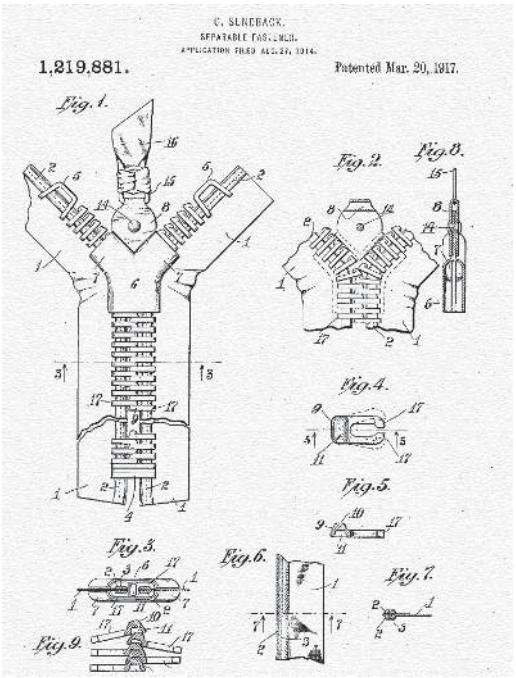
Chapter 8 Merge-Sort and Quick-Sort



Separable Fastener, U.S. Patent 1,219,881, 1917. Public domain image.

Contents

Recall that in the sorting problem, we are given a collection of n comparable items and we are asked to place them in order.

Efficient sorting algorithms have a wide range of applications, including uses in the underlying technologies behind Internet search engines Sorting arises, for example, in the steps needed to build a data structure that allows a search engine to quickly return a list of the documents that contain a given keyword. This data structure is known as the ***inverted file***.

Given a collection of documents (such as the web pages found by a search engine when it was crawling the web), an inverted file is a lookup table that matches words to the documents containing those words. It is indexed by keywords found in the document collection and it includes, for each keyword, a list of the documents where that keyword appears. This lookup table allows the search engine to quickly return the documents containing a given keyword just by doing a lookup for that keyword in the table.

The data is not given in this format, however. In fact, it starts out as just a collection of documents. To construct an inverted file, we must first create a set of keyword-document pairs, (k, d), where k is a keyword and d is the identifier for a document where k appears. Fortunately, constructing such a set of keyword-document pairs is fairly easy—we can simply scan the contents of each document, d, and output a pair, (k, d), for each keyword, k, found in d. Thus, we can assume that we can start with a set of keyword-document pairs, from which we then want to build an inverted file.

Building an inverted file data structure from a set of keyword-document pairs requires that we bring together, for each keyword, k, all the documents that contain k. Bringing all such documents together can be done simply by sorting the set of keyword-document pairs by keywords. This places all the (k, d) pairs with the same keyword, k, right next to one another in the output list. From this sorted list, it is then a simple computation to scan the list and build a lookup table of documents for each keyword that appears in this sorted list.

In practice, most search engines go one step further, and not only sort the set of keyword-document pairs by keywords, but break ties between (k, d) pairs with the same keyword, k, by using a relevance (or ranking) score for the document, d, as a secondary key (following a ***lexicographic*** ordering rule). Taking this approach implies that the (k, d) pairs with the same keyword, k, will be ordered in the sorted list according to the score of their document, d. Thus, having a fast algorithm for sorting can be very helpful for a search engine, particularly if that algorithm is designed to work quickly for large sets of input data. We study two sorting algorithms in this chapter. The first algorithm, called merge-sort, is ideally suited for very large data sets, which must be accessed on a hard drive or network storage system. The second algorithm, called quick-sort, is very efficient for moderately large data sets that fit in the computer's main memory (RAM).

8.1 Merge-Sort

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

8.1.1 Divide-and-Conquer

Merge-sort is based on an algorithmic paradigm called ***divide-and-conquer***. The divide-and-conquer paradigm can be described in general terms as consisting of the following three steps (see Figure 8.1):

1. ***Divide:*** If the input size is smaller than a certain threshold (say, 10 elements), solve the problem directly using a straightforward method and return the solution so obtained. Otherwise, divide the input data into two or more disjoint subsets.
2. ***Recur:*** Recursively solve the subproblems associated with the subsets.
3. ***Conquer:*** Take the solutions to the subproblems and "merge" them into a solution to the original problem.

