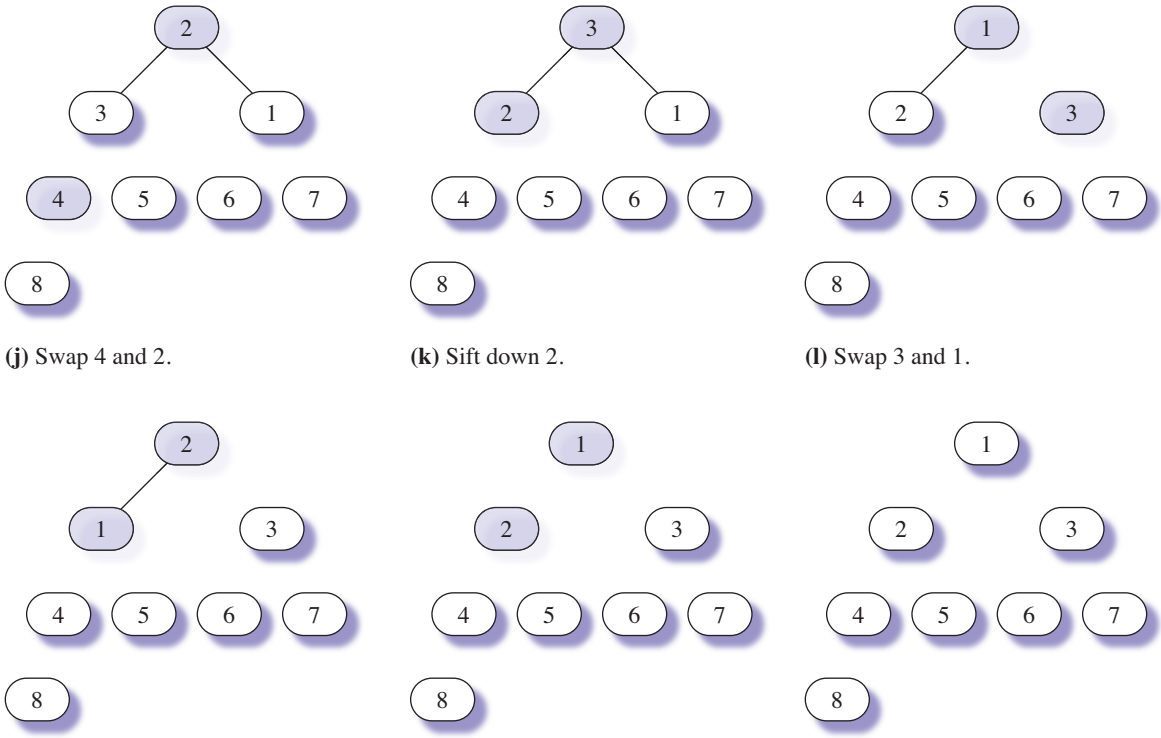
(d) Swap 7 and 3. **(e)** Sift down 3. **(f)** Swap 6 and 1.

(g) Sift down 1. **(h)** Swap 5 and 2. **(i)** Sift down 2.

Figure 4.27 A trace of the heap sort algorithm.

The swap and sift operations execute in pairs until the last pair executes with two elements. After each sift, shown in parts (c), (e), (g), (i), (k), and (m), the maximum value of the heap is at the root, ready to be split into its final resting place by the next swap. The last pair of operations shown in parts (n) and (o) leave the segment sorted.

What is the asymptotic execution time of the heap sort? It consists of the call to the



(j) Swap 4 and 2. **(k)** Sift down 2. **(l)** Swap 3 and 1.

(m) Sift down 1. **(n)** Swap 2 and 1. **(o)** Sift down 1.

Figure 4.28 A trace of the heap sort algorithm (continued).

constructor to build the initial heap, which in the worst case is $\Theta(n \lg n)$. This processing happens only once. It then calls the split operation on the order of n times. For each call, the sift down operation executes in the worst case $\Theta(\lg n)$ times. So, the total time for heap sort is

$$\Theta(n \lg n) + \Theta(n) \cdot \Theta(\lg n) = \Theta(n \lg n) + \Theta(n \lg n) = \Theta(n \lg n)$$

Recall the claim (without proof) in the preceding analysis of the build heap operation that it executes in time $\Theta(n)$. With this assumption, the analysis for the the execution time of heap sort is

$$\Theta(n) + \Theta(n) \cdot \Theta(\lg n) = \Theta(n) + \Theta(n \lg n) = \Theta(n \lg n)$$

This result is identical because the $n \lg n$ term predominates over the n term. The conclusion is that heap sort executes with asymptotic bound $T(n) = \Theta(n \lg n)$.

