Chapter 11

Sorting Algorithms

This chapter discusses algorithms for sorting a set of n items. This might seem like a strange topic for a book on data structures, but there are several good reasons for including it here. The most obvious reason is that two of these sorting algorithms (quicksort and heap-sort) are intimately related to two of the data structures we have already studied (random binary search trees and heaps, respectively).

The first part of this chapter discusses algorithms that sort using only comparisons and presents three algorithms that run in $O(n \log n)$ time. As it turns out, all three algorithms are asymptotically optimal; no algorithm that uses only comparisons can avoid doing roughly $n \log n$ comparisons in the worst case and even the average case.

Before continuing, we should note that any of the SSet or priority Queue implementations presented in previous chapters can also be used to obtain an $O(n \log n)$ time sorting algorithm. For example, we can sort n items by performing n add(x) operations followed by n remove() operations on a BinaryHeap or MeldableHeap. Alternatively, we can use n add(x) operations on any of the binary search tree data structures and then perform an in-order traversal (Exercise 6.8) to extract the elements in sorted order. However, in both cases we go through a lot of overhead to build a structure that is never fully used. Sorting is such an important problem that it is worthwhile developing direct methods that are as fast, simple, and space-efficient as possible.

The second part of this chapter shows that, if we allow other operations besides comparisons, then all bets are off. Indeed, by using arrayindexing, it is possible to sort a set of n integers in the range $\{0,\ldots,n^c-1\}$ in $O(cn)$ time.

11.1 Comparison-Based Sorting

In this section, we present three sorting algorithms: merge-sort, quicksort, and heap-sort. Each of these algorithms takes an input array a and sorts the elements of a into non-decreasing order in $O(n\log n)$ (expected) time. These algorithms are all *comparison-based*. Their second argument, c, is a Comparator that implements the compare(a,b) method. These algorithms don't care what type of data is being sorted; the only operation they do on the data is comparisons using the compare(a,b) method. Recall, from Section 1.2.4, that compare(a,b) returns a negative value if $a < b$, a positive value if $a > b$, and zero if $a = b$.

11.1.1 Merge-Sort

The *merge-sort* algorithm is a classic example of recursive divide and conquer: If the length of a is at most 1, then a is already sorted, so we do nothing. Otherwise, we split a into two halves, $a0 = a[0],\ldots,a[n/2-1]$ and $a1 = a[n/2],\ldots,a[n-1]$. We recursively sort a0 and a1, and then we merge (the now sorted) a0 and a1 to get our fully sorted array a:

```
Algorithms
<T> void mergeSort(T[] a, Comparator<T> c) {
  if (a.length <= 1) return;
  T[] a0 = Arrays.copyOfRange(a, 0, a.length/2);
  T[] a1 = Arrays.copyOfRange(a, a.length/2, a.length);
  mergeSort(a0, c);
  mergeSort(a1, c);
  merge(a0, a1, a, c);
}
```

An example is shown in Figure 11.1.

Compared to sorting, merging the two sorted arrays a0 and a1 is fairly easy. We add elements to a one at a time. If a0 or a1 is empty, then we add the next elements from the other (non-empty) array. Otherwise, we

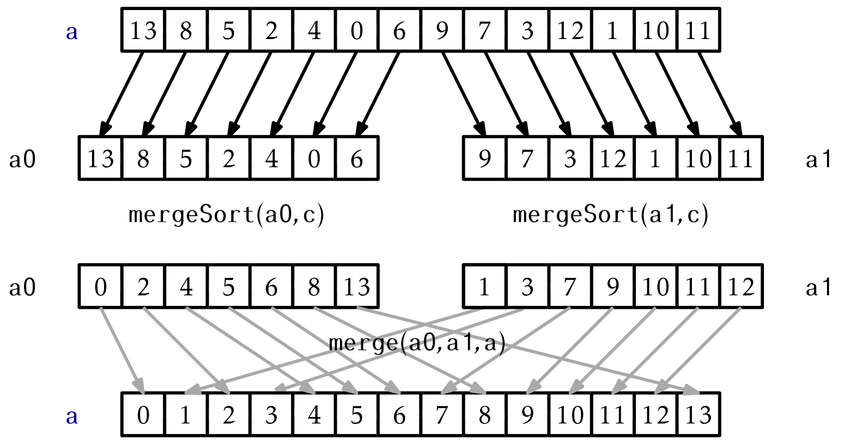

Figure 11.1: The execution of mergeSort(a,c)

take the minimum of the next element in a0 and the next element in a1 and add it to a:

```
                              Algorithms
<T> void merge(T[] a0, T[] a1, T[] a, Comparator<T> c) {
  int i0 = 0, i1 = 0;
  for (int i = 0; i < a.length; i++) {
    if (i0 == a0.length)
      a[i] = a1[i1++];
    else if (i1 == a1.length)
      a[i] = a0[i0++];
    else if (compare(a0[i0], a1[i1]) < 0)
      a[i] = a0[i0++];
    else
      a[i] = a1[i1++];
  }
}
```

Notice that the merge(a0,a1,a,c) algorithm performs at most $n-1$ comparisons before running out of elements in one of a0 or a1.