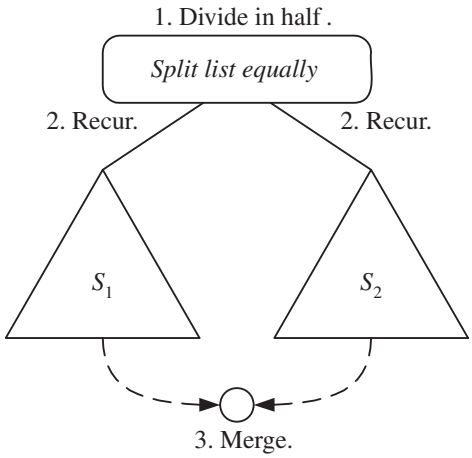


Figure 8.1: A visual schematic of the divide-and-conquer paradigm, applied to a problem involving a list that is divided equally in two in the divide step.

Merge-sort applies the divide-and-conquer technique to the sorting problem, where, for the sake of generality, let us consider the sorting problem to take a sequence, S, of objects as input, which could be represented with either a list or an array, and returns S in sorted order.

For the problem of sorting a sequence S with n elements, the three divide-and-conquer steps are as follows:

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

We can visualize an execution of the merge-sort algorithm using a binary tree T, called the ***merge-sort tree***. (See Figure 8.2.) Each node of the merge-sort tree, T, represents a recursive call of the merge-sort algorithm. We associate with each node v of T the sequence S that is processed by the call associated with v. The children of node v are associated with the recursive calls that process the subsequences S_1 and S_2 of S. The external nodes of T are associated with individual elements of S, corresponding to instances of the algorithm that make no recursive calls.

Figure 8.2 summarizes an execution of the merge-sort algorithm by showing the input and output sequences processed at each node of the merge-sort tree. This algorithm visualization in terms of the merge-sort tree helps us analyze the running time of the merge-sort algorithm. In particular, since the size of the input sequence roughly halves at each recursive call of merge-sort, the height of the merge-sort tree is about $\log n$ (recall that the base of $\log$ is 2 if omitted).

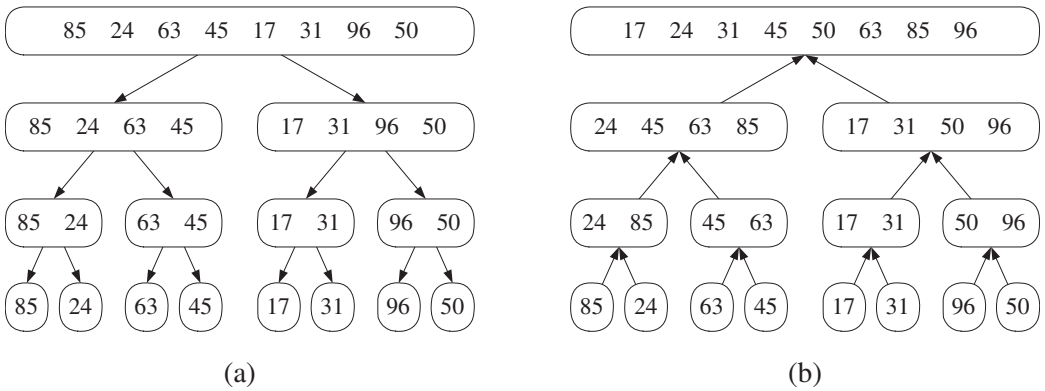


Figure 8.2: Merge-sort tree T for an execution of the merge-sort algorithm on a sequence with eight elements: (a) input sequences processed at each node of T; (b) output sequences generated at each node of T.

Having given an overview of merge-sort and an illustration of how it works, let us consider each of the steps of this divide-and-conquer algorithm in more detail. The divide and recur steps of the merge-sort algorithm are simple; dividing a sequence of size n involves separating it at the element with rank $\lceil n/2 \rceil$, and the recursive calls simply involve passing these smaller sequences as parameters. The difficult step is the conquer step, which merges two sorted sequences into a single sorted sequence. We provide a pseudocode description of the method for merging two sorted arrays in Algorithm 8.3. It merges two sorted arrays, S_1 and S_2, by iteratively removing a smallest element from one of these two and adding it to the end of an output array, S, until one of these two arrays is empty, at which point we copy the remainder of the other array to the output array.