Here is a proof that `split()` is correct. The precondition is maxHeap$(a[l..h])$.

Proof: There are three conjuncts in the postcondition.

First conjunct: maxHeap$(a[l..h-1])$. After the swap of $a[h]$ and $a[l]$, $a[l..h-1]$ is no longer a max-heap. By the precondition, it was a max-heap before the swap. So,

the only node that violates the max-heap property after the swap is the root $a[l]$. By the postcondition of `siftDown()`, the sift restores the max-heap property of $a[l..h-1]$

Second conjunct: $a[h]$ is equal to the old $a[l]$. The initial swap sets $a[h]$ to the old $a[l]$. Because the subsequent call to `siftDown()` does not include $a[h]$, $a[h]$ cannot be affected by it.

Third conjunct: $m = h$. This is satisfied, because the last executable statement in `split()`, which is `mid = hi`, sets m to the value of h. ∎

The recursive implementation of heap sort presented in this chapter is rarely used in practice. One objective of this book is to teach OO design patterns. This chapter presents the template method pattern, because all the common sort algorithms, including merge sort, fit the Merritt classification taxonomy, which is a natural application of that pattern.

In practice, the designer of a software library would present the user with a sort function and would not ask the user which sort algorithm to use. Furthermore, it is straightforward to implement heap sort without recursion. The trace in Figures 4.27 and 4.28 shows that the swap-sift pair of operations executes linearly from the end of the array segment to the beginning. That processing can be implemented with a loop. Also, Exercise 7 is a problem for the student to implement `siftDown` without recursion.

4.3 Performance Metrics and Decorator Patterns

Here is a summary of the theoretical asymptotic bounds of the comparison sort algorithms.

- Merge sort — $\Theta(n \lg n)$
- Worst case quick sort — $\Theta(n^2)$
- Average case quick sort — $\Theta(n \lg n)$
- Selection sort — $\Theta(n^2)$
- Insertion sort — $\Theta(n^2)$
- Heap sort — $\Theta(n \lg n)$

These bounds are all based on theoretical statement execution counts. In practice, the user of any program is interested in the elapsed time for it to execute. To experimentally verify the above theoretical asymptotic bounds you could try to time the execution with calls to the system clock. The elapsed time is difficult to measure, however, because a program runs under control of an operating system that typically manages scores of processes simultaneously. Time measurements depend on external factors that are hard to control such as the number of other jobs in the system and the operating system's scheduling policy.

The remainder of this chapter applies decorator design patterns to perform statement execution counts experimentally without relying on the system clock. In software design, a decoration is a functionality that is added to an existing operation to produce some desired side effect.

While it would be difficult to experimentally count the execution of every statement in a sorting program, a decorator pattern can count the execution of some parts of some statements. The execution count is the side effect the pattern adds to the program. Because the pattern does not provide a total count of the number of CPU cycles to do the sort, the count it does provide is a relative indication of the performance of the algorithm.

A metric is a standard of measurement. This section describes decorator patterns for the following performance metrics.

- The number of array element comparisons.
- The number of array element assignments.
- The number of array element probes.

An example of an element comparison is in the comparison statement

```
if (a[i] < a[child])
```

in the `siftDown()` method for heap sort. A decorator to count the number of comparisons would increase the count by one for each execution of the `if` guard. A decorator to count the number of probes would increase the count by two for each execution of the `if` guard, once for the access to `a[i]` and once for the access to `a[child]`. An example of an element assignment is the assignment statement

```
a[mid] = a[j];
```

in the `split()` method of quick sort. A decorator to count the number of assignments would increase the count by one for each execution of the assignment statement.

This chapter describes two decorators that collect data using the above metrics.

- Parametric decoration — for comparison and assignment metrics.
- Decorator design pattern — for the probe metric.