To understand the running-time of merge-sort, it is easiest to think of it in terms of its recursion tree. Suppose for now that n is a power of

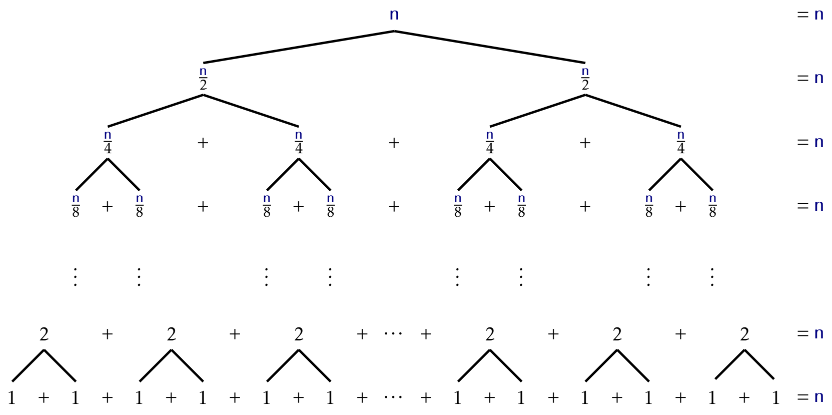

Figure 11.2: The merge-sort recursion tree.

two, so that $n = 2^{\log n}$, and $\log n$ is an integer. Refer to Figure 11.2. Merge-sort turns the problem of sorting n elements into two problems, each of sorting $n/2$ elements. These two subproblem are then turned into two problems each, for a total of four subproblems, each of size $n/4$. These four subproblems become eight subproblems, each of size $n/8$, and so on. At the bottom of this process, $n/2$ subproblems, each of size two, are converted into n problems, each of size one. For each subproblem of size $n/2^i$, the time spent merging and copying data is $O(n/2^i)$. Since there are 2^i subproblems of size $n/2^i$, the total time spent working on problems of size 2^i, not counting recursive calls, is

$$2^i \times O(n/2^i) = O(n) \ .$$

Therefore, the total amount of time taken by merge-sort is

$$\sum_{i=0}^{\log n} O(n) = O(n \log n) \ .$$

The proof of the following theorem is based on preceding analysis, but has to be a little more careful to deal with the cases where n is not a power of 2.

Theorem 11.1. *The* `mergeSort(a,c)` *algorithm runs in* $O(n \log n)$ *time and performs at most* $n \log n$ *comparisons.*

Proof. The proof is by induction on n. The base case, in which $n = 1$, is trivial; when presented with an array of length 0 or 1 the algorithm simply returns without performing any comparisons.

Merging two sorted lists of total length n requires at most $n-1$ comparisons. Let $C(n)$ denote the maximum number of comparisons performed by `mergeSort(a,c)` on an array a of length n. If n is even, then we apply the inductive hypothesis to the two subproblems and obtain

$$\begin{aligned} C(n) &\le n - 1 + 2C(n/2) \\ &\le n - 1 + 2((n/2)\log(n/2)) \\ &= n - 1 + n\log(n/2) \\ &= n - 1 + n\log n - n \\ &< n\log n \enspace . \end{aligned}$$

The case where n is odd is slightly more complicated. For this case, we use two inequalities that are easy to verify:

$$\log(x+1) \le \log(x) + 1 \enspace , \tag{11.1}$$

for all $x \ge 1$ and

$$\log(x+1/2) + \log(x-1/2) \le 2\log(x) \enspace , \tag{11.2}$$

for all $x \ge 1/2$. Inequality (11.1) comes from the fact that $\log(x) + 1 = \log(2x)$ while (11.2) follows from the fact that log is a concave function. With these tools in hand we have, for odd n,

$$\begin{aligned} C(n) &\le n - 1 + C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) \\ &\le n - 1 + \lceil n/2 \rceil \log\lceil n/2 \rceil + \lfloor n/2 \rfloor \log\lfloor n/2 \rfloor \\ &= n - 1 + (n/2 + 1/2)\log(n/2 + 1/2) + (n/2 - 1/2)\log(n/2 - 1/2) \\ &\le n - 1 + n\log(n/2) + (1/2)(\log(n/2 + 1/2) - \log(n/2 - 1/2)) \\ &\le n - 1 + n\log(n/2) + 1/2 \\ &< n + n\log(n/2) \\ &= n + n(\log n - 1) \\ &= n\log n \enspace . \end{aligned}$$

□

11.1.2 Quicksort

The *quicksort* algorithm is another classic divide and conquer algorithm. Unlike merge-sort, which does merging after solving the two subproblems, quicksort does all of its work upfront.

Quicksort is simple to describe: Pick a random *pivot* element, x, from a; partition a into the set of elements less than x, the set of elements equal to x, and the set of elements greater than x; and, finally, recursively sort the first and third sets in this partition. An example is shown in Figure 11.3.