Algorithm merge(S_1, S_2, S):
Input: Two arrays, S_1 and S_2, of size n_1 and n_2, respectively, sorted in non-decreasing order, and an empty array, S, of size at least $n_1 + n_2$
Output: S, containing the elements from S_1 and S_2 in sorted order

```
i ← 1
j ← 1
while i ≤ n and j ≤ n do
    if S1[i] ≤ S2[j] then
        S[i + j − 1] ← S1[i]
        i ← i + 1
    else
        S[i + j − 1] ← S2[j]
        j ← j + 1
while i ≤ n do
    S[i + j − 1] ← S1[i]
    i ← i + 1
while j ≤ n do
    S[i + j − 1] ← S2[j]
    j ← j + 1
```

Algorithm 8.3: Merging two sorted arrays, with indexing beginning at 1.

One of the nice properties of this merge algorithm is that the while loops involve simple scans of the input arrays, S_1 and S_2. For large data sets, this kind of data access is efficient, because of the sequential way that data is typically accessed in external storage devices, like disk drives. The trade-off for this benefit, however, is that we have to use an output array, S, rather than reusing the space in the input arrays themselves.

If we want to merge two sorted sequences given as linked lists, instead of arrays, then we would use a similar method to the array-based merge algorithm, which would involve our comparing the front elements in the two lists, removing

the smaller one from its list, and adding that element to the end of an output linked list. Once one of the lists is empty, we would then copy the remainder of the other list to the output list.

We show an example execution of a list-based merge algorithm in Figure 8.4.

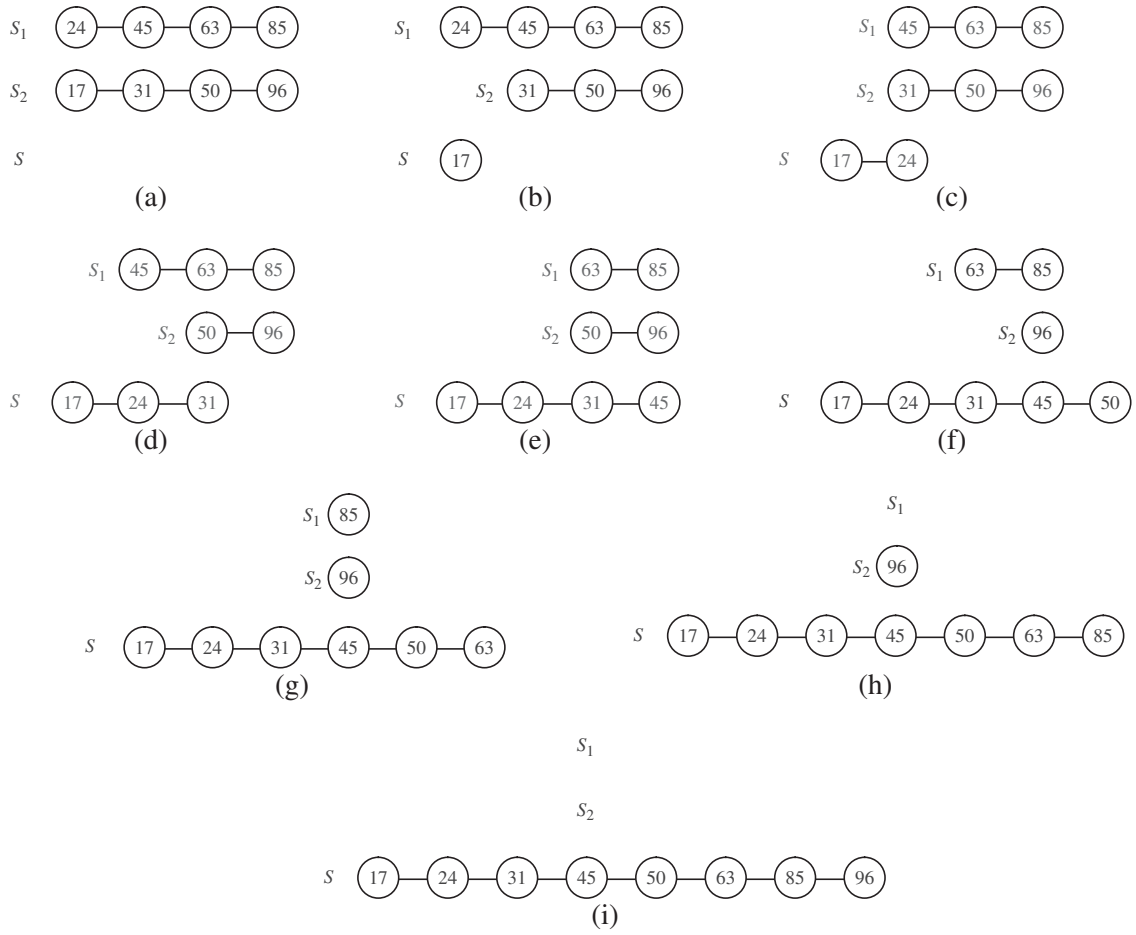


Figure 8.4: Example execution of a merge algorithm for sorted linked lists.