Both decorator patterns use C++ templates in conjunction with function overloading. Strictly speaking, parametric decoration is not an OO design pattern as it contains no inheritance. Implementation of the probe metric uses the OO decorator design pattern.

A comparison sort bound

Obviously, the sort algorithms that are $\Theta(n \lg n)$ are faster than the ones that are $\Theta(n^2)$. A natural question is, Can any sort algorithm be designed that is even faster than $\Theta(n \lg n)$?

All these sorts described in this chapter are based on pair-wise comparisons of values in an array. A remarkable theorem about algorithms that depend on such pair-wise comparisons is the following.

Theorem, Comparison sort bound: Any sort of n elements that is based on element comparison and exchange is $\Omega(n \lg n)$.

This theorem assumes the comparison metric as a measure of the performance of the sort algorithm.

Because Ω is a lower bound, the theorem asserts that *no* comparison sort algorithm can be better than $\Theta(n \lg n)$. According to the theorem, the $\Theta(n \lg n)$ sorts of this chapter are absolutely the best that can be achieved by anyone, anywhere, any time. How can we know that someone in the future will not be clever enough to improve on the asymptotic behavior of these sorts? After all, an infinite number of algorithm designs are possible, and many have not yet been discovered.

In spite of the apparent difficulty of proving the impossibility of a particular algorithm design, this section gives such a mathematical proof. The proof depends on two facts — a lemma that specifies a lower bound on $n!$ and the binary decision tree model

for comparison sort algorithms. Here is the lemma and its proof.

Lemma: $n! \geq (n/2)^{n/2}$

Proof:

$$
\begin{aligned}
& n! \\
= \quad & \langle \text{Definition of } n!, \text{ assuming } n \text{ is even} \rangle \\
& n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (\tfrac{n}{2}+1) \cdot (\tfrac{n}{2}) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 \\
= \quad & \langle \text{Multiplication is symmetric, } a \cdot b = b \cdot a \rangle \\
& 1n \cdot 2(n-1) \cdot 3(n-2) \cdot \ldots \cdot (\tfrac{n}{2})(\tfrac{n}{2}+1) \\
\geq \quad & \langle \text{Product of } n/2 \text{ terms, each of which is greater than } n/2 \rangle \\
& (n/2)^{n/2} \quad \blacksquare
\end{aligned}
$$

It is an exercise for the student to show that if n is odd, the lower bound on $n!$ is $(n/2)^{(n+1)/2}$. The comparison sort bound theorem is still valid with this bound on $n!$.

The binary decision tree model is a tree representation of the decision process that must occur with any comparison sort. For example, suppose the algorithm is to sort three input values — a, b, and c. For the algorithm to be correct, it must produce the correct result for all possible initial permutations. There are six possible permutations of three values, namely

abc acb bac bca cab cba

A decision in a comparison sort algorithm is an executable statement that is usually the guard of an `if` statement like the above `if` statement in the `siftDown()` method of the heap sort. In general, there exists a minimum number of such comparisons that must be made if the algorithm is to be correct for all possible input permutations.

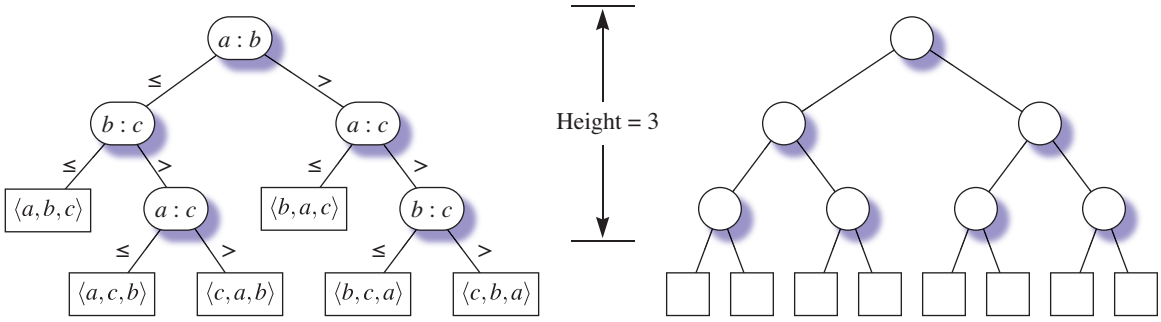
Figure 4.29(a) shows the decision tree for this three-input example. The notation $a : b$ at the root of the decision tree represents the guard of an `if` statement or a loop that compares values a and b. The branch to the left is for the processing that must be done if $a \leq b$ and the branch to the right is the processing that must be done if $a > b$.