Algorithms
```
<T> void quickSort(T[] a, Comparator<T> c) {
  quickSort(a, 0, a.length, c);
}
<T> void quickSort(T[] a, int i, int n, Comparator<T> c) {
  if (n <= 1) return;
  T x = a[i + rand.nextInt(n)];
  int p = i-1, j = i, q = i+n;
  // a[i..p]<x,  a[p+1..q-1]??x, a[q..i+n-1]>x
  while (j < q) {
    int comp = compare(a[j], x);
    if (comp < 0) {        // move to beginning of array
      swap(a, j++, ++p);
    } else if (comp > 0) {
      swap(a, j, --q);  // move to end of array
    } else {
      j++;              // keep in the middle
    }
  }
  // a[i..p]<x,  a[p+1..q-1]=x, a[q..i+n-1]>x
  quickSort(a, i, p-i+1, c);
  quickSort(a, q, n-(q-i), c);
}
```

All of this is done in place, so that instead of making copies of subarrays being sorted, the quickSort(a, i, n, c) method only sorts the subarray a[i], ..., a[i + n − 1]. Initially, this method is invoked with the arguments quickSort(a, 0, a.length, c).

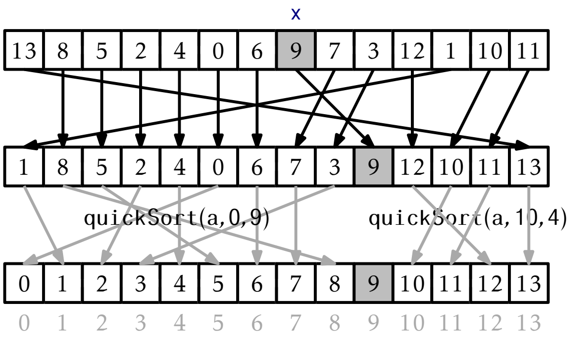

Figure 11.3: An example execution of quickSort(a,0,14,c)

At the heart of the quicksort algorithm is the in-place partitioning algorithm. This algorithm, without using any extra space, swaps elements in a and computes indices p and q so that

$$a[i]\begin{cases} < x & \text{if } 0 \le i \le p \\ = x & \text{if } p < i < q \\ > x & \text{if } q \le i \le n-1 \end{cases}$$

This partitioning, which is done by the while loop in the code, works by iteratively increasing p and decreasing q while maintaining the first and last of these conditions. At each step, the element at position j is either moved to the front, left where it is, or moved to the back. In the first two cases, j is incremented, while in the last case, j is not incremented since the new element at position j has not yet been processed.

Quicksort is very closely related to the random binary search trees studied in Section 7.1. In fact, if the input to quicksort consists of n distinct elements, then the quicksort recursion tree is a random binary search tree. To see this, recall that when constructing a random binary search tree the first thing we do is pick a random element x and make it the root of the tree. After this, every element will eventually be compared to x, with smaller elements going into the left subtree and larger elements into the right.

In quicksort, we select a random element x and immediately compare everything to x, putting the smaller elements at the beginning of the array and larger elements at the end of the array. Quicksort then recursively sorts the beginning of the array and the end of the array, while the random binary search tree recursively inserts smaller elements in the left subtree of the root and larger elements in the right subtree of the root.

The above correspondence between random binary search trees and quicksort means that we can translate Lemma 7.1 to a statement about quicksort:

Lemma 11.1. *When quicksort is called to sort an array containing the integers* $0,\dots,n-1$, *the expected number of times element* i *is compared to a pivot element is at most* $H_{i+1} + H_{n-i}$.

A little summing up of harmonic numbers gives us the following theorem about the running time of quicksort:

Theorem 11.2. *When quicksort is called to sort an array containing* n *distinct elements, the expected number of comparisons performed is at most* $2n\ln n + O(n)$.

Proof. Let T be the number of comparisons performed by quicksort when sorting n distinct elements. Using Lemma 11.1 and linearity of expectation, we have:

$$\begin{aligned}
\mathrm{E}[T] &= \sum_{i=0}^{n-1}(H_{i+1} + H_{n-i}) \\
&= 2\sum_{i=1}^{n} H_i \\
&\le 2\sum_{i=1}^{n} H_n \\
&\le 2n\ln n + 2n = 2n\ln n + O(n)
\end{aligned}$$ □

Theorem 11.3 describes the case where the elements being sorted are all distinct. When the input array, a, contains duplicate elements, the expected running time of quicksort is no worse, and can be even better; any time a duplicate element x is chosen as a pivot, all occurrences of x get grouped together and do not take part in either of the two subproblems.

Theorem 11.3. *The* `quickSort(a,c)` *method runs in* $O(n\log n)$ *expected time and the expected number of comparisons it performs is at most* $2n\ln n + O(n)$.

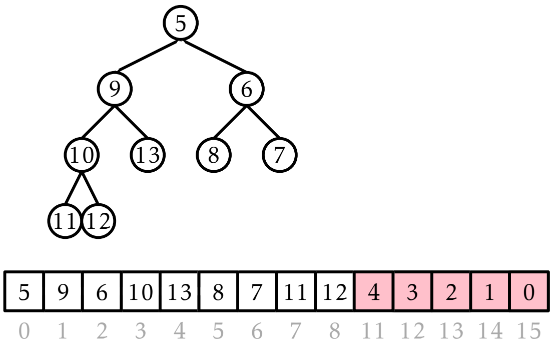

Figure 11.4: A snapshot of the execution of heapSort(a,c). The shaded part of the array is already sorted. The unshaded part is a BinaryHeap. During the next iteration, element 5 will be placed into array location 8.