Analysis of the Merge-Sort Algorithm

Our analysis for the merge-sort algorithm begins with the merge algorithm. Let n_1 and n_2 be the number of elements of S_1 and S_2, respectively. Algorithm merge has three **while** loops. The operations performed inside each loop take $O(1)$ time each. The key observation is that during each iteration of any one of the loops, one element is added to the output array S and is never considered again. This observation implies that the overall number of iterations of the three loops is $n_1 +$

n_2. Thus, the running time of algorithm merge is $O(n_1 + n_2)$, as we summarize:

Theorem 8.1: *Merging two sorted arrays S_1 and S_2 takes $O(n_1+n_2)$ time, where n_1 is the size of S_1 and n_2 is the size of S_2.*

Having given the details of the merge algorithm, let us analyze the running time of the entire merge-sort algorithm, assuming it is given an input sequence of n elements. For simplicity, let us also assume n is a power of 2. We analyze the merge-sort algorithm by referring to the merge-sort tree, T.

First, we analyze the height of the merge-sort tree, T, referring to Figure 8.5.

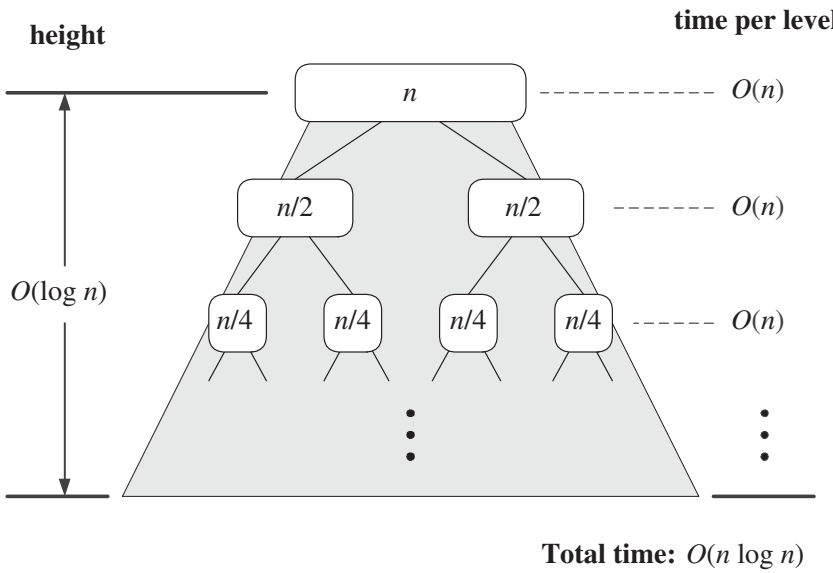


Figure 8.5: A visual analysis of the running time of merge-sort. Each node of the merge-sort tree is labeled with the size of its subproblem.

We observe that the length of the subsequence stored at a node of T with depth (distance from the root) i is $n/2^i$ since we halve the length of the sequence each time we go down one level. Thus, a node at depth $i = \log n$ stores a single-element subsequence and therefore is a leaf of the tree. We conclude that the height of the merge-sort tree is $\log n$.

We call the ***time spent at a node*** v of T the running time of the recursive call associated with v, excluding the time taken waiting for the recursive calls associated with the children of v to terminate. In other words, the time spent at node v includes the running times of the divide and conquer steps, but excludes the running time of the recur step. We have already observed that the details of the divide step are straightforward; this step runs in time proportional to the size of the sequence for v. Also, as shown in Theorem 8.1, the conquer step, which consists of merging two

sorted subsequences, also takes linear time. That is, letting i denote the depth of node v, the time spent at node v is $O(n/2^i)$, since the size of the sequence handled by the recursive call associated with v is equal to $n/2^i$.