The left child of the root $b : c$ represents a comparison of b and c. Its left child represents the outcome if $b \leq c$. Because the first test determines that $a \leq b$ and the second test determines that $b \leq c$, the conclusion is that the sorted order is $\langle a, b, c \rangle$ indicated by the leaf. On the other hand, if the second test determines that $b > c$, the only possible conclusion is that b has the maximum value. To determine whether the correct sort is $\langle a, c, b \rangle$ or $\langle c, a, b \rangle$ requires the additional comparison of a and c.

In general, any sort algorithm that depends on pairwise comparison of elements must be modeled by a decision tree that has $n!$ leaves, because there are $n!$ possible initial permutations of n elements and the algorithm must be able to sort the elements correctly for all possible input permutations. Therefore, the decision tree must have $n!$ leaves.

Figure 4.29 shows that the height of the decision tree equals the maximum number of comparisons to sort the list. It could take fewer. In the above example, if $a \leq b$ and $b \leq c$ the list is sorted with only two comparisons. But, if $a \leq b$ and $b > c$, a third comparison is required.

The Ω bound exists, because to sort n elements the decision tree must have $n!$ leaves. What is the smallest height a tree can have if it has $n!$ leaves? Part (b) of the figures shows a *perfect* binary tree of height three. A perfect binary tree of height h is the tree with the maximum number of leaves. In general, a height-h tree has at most 2^h leaves. For



(a) Maximum decisions = 3. Number of leaves = 6.

(b) Maximum decisions = 3. Number of leaves = 8.

Figure 4.29 Two decision trees. Part (a) is the decision tree for sorting three elements. Part (b) is a full tree containing the maximum number of leaves for a tree of height 3.

example, the height-3 tree in Part (a) has six leaves, and the height-3 tree in Part (b) has eight leaves, which is the maximum.

Because the decision tree must have $n!$ leaves, $2^h \geq n!$. Here is a proof of the comparison sort bound starting with this relationship.

Proof:

$2^h \geq n!$

$=$ ⟨Math, take the log base 2 of both sides⟩

$h \geq \lg(n!)$

$\Rightarrow$ ⟨Lemma, $n! \geq (n/2)^{n/2}$, and monotonicity of lg⟩

$h \geq \lg(n/2)^{n/2}$

$=$ ⟨Math, $\lg a^b = b \lg a$⟩

$h \geq (n/2)\lg(n/2)$

$=$ ⟨$h =$ The number of decisions⟩

The number of decisions $\geq (n/2)\lg(n/2)$

$=$ ⟨Definition of Ω⟩

The number of decisions $= \Omega((n/2)\lg(n/2))$

$=$ ⟨Property of Ω, dropping the constants⟩

The number of decisions $= \Omega(n \lg n)$ ∎

The proof uses the correspondence between the number of decisions in the decision tree model and the number of comparisons in the sort algorithm.

Comparison and assignment metrics with parametric decoration

[*To be written. The remainder of this chapter contains project code from the dp4ds distribution interspersed with paragraphs of blind text as filler.*]

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.


```
// ========= Counter =========
class Counter {
private:
    int _delta;
    int _count;

public:
    // Post: _count == 0 && _delta == cost.
    Counter(int cost = 1) :
        _delta(cost),
        _count(0) {
    }

    // Post: _count is returned.
    int getCount() const {
        return _count;
    }

    // Post: _count is increased by _delta.
    void update() {
        _count += _delta;
    }

    // Post: _count == 0.
    void clear() {
        _count = 0;
    }

    // Post: _delta == cost.
    void setDelta(int cost) {
        _delta = cost;
    }
};
```