11.1.3 Heap-sort

The *heap-sort* algorithm is another in-place sorting algorithm. Heap-sort uses the binary heaps discussed in Section 10.1. Recall that the BinaryHeap data structure represents a heap using a single array. The heap-sort algorithm converts the input array a into a heap and then repeatedly extracts the minimum value.

More specifically, a heap stores n elements in an array, a, at array locations $a[0],\ldots,a[n-1]$ with the smallest value stored at the root, $a[0]$. After transforming a into a BinaryHeap, the heap-sort algorithm repeatedly swaps $a[0]$ and $a[n-1]$, decrements n, and calls trickleDown(0) so that $a[0],\ldots,a[n-2]$ once again are a valid heap representation. When this process ends (because $n=0$) the elements of a are stored in decreasing order, so a is reversed to obtain the final sorted order.[1] Figure 11.4 shows an example of the execution of heapSort(a,c).

BinaryHeap
```
<T> void sort(T[] a, Comparator<T> c) {
  BinaryHeap<T> h = new BinaryHeap<T>(a, c);
  while (h.n > 1) {
```

[1]The algorithm could alternatively redefine the compare(x,y) function so that the heap sort algorithm stores the elements directly in ascending order.

```
    h.swap(--h.n, 0);
    h.trickleDown(0);
  }
  Collections.reverse(Arrays.asList(a));
}
```

A key subroutine in heap sort is the constructor for turning an unsorted array a into a heap. It would be easy to do this in $O(n \log n)$ time by repeatedly calling the BinaryHeap add(x) method, but we can do better by using a bottom-up algorithm. Recall that, in a binary heap, the children of a[i] are stored at positions a[2i + 1] and a[2i + 2]. This implies that the elements $a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ have no children. In other words, each of $a[\lfloor n/2 \rfloor], \ldots, a[n-1]$ is a sub-heap of size 1. Now, working backwards, we can call trickleDown(i) for each $i \in \{\lfloor n/2 \rfloor - 1, \ldots, 0\}$. This works, because by the time we call trickleDown(i), each of the two children of a[i] are the root of a sub-heap, so calling trickleDown(i) makes a[i] into the root of its own subheap.

```
BinaryHeap
BinaryHeap(T[] a, Comparator<T> c) {
  this.c = c;
  this.a = a;
  n = a.length;
  for (int i = n/2-1; i >= 0; i--) {
    trickleDown(i);
  }
}
```

The interesting thing about this bottom-up strategy is that it is more efficient than calling add(x) n times. To see this, notice that, for n/2 elements, we do no work at all, for n/4 elements, we call trickleDown(i) on a subheap rooted at a[i] and whose height is one, for n/8 elements, we call trickleDown(i) on a subheap whose height is two, and so on. Since the work done by trickleDown(i) is proportional to the height of the sub-heap rooted at a[i], this means that the total work done is at most

$$\sum_{i=1}^{\log n} O((i-1)n/2^i) \le \sum_{i=1}^{\infty} O(in/2^i) = O(n)\sum_{i=1}^{\infty} i/2^i = O(2n) = O(n) \ .$$

The second-last equality follows by recognizing that the sum $\sum_{i=1}^{\infty} i/2^i$ is equal, by definition of expected value, to the expected number of times we toss a coin up to and including the first time the coin comes up as heads and applying Lemma 4.2.

The following theorem describes the performance of `heapSort(a,c)`.

Theorem 11.4. *The* `heapSort(a,c)` *method runs in* $O(\mathtt{n}\log\mathtt{n})$ *time and performs at most* $2\mathtt{n}\log\mathtt{n}+O(\mathtt{n})$ *comparisons.*

Proof. The algorithm runs in three steps: (1) transforming a into a heap, (2) repeatedly extracting the minimum element from a, and (3) reversing the elements in a. We have just argued that step 1 takes $O(\mathtt{n})$ time and performs $O(\mathtt{n})$ comparisons. Step 3 takes $O(\mathtt{n})$ time and performs no comparisons. Step 2 performs n calls to `trickleDown(0)`. The ith such call operates on a heap of size $\mathtt{n}-i$ and performs at most $2\log(\mathtt{n}-i)$ comparisons. Summing this over i gives

$$\sum_{i=0}^{\mathtt{n}-i} 2\log(\mathtt{n}-i) \le \sum_{i=0}^{\mathtt{n}-i} 2\log\mathtt{n} = 2\mathtt{n}\log\mathtt{n}$$

Adding the number of comparisons performed in each of the three steps completes the proof. □

11.1.4 A Lower-Bound for Comparison-Based Sorting

We have now seen three comparison-based sorting algorithms that each run in $O(\mathtt{n}\log\mathtt{n})$ time. By now, we should be wondering if faster algorithms exist. The short answer to this question is no. If the only operations allowed on the elements of a are comparisons, then no algorithm can avoid doing roughly $\mathtt{n}\log\mathtt{n}$ comparisons. This is not difficult to prove, but requires a little imagination. Ultimately, it follows from the fact that

$$\log(\mathtt{n}!) = \log\mathtt{n} + \log(\mathtt{n}-1) + \dots + \log(1) = \mathtt{n}\log\mathtt{n} - O(\mathtt{n}) \ .$$

(Proving this fact is left as Exercise 11.11.)