Looking at the tree T more globally, as shown in Figure 8.5, we see that, given our definition of "time spent at a node," the running time of merge-sort is equal to the sum of the times spent at the nodes of T. Observe that T has exactly 2^i nodes at depth i. This simple observation has an important consequence, for it implies that the overall time spent at all the nodes of T at depth i is $O(2^i \cdot n/2^i)$, which is $O(n)$. We have previously seen that the height of T is $\log n$. Thus, since the time spent at each of the $\log n + 1$ levels of T is $O(n)$, we have the following result:

Theorem 8.2: *Merge-sort on sequence of n elements runs in $O(n \log n)$ time.*

The above analysis was done under the simplifying assumption that n is a power of 2. If this is not the case, the analysis becomes a bit more complicated.

Regarding the height of the merge-sort tree, we have:

Theorem 8.3: *The merge-sort tree associated with an execution of merge-sort on a sequence of size n has height $\lceil \log n \rceil$.*

The justification of Theorem 8.3 is left to a simple exercise (R-8.1).

Finally, we leave it to another exercise (R-8.3) how to extend the rest of the analysis of the running time of merge-sort to the general case when n is not a power of 2.

8.1.2 Merge-Sort and Recurrence Equations

There is another way to justify that the running time of the merge-sort algorithm is $O(n \log n)$. Let the function $t(n)$ denote the worst-case running time of merge-sort on an input sequence of size n. Since merge-sort is recursive, we can characterize function $t(n)$ by means of the following equalities, where function $t(n)$ is recursively expressed in terms of itself, as follows:

$$t(n) = \begin{cases} b & \text{if } n = 1 \text{ or } n = 0 \\ t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + cn & \text{otherwise} \end{cases}$$

where $b > 0$ and $c > 0$ are constants. A characterization of a function such as the one above is called a ***recurrence equation*** (Sections 1.1.4 and 11.1), since the function appears on both the left- and right-hand sides of the equal sign. Although such a characterization is correct and accurate, what we really desire is a big-Oh type of characterization of $t(n)$ that does not involve the function $t(n)$ itself (that is, we want a ***closed-form*** characterization of $t(n)$).

In order to provide a closed-form characterization of $t(n)$, let us restrict our attention to the case when n is a power of 2. We leave the problem of showing

that our asymptotic characterization still holds in the general case as an exercise (R-8.3). In this case, we can simplify the definition of $t(n)$ as follows:

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

But, even so, we must still try to characterize this recurrence equation in a closed-form way. One way to do this is to iteratively apply this equation, assuming n is relatively large. For example, after one more application of this equation, we can write a new recurrence for $t(n)$ as follows:

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

If we apply the equation again, we get

$$t(n) = 2^3 t\left(n/2^3\right) + 3cn.$$

Applying the equation once again, we obtain

$$t(n) = 2^4 t\left(n/2^4\right) + 4cn.$$

Now, a clear pattern emerges, and we infer that after applying this equation i times, we get

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

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

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

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

8.2 Quick-Sort

The quick-sort algorithm sorts a sequence S using a simple divide-and-conquer approach, whereby we divide S into subsequences, recur to sort each subsequence, and then combine the sorted subsequences by a simple concatenation. In particular, the quick-sort algorithm consists of the following three steps (see Figure 8.6):

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

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

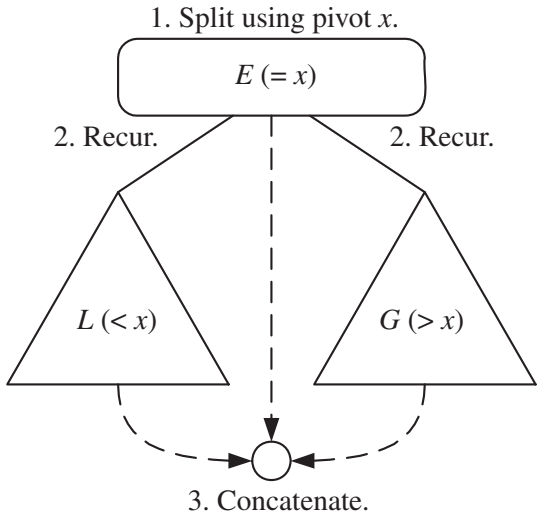


Figure 8.6: A visual schematic of the quick-sort algorithm.

Like merge-sort, we can visualize quick-sort using a binary recursion tree, called the ***quick-sort tree***. Figure 8.7 visualizes the quick-sort algorithm, showing example input and output sequences for each node of the quick-sort tree.