Figure 4.30 `Counter.hpp`. The `Counter` class for counting assignments, comparisons, and probes in sort algorithms.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed


```
// ========= CAMetrics =========
template<class T>
class CAMetrics {
private:
    static Counter _compareCount;
    static Counter _assignCount;

    T _data;

public:
    CAMetrics(); // for array initialization.
    CAMetrics(CAMetrics const &rhs);
    T toT() const;
    void setT(const T &source);
    bool operator<(CAMetrics const &rhs);
    bool operator<(CAMetrics const &rhs) const;
    bool operator<=(CAMetrics const &rhs);
    bool operator<=(CAMetrics const &rhs) const;
    bool operator==(CAMetrics const &rhs);
    bool operator==(CAMetrics const &rhs) const;
    bool operator!=(CAMetrics const &rhs);
    bool operator!=(CAMetrics const &rhs) const;
    bool operator>=(CAMetrics const &rhs);
    bool operator>=(CAMetrics const &rhs) const;
    bool operator>(CAMetrics const &rhs);
    bool operator>(CAMetrics const &rhs) const;
    CAMetrics &operator=(CAMetrics const &rhs);

    static int getCompareCount();
    static int getAssignCount();
    static void clearCompareCount();
    static void clearAssignCount();
    static void setCost(int cost);

public:
    friend ostream &operator<<(ostream &os, CAMetrics<T> const &rhs) {
        os << rhs._data;
        return os;
    }

    friend istream &operator>>(istream &is, CAMetrics<T> &rhs) {
        rhs._assignCount.update();
        is >> rhs._data;
        return is;
    }
};
```

Figure 4.31 `CAMetrics.hpp`. The `CAMetrics` class, which overloads the comparison and assignment operators. The program listing continues in the next figure.


```
// ========= The static counters =========
template<class T>
Counter CAMetrics<T>::_compareCount;

template<class T>
Counter CAMetrics<T>::_assignCount;


// ========= Constructors =========
template<class T>
CAMetrics<T>::CAMetrics(void) {
}

template<class T>
CAMetrics<T>::CAMetrics(CAMetrics const &rhs) {
    _assignCount.update();
    _data = rhs._data;
}

// ========= toT =========
template<class T>
T CAMetrics<T>::toT() const {
    return _data;
}

// ========= setT =========
template<class T>
void CAMetrics<T>::setT(T const &source) {
    _assignCount.update();
    _data = source;
}
```

Figure 4.32 `CAMetrics.hpp` (continued). The static counters, the constructors, and the `toT()`, and `setT()` methods for the `CAMetrics` class. The program listing continues in the next figure.

text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.


```
// ========= Comparison operators =========
template<class T>
bool CAMetrics<T>::operator<(CAMetrics<T> const &rhs) {
    _compareCount.update();
    return _data < rhs._data;
}

template<class T>
bool CAMetrics<T>::operator<(CAMetrics<T> const &rhs) const {
    _compareCount.update();
    return _data < rhs._data;
}

template<class T>
bool CAMetrics<T>::operator<=(CAMetrics<T> const &rhs) {
    _compareCount.update();
    return _data <= rhs._data;
}

template<class T>
bool CAMetrics<T>::operator<=(CAMetrics<T> const &rhs) const {
    _compareCount.update();
    return _data <= rhs._data;
}
```

Figure 4.33 `CAMetrics.hpp` (continued). Implementation of the first two overloaded comparison operators for the `CAMetrics` class. The other four operators are similar and are not shown. The program listing continues in the next figure.

The probe metric with the decorator design pattern

[*Decorate the ArrayT class to intercept calls to the [ ] operator to count the probes*]

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This


```
// ========= operator= =========
template<class T>
CAMetrics<T> &CAMetrics<T>::operator=(CAMetrics<T> const &rhs) {
    _assignCount.update();
    if (_data != rhs._data ) {
        _data = rhs._data;
    }
    return *this;
}

// ========= Access methods =========
template<class T>
int  CAMetrics<T>::getCompareCount() {
    return _compareCount.getCount();
}

template<class T>
int  CAMetrics<T>::getAssignCount() {
    return _assignCount.getCount();
}

// ========= Clear methods =========
template<class T>
void  CAMetrics<T>::clearCompareCount() {
    _compareCount.clear();
}

template<class T>
void CAMetrics<T>::clearAssignCount() {
    _assignCount.clear();
}

// ========= setCost =========
template<class T>
void CAMetrics<T>::setCost(int cost) {
    _compareCount.setDelta(cost);
}
```