We will start by focusing our attention on deterministic algorithms like merge-sort and heap-sort and on a particular fixed value of n. Imagine such an algorithm is being used to sort n distinct elements. The key

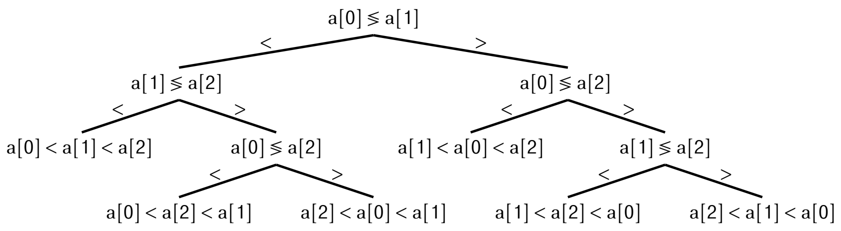

Figure 11.5: A comparison tree for sorting an array $a[0], a[1], a[2]$ of length $n = 3$.

to proving the lower-bound is to observe that, for a deterministic algorithm with a fixed value of n, the first pair of elements that are compared is always the same. For example, in heapSort(a,c), when n is even, the first call to trickleDown(i) is with $i = n/2-1$ and the first comparison is between elements $a[n/2-1]$ and $a[n-1]$.

Since all input elements are distinct, this first comparison has only two possible outcomes. The second comparison done by the algorithm may depend on the outcome of the first comparison. The third comparison may depend on the results of the first two, and so on. In this way, any deterministic comparison-based sorting algorithm can be viewed as a rooted binary *comparison tree*. Each internal node, u, of this tree is labelled with a pair of indices $u.i$ and $u.j$. If $a[u.i] < a[u.j]$ the algorithm proceeds to the left subtree, otherwise it proceeds to the right subtree. Each leaf w of this tree is labelled with a permutation $w.p[0],\ldots,w.p[n-1]$ of $0,\ldots,n-1$. This permutation represents the one that is required to sort a if the comparison tree reaches this leaf. That is,

$$a[w.p[0]] < a[w.p[1]] < \cdots < a[w.p[n-1]] \ .$$

An example of a comparison tree for an array of size $n = 3$ is shown in Figure 11.5.

The comparison tree for a sorting algorithm tells us everything about the algorithm. It tells us exactly the sequence of comparisons that will be performed for any input array, a, having n distinct elements and it tells us how the algorithm will reorder a in order to sort it. Consequently, the comparison tree must have at least $n!$ leaves; if not, then there are two distinct permutations that lead to the same leaf; therefore, the algorithm

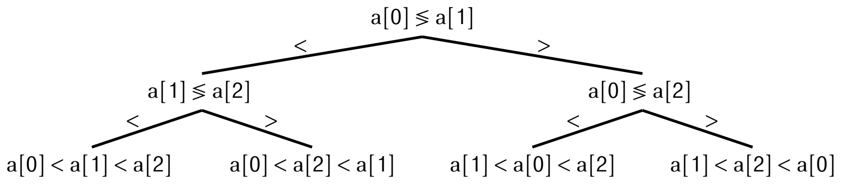

Figure 11.6: A comparison tree that does not correctly sort every input permutation.

does not correctly sort at least one of these permutations.

For example, the comparison tree in Figure 11.6 has only $4 < 3! = 6$ leaves. Inspecting this tree, we see that the two input arrays $3,1,2$ and $3,2,1$ both lead to the rightmost leaf. On the input $3,1,2$ this leaf correctly outputs $a[1] = 1, a[2] = 2, a[0] = 3$. However, on the input $3,2,1$, this node incorrectly outputs $a[1] = 2, a[2] = 1, a[0] = 3$. This discussion leads to the primary lower-bound for comparison-based algorithms.

Theorem 11.5. *For any deterministic comparison-based sorting algorithm $\mathcal{A}$ and any integer $n \geq 1$, there exists an input array a of length n such that $\mathcal{A}$ performs at least $\log(n!) = n\log n - O(n)$ comparisons when sorting a.*

Proof. By the preceding discussion, the comparison tree defined by $\mathcal{A}$ must have at least $n!$ leaves. An easy inductive proof shows that any binary tree with k leaves has a height of at least $\log k$. Therefore, the comparison tree for $\mathcal{A}$ has a leaf, w, with a depth of at least $\log(n!)$ and there is an input array a that leads to this leaf. The input array a is an input for which $\mathcal{A}$ does at least $\log(n!)$ comparisons. □

Theorem 11.5 deals with deterministic algorithms like merge-sort and heap-sort, but doesn't tell us anything about randomized algorithms like quicksort. Could a randomized algorithm beat the $\log(n!)$ lower bound on the number of comparisons? The answer, again, is no. Again, the way to prove it is to think differently about what a randomized algorithm is.

In the following discussion, we will assume that our decision trees have been "cleaned up" in the following way: Any node that can not be reached by some input array a is removed. This cleaning up implies that the tree has exactly $n!$ leaves. It has at least $n!$ leaves because, otherwise, it could not sort correctly. It has at most $n!$ leaves since each of the possible $n!$ permutation of n distinct elements follows exactly one root to leaf path in the decision tree.