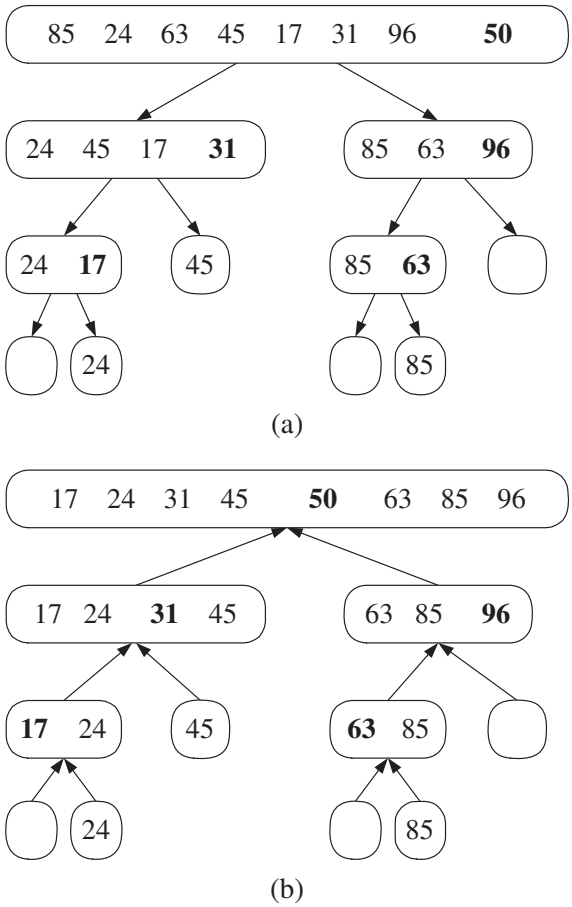


Figure 8.7: Quick-sort tree T for an execution of the quick-sort algorithm on a sequence with eight elements: (a) input sequences processed at each node of T; (b) output sequences generated at each node of T. The pivot used at each level of the recursion is shown in bold.

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

Running Time of Quick-Sort

We can analyze the running time of quick-sort with the same technique used for merge-sort in Section 8.1.1. Namely, we identify the time spent at each node of the quick-sort tree T (Figure 8.7) and we sum up the running times for all the nodes. The divide step and the conquer step of quick-sort are easy to implement in linear time. Thus, the time spent at a node v of T is proportional to the ***input size*** $s(v)$ of v, defined as the size of the sequence handled by the invocation of quick-sort associated with node v. Since subsequence E has at least one element (the pivot), the sum of the input sizes of the children of v is at most $s(v) - 1$.

Given a quick-sort tree T, let s_i denote the sum of the input sizes of the nodes at depth i in T. Clearly, $s_0 = n$, since the root r of T is associated with the entire sequence. Also, $s_1 \leq n - 1$, since the pivot is not propagated to the children of r. Consider next s_2. If both children of r have nonzero input size, then $s_2 = n - 3$. Otherwise (one child of the root has zero size, the other has size $n-1$), $s_2 = n-2$. Thus, $s_2 \leq n - 2$. Continuing this line of reasoning, we obtain that $s_i \leq n - i$.

As observed in Section 8.2, the height of T is $n - 1$ in the worst case. Thus, the worst-case running time of quick-sort is

$$O\left(\sum_{i=0}^{n-1} s_i\right), \text{ which is } O\left(\sum_{i=0}^{n-1}(n-i)\right) \text{ that is, } O\left(\sum_{i=1}^{n} i\right).$$

By Theorem 1.13, $\sum_{i=1}^{n} i$ is $O(n^2)$. Thus, quick-sort runs in $O(n^2)$ worst-case time. Given its name, we would expect quick-sort to run quickly. However, the above quadratic bound indicates that quick-sort is slow in the worst case. Paradoxically, this worst-case behavior occurs for problem instances when sorting should be easy—if the sequence is already sorted. Still, note that the best case for quick-sort on a sequence of distinct elements occurs when subsequences L and G happen to have roughly the same size. Indeed, in this case we save one pivot at each internal node and make two equal-sized calls for its children. Thus, we save 1 pivot at the root, 2 at level 1, 2^2 at level 2, and so on. That is, in the best case, we have

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

and so on. Thus, in the best case, T has height $O(\log n)$ and quick-sort runs in $O(n \log n)$ time. We leave the justification of this fact as an exercise (R-8.6).