Figure 4.34 `CAMetrics.hpp` (continued). Implementation of the overloaded assignment operator for the `CAMetrics` class. This concludes the program listing.

text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really?


```
// ========= ArrProbe =========
template<class T>
class ArrProbe : public ASeq<T> {
private:
    ArrayT<T> *_arrayT; // Does not own _arrayT.
    Counter _probeCount;

public:
    ArrProbe(ArrayT<T> *arrayT);
    virtual ~ArrProbe();

    T &operator[](int i) override; // For read/write.
    T const &operator[](int i) const override; // For read-only.

    int cap() const override { return _arrayT->cap(); }

    void clearProbeCount() { _probeCount.clear(); }

    int getProbeCount() const { return _probeCount.getCount(); }

    void setCost(int cost) { _probeCount.setDelta(cost); }

private:
    ArrProbe(ArrProbe const &rhs); // Disabled.
    ArrProbe &operator=(ArrProbe const &rhs); // Disabled.
};
```

Figure 4.35 Declaration of the `ArrProbe` class to count the number of array probes.

Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.


```
// ========= Constructor =========
template<class T>
ArrProbe<T>::ArrProbe(ArrayT<T>* arrayT) :
    _arrayT(arrayT),
    _probeCount(1) {
}

// ========= Destructor =========
template<class T>
ArrProbe<T>::~ArrProbe() {
    _arrayT = nullptr;
}

// ========= operator[] =========
template<class T>
T &ArrProbe<T>::operator[](int i) {
    _probeCount.update();
    return (*_arrayT)[i];
}

template<class T>
T const &ArrProbe<T>::operator[](int i) const {
    return (*_arrayT)[i];
}
```

Figure 4.36 Implementation of the `ArrProbe` class to count the number of array probes.

Exercises

4–1 Modify `SortIntMain.cpp` in the dp4ds distribution `SortInt` project to sort an array of strings with the C++ `String` class. Test it with the strings from an excerpt of Hamlet in the distribution software (file `hamlet`). How many lines of code in how many different files do you need to change to sort strings instead of integers?

4–2 Implement function `join()` in class `MergeSorter`, Figure 4.9, in the distribution software. Merge the elements from array `a` into array `_tempA`, then copy the merged elements from `_tempA` back into `a`.

4–3 The text shows that merge sort and best-case quick sort have the same recurrence. Prove using mathematical induction that $T(n) = n \lg n$ is the solution to the following recurrence, which has the same form.

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n > 2. \end{cases}$$

4–4 Prove that the loop in `split()` for `SelectSorter` in Figure 4.18 is correct.

4–5 **(a)** Implement function `join()` in class `InsertSorter`, Figure 4.19, provided in the distribution software. **(b)** Prove that your implementation is correct.

4–6 The text shows that selection sort, insertion sort, and worst-case quick sort have the same recurrence. Prove using mathematical induction that $T(n) = \frac{1}{2}n^2 + \frac{1}{2}n + 1$ is the solution to the following recurrence, which has the same form.

$$T(n) = \begin{cases} 1 & \text{if } n = 0, \\ T(n-1) + n & \text{if } n > 0. \end{cases}$$

4–7 Rewrite function `siftDown()` in `Heapifier.hpp`, Figure 4.22, without recursion. Replace the recursive call with a `while` loop.

(a) Optimize your loop by not doing the pair-wise exchange of array elements within the loop. Instead, save the value to be sifted down before you enter the loop. Then, as you sift down, do a single copy from child to parent within the loop. The body of your loop should have five probes (that is, element accesses with `[]`) and two element comparisons. To test your implementation use it to sort a list of numbers with the heap sorter.

(b) Optimize your `siftDown()` function from part(a) even more by eliminating the test to terminate in the interior of the heap. Instead, sift the item all the way down to a leaf. Then, call `siftUp()` to get the item now at a leaf position up to its proper place. The body of your `siftDown()` loop should have four probes and one element comparison.

4–8 **(a)** Prove that the lower bound of $n!$ is $(n/2)^{(n+1)/2}$ when n is odd. **(b)** Show that the comparison sort bound theorem is still valid with this bound on $n!$.