We can think of a randomized sorting algorithm, $\mathcal{R}$, as a deterministic algorithm that takes two inputs: The input array a that should be sorted and a long sequence $b = b_1, b_2, b_3, \ldots, b_m$ of random real numbers in the range $[0,1]$. The random numbers provide the randomization for the algorithm. When the algorithm wants to toss a coin or make a random choice, it does so by using some element from b. For example, to compute the index of the first pivot in quicksort, the algorithm could use the formula $\lfloor nb_1 \rfloor$.

Now, notice that if we fix b to some particular sequence $\hat{b}$ then $\mathcal{R}$ becomes a deterministic sorting algorithm, $\mathcal{R}(\hat{b})$, that has an associated comparison tree, $\mathcal{T}(\hat{b})$. Next, notice that if we select a to be a random permutation of $\{1,\ldots,n\}$, then this is equivalent to selecting a random leaf, w, from the $n!$ leaves of $\mathcal{T}(\hat{b})$.

Exercise 11.13 asks you to prove that, if we select a random leaf from any binary tree with k leaves, then the expected depth of that leaf is at least $\log k$. Therefore, the expected number of comparisons performed by the (deterministic) algorithm $\mathcal{R}(\hat{b})$ when given an input array containing a random permutation of $\{1,\ldots,n\}$ is at least $\log(n!)$. Finally, notice that this is true for every choice of $\hat{b}$, therefore it holds even for $\mathcal{R}$. This completes the proof of the lower-bound for randomized algorithms.

Theorem 11.6. *For any integer $n \geq 1$ and any (deterministic or randomized) comparison-based sorting algorithm $\mathcal{A}$, the expected number of comparisons done by $\mathcal{A}$ when sorting a random permutation of $\{1,\ldots,n\}$ is at least* $\log(n!) = n\log n - O(n)$.

11.2 Counting Sort and Radix Sort

In this section we study two sorting algorithms that are not comparison-based. Specialized for sorting small integers, these algorithms elude the lower-bounds of Theorem 11.5 by using (parts of) the elements in a as indices into an array. Consider a statement of the form

$$\texttt{c[a[i]]} = 1 \ .$$

This statement executes in constant time, but has c.length possible different outcomes, depending on the value of a[i]. This means that the execution of an algorithm that makes such a statement cannot be modelled as a binary tree. Ultimately, this is the reason that the algorithms in this section are able to sort faster than comparison-based algorithms.

11.2.1 Counting Sort

Suppose we have an input array a consisting of n integers, each in the range $0,\ldots,\mathtt{k}-1$. The *counting-sort* algorithm sorts a using an auxiliary array c of counters. It outputs a sorted version of a as an auxiliary array b.

The idea behind counting-sort is simple: For each $\mathtt{i} \in \{0,\ldots,\mathtt{k}-1\}$, count the number of occurrences of i in a and store this in c[i]. Now, after sorting, the output will look like c[0] occurrences of 0, followed by c[1] occurrences of 1, followed by c[2] occurrences of 2,..., followed by c[k − 1] occurrences of k − 1. The code that does this is very slick, and its execution is illustrated in Figure 11.7:

Algorithms
```
int[] countingSort(int[] a, int k) {
  int c[] = new int[k];
  for (int i = 0; i < a.length; i++)
    c[a[i]]++;
  for (int i = 1; i < k; i++)
    c[i] += c[i-1];
  int b[] = new int[a.length];
  for (int i = a.length-1; i >= 0; i--)
    b[--c[a[i]]] = a[i];
  return b;
}
```

The first for loop in this code sets each counter c[i] so that it counts the number of occurrences of i in a. By using the values of a as indices,

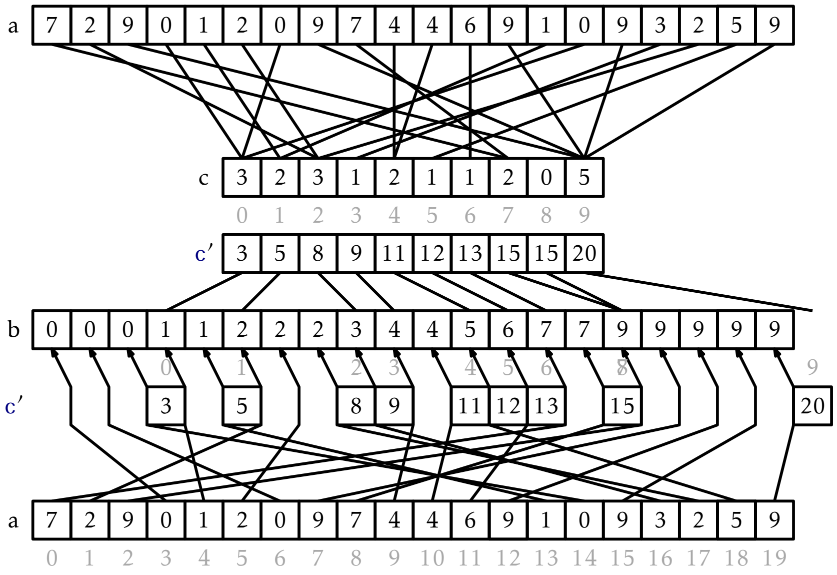

Figure 11.7: The operation of counting sort on an array of length $n = 20$ that stores integers $0,\ldots,k-1 = 9$.

these counters can all be computed in $O(\mathtt{n})$ time with a single for loop. At this point, we could use c to fill in the output array b directly. However, this would not work if the elements of a have associated data. Therefore we spend a little extra effort to copy the elements of a into b.