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

8.2.1 Randomized Quick-Sort

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

Since the goal of the partition step of the quick-sort method is to divide the sequence S almost equally, let us use a new rule to pick the pivot—choose a ***random element*** of the input sequence. As we show next, the resulting algorithm, called ***randomized quick-sort***, has an expected running time of $O(n \log n)$ given a sequence with n elements.

Theorem 8.4: *The expected running time of randomized quick-sort on a sequence of size n is $O(n \log n)$.*

Proof: We make use of a simple fact from probability theory:

> *The expected number of times that a fair coin must be flipped until it shows "heads" k times is $2k$.*

Consider now a particular recursive invocation of randomized quick-sort, and let m denote the size of the input sequence for this invocation. Say that this invocation is "good" if the pivot chosen creates subsequences L and G that have size at least $m/4$ and at most $3m/4$ each. Since the pivot is chosen uniformly at random and there are $m/2$ pivots for which this invocation is good, the probability that an invocation is good is $1/2$ (the same as the probability a coin comes up heads).

If a node v of the quick-sort tree T, as shown in Figure 8.8, is associated with a "good" recursive call, then the input sizes of the children of v are each at most $3s(v)/4$ (which is the same as $s(v)/(4/3)$). If we take any path in T from the root to an external node, then the length of this path is at most the number of invocations that have to be made (at each node on this path) until achieving $\log_{4/3} n$ good invocations. Applying the probabilistic fact reviewed above, the expected number of invocations we must make until this occurs is $2 \log_{4/3} n$ (if a path terminates before this level, that is all the better). Thus, the expected length of any path from the root to an external node in T is $O(\log n)$. Recalling that the time spent at each level of T is $O(n)$, the expected running time of randomized quick-sort is $O(n \log n)$. ■

We note that the expectation in the running time is taken over all the possible choices the algorithm makes, and is independent of any assumptions about the distribution of input sequences the algorithm is likely to encounter. Actually, by using powerful facts from probability, we can show that the running time of randomized quick-sort is $O(n \log n)$ with high probability. (See Exercise C-8.4.)

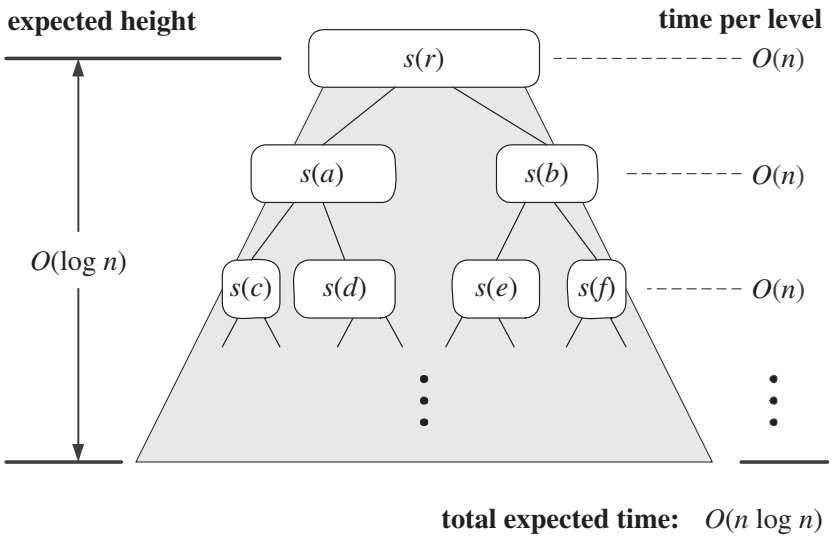


Figure 8.8: A visual time analysis of the quick-sort tree T.

8.2.2 In-Place Quick-Sort

Recall from Section 5.4 that a sorting algorithm is ***in-place*** if it uses only a small amount of memory in addition to that needed for the objects being sorted themselves. The merge-sort algorithm, as we have described it above, is not in-place, and making it be in-place seems quite difficult. In-place sorting is not inherently difficult, however. For, as with heap-sort, quick-sort can be adapted to be in-place.

Performing the quick-sort algorithm in-place requires a bit of ingenuity, however, for we must use an input array itself to store the subarrays for all the recursive calls. We show algorithm inPlaceQuickSort, which performs in-place quick-sort, in Algorithm 8.9. Algorithm inPlaceQuickSort assumes that the input array, S, has distinct elements. The reason for this restriction is explored in Exercise R-8.7. The extension to the general case is discussed in Exercise C-8.8.