The next for loop, which takes $O(\mathtt{k})$ time, computes a running-sum of the counters so that c[i] becomes the number of elements in a that are less than or equal to i. In particular, for every $\mathtt{i} \in \{0,\ldots,\mathtt{k}-1\}$, the output array, b, will have

$$\mathtt{b}[\mathtt{c}[\mathtt{i}-1]] = \mathtt{b}[\mathtt{c}[\mathtt{i}-1]+1] = \cdots = \mathtt{b}[\mathtt{c}[\mathtt{i}]-1] = \mathtt{i} \enspace .$$

Finally, the algorithm scans a backwards to place its elements, in order, into an output array b. When scanning, the element $\mathtt{a}[\mathtt{i}] = \mathtt{j}$ is placed at location $\mathtt{b}[\mathtt{c}[\mathtt{j}]-1]$ and the value c[j] is decremented.

Theorem 11.7. *The* countingSort(a,k) *method can sort an array* a *containing* n *integers in the set* $\{0,\ldots,\mathtt{k}-1\}$ *in* $O(\mathtt{n}+\mathtt{k})$ *time.*

The counting-sort algorithm has the nice property of being *stable*; it preserves the relative order of equal elements. If two elements a[i] and a[j] have the same value, and $\mathtt{i} < \mathtt{j}$ then a[i] will appear before a[j] in b. This will be useful in the next section.

11.2.2 Radix-Sort

Counting-sort is very efficient for sorting an array of integers when the length, n, of the array is not much smaller than the maximum value, $\mathtt{k}-1$, that appears in the array. The *radix-sort* algorithm, which we now describe, uses several passes of counting-sort to allow for a much greater range of maximum values.

Radix-sort sorts w-bit integers by using w/d passes of counting-sort to sort these integers d bits at a time.[2] More precisely, radix sort first sorts the integers by their least significant d bits, then their next significant d bits, and so on until, in the last pass, the integers are sorted by their most significant d bits.

[2]We assume that d divides w, otherwise we can always increase w to $\mathtt{d}\lceil \mathtt{w}/\mathtt{d}\rceil$.

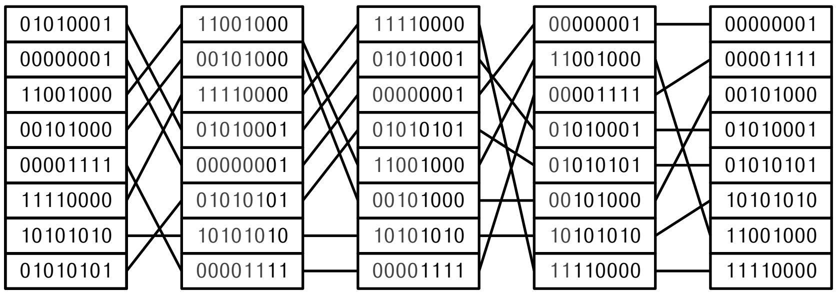

Figure 11.8: Using radixsort to sort w = 8-bit integers by using 4 passes of counting sort on d = 2-bit integers.

Algorithms
```
int[] radixSort(int[] a) {
  int[] b = null;
  for (int p = 0; p < w/d; p++) {
    int c[] = new int[1<<d];
    // the next three for loops implement counting-sort
    b = new int[a.length];
    for (int i = 0; i < a.length; i++)
      c[(a[i] >> d*p)&((1<<d)-1)]++;
    for (int i = 1; i < 1<<d; i++)
      c[i] += c[i-1];
    for (int i = a.length-1; i >= 0; i--)
      b[--c[(a[i] >> d*p)&((1<<d)-1)]] = a[i];
    a = b;
  }
  return b;
}
```

(In this code, the expression $(a[i]{>>}d*p)\&((1{<<}d)-1)$ extracts the integer whose binary representation is given by bits $(p+1)d-1,\ldots,pd$ of a[i].) An example of the steps of this algorithm is shown in Figure 11.8.

This remarkable algorithm sorts correctly because counting-sort is a stable sorting algorithm. If $x < y$ are two elements of a, and the most significant bit at which x differs from y has index r, then x will be placed before y during pass $\lfloor r/d \rfloor$ and subsequent passes will not change the relative order of x and y.

Radix-sort performs w/d passes of counting-sort. Each pass requires $O(n + 2^d)$ time. Therefore, the performance of radix-sort is given by the following theorem.

Theorem 11.8. *For any integer* $d > 0$, *the* `radixSort(a,k)` *method can sort an array* a *containing* n w*-bit integers in* $O((w/d)(n + 2^d))$ *time.*

If we think, instead, of the elements of the array being in the range $\{0,\ldots,n^c - 1\}$, and take $d = \lceil \log n \rceil$ we obtain the following version of Theorem 11.8.

Corollary 11.1. *The* `radixSort(a,k)` *method can sort an array* a *containing* n *integer values in the range* $\{0,\ldots,n^c - 1\}$ *in* $O(cn)$ *time.*

11.3 Discussion and Exercises

Sorting is *the* fundamental algorithmic problem in computer science, and it has a long history. Knuth [48] attributes the merge-sort algorithm to von Neumann (1945). Quicksort is due to Hoare [39]. The original heap-sort algorithm is due to Williams [78], but the version presented here (in which the heap is constructed bottom-up in $O(n)$ time) is due to Floyd [28]. Lower-bounds for comparison-based sorting appear to be folklore. The following table summarizes the performance of these comparison-based algorithms:

	comparisons	in-place
Merge-sort	$n\log n$ worst-case	No
Quicksort	$1.38n\log n + O(n)$ expected	Yes
Heap-sort	$2n\log n + O(n)$ worst-case	Yes

Each of these comparison-based algorithms has its advantages and disadvantages. Merge-sort does the fewest comparisons and does not rely on randomization. Unfortunately, it uses an auxilliary array during its merge phase. Allocating this array can be expensive and is a potential point of failure if memory is limited. Quicksort is an *in-place* algorithm and is a close second in terms of the number of comparisons, but is randomized, so this running time is not always guaranteed. Heap-sort does the most comparisons, but it is in-place and deterministic.

There is one setting in which merge-sort is a clear-winner; this occurs when sorting a linked-list. In this case, the auxiliary array is not needed; two sorted linked lists are very easily merged into a single sorted linked-list by pointer manipulations (see Exercise 11.2).

The counting-sort and radix-sort algorithms described here are due to Seward [68, Section 2.4.6]. However, variants of radix-sort have been used since the 1920s to sort punch cards using punched card sorting machines. These machines can sort a stack of cards into two piles based on the existence (or not) of a hole in a specific location on the card. Repeating this process for different hole locations gives an implementation of radix-sort.

Finally, we note that counting sort and radix-sort can be used to sort other types of numbers besides non-negative integers. Straightforward modifications of counting sort can sort integers, in any interval $\{a,\ldots,b\}$, in $O(\mathtt{n}+b-a)$ time. Similarly, radix sort can sort integers in the same interval in $O(\mathtt{n}(\log_{\mathtt{n}}(b-a))$ time. Finally, both of these algorithms can also be used to sort floating point numbers in the IEEE 754 floating point format. This is because the IEEE format is designed to allow the comparison of two floating point numbers by comparing their values as if they were integers in a signed-magnitude binary representation [2].

Exercise 11.1. Illustrate the execution of merge-sort and heap-sort on an input array containing 1,7,4,6,2,8,3,5. Give a sample illustration of one possible execution of quicksort on the same array.

Exercise 11.2. Implement a version of the merge-sort algorithm that sorts a `DLList` without using an auxiliary array. (See Exercise 3.13.)

Exercise 11.3. Some implementations of `quickSort(a,i,n,c)` always use `a[i]` as a pivot. Give an example of an input array of length `n` in which such an implementation would perform $\binom{\mathtt{n}}{2}$ comparisons.

Exercise 11.4. Some implementations of `quickSort(a,i,n,c)` always use `a[i+n/2]` as a pivot. Given an example of an input array of length `n` in which such an implementation would perform $\binom{\mathtt{n}}{2}$ comparisons.

Exercise 11.5. Show that, for any implementation of `quickSort(a,i,n,c)` that chooses a pivot deterministically, without first looking at any values in $a[i], \ldots, a[i+n-1]$, there exists an input array of length n that causes this implementation to perform $\binom{n}{2}$ comparisons.

Exercise 11.6. Design a Comparator, c, that you could pass as an argument to quickSort(a,i,n,c) and that would cause quicksort to perform $\binom{n}{2}$ comparisons. (Hint: Your comparator does not actually need to look at the values being compared.)

Exercise 11.7. Analyze the expected number of comparisons done by Quicksort a little more carefully than the proof of Theorem 11.3. In particular, show that the expected number of comparisons is $2nH_n - n + H_n$.

Exercise 11.8. Describe an input array that causes heap sort to perform at least $2n\log n - O(n)$ comparisons. Justify your answer.

Exercise 11.9. The heap sort implementation described here sorts the elements into reverse sorted order and then reverses the array. This last step could be avoided by defining a new Comparator that negates the results of the input Comparator, c. Explain why this would not be a good optimization. (Hint: Consider how many negations would need to be done in relation to how long it takes to reverse the array.)

Exercise 11.10. Find another pair of permutations of 1, 2, 3 that are not correctly sorted by the comparison tree in Figure 11.6.

Exercise 11.11. Prove that $\log n! = n\log n - O(n)$.

Exercise 11.12. Prove that a binary tree with k leaves has height at least $\log k$.

Exercise 11.13. Prove that, if we pick a random leaf from a binary tree with k leaves, then the expected height of this leaf is at least $\log k$.

Exercise 11.14. The implementation of radixSort(a,k) given here works when the input array, a contains only non-negative integers. Extend this implementation so that it also works correctly when a contains both negative and non-negative integers.