In-place quick-sort modifies the input sequence using swapElements operations and does not explicitly create subsequences. Indeed, a subsequence of the input sequence is implicitly represented by a range of positions specified by a left-most rank l and a right-most rank r. The divide step is performed by scanning the sequence simultaneously from l forward and from r backward, swapping pairs of elements that are in reverse order, as shown in Figure 8.10. When these two indices "meet," subsequences L and G are on opposite sides of the meeting point. The algorithm completes by recursing on these two subsequences. In-place quick-sort reduces the running time, caused by the creation of new sequences and the movement of elements between them, by a constant factor.

Algorithm inPlacePartition(S, a, b):
Input: An array, S, of distinct elements; integers a and b such that $a \leq b$
Output: An integer, l, such that the subarray $S[a\,..\,b]$ is partitioned into $S[a..l-1]$ and $S[l..b]$ so that every element in $S[a..l-1]$ is less than each element in $S[l..b]$

```
Let r be a random integer in the range [a, b]
Swap S[r] and S[b]
p ← S[b]        // the pivot
l ← a           // l will scan rightward
r ← b − 1       // r will scan leftward
while l ≤ r do  // find an element larger than the pivot
    while l ≤ r and S[l] ≤ p do
        l ← l + 1
    while r ≥ l and S[r] ≥ p do   // find an element smaller than the pivot
        r ← r − 1
    if l < r then
        Swap S[l] and S[r]
Swap S[l] and S[b]        // put the pivot into its final place
return l
```

Algorithm inPlaceQuickSort(S, a, b):
Input: An array, S, of distinct elements; integers a and b
Output: The subarray $S[a\,..\,b]$ arranged in nondecreasing order

```
if a ≥ b then return        // subrange with 0 or 1 elements
l ← inPlacePartition(S, a, b)
inPlaceQuickSort(S, a, l − 1)
inPlaceQuickSort(S, l + 1, b)
```

Algorithm 8.9: In-place randomized quick-sort for an array, S.

Dealing with the Recursion Stack

Actually, the above description of quick-sort is not quite in-place, as it could, in the worst case, require a linear amount of additional space besides the input array. Of course, we are using no additional space for the subsequences, and we are using only a constant amount of additional space for local variables (such as l and r).

So, where does this additional space come from?

It comes from the recursion, since we need space for a stack proportional to the depth of the recursion tree in order to keep track of the recursive calls for quick-sort. This stack can become as deep as $\Theta(n)$, in fact, if we have a series of bad pivots, since we need to have a method frame for every active call when we make the call for the deepest node in the quick-sort tree.

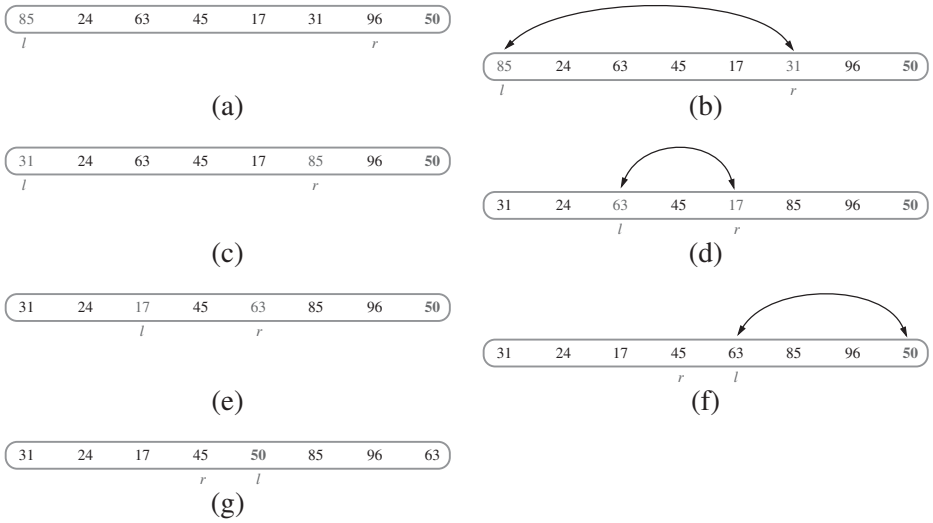


Figure 8.10: An example execution of the inPlacePartition algorithm.

Fortunately, we can fix our quick-sort algorithm to actually be in-place, and use only $O(\log n)$ additional space, by changing the way we do the recursive calls. The key detail for such an implementation is that if we always do the recursive call for the smaller subproblem first, then we can replace the second recursive call with a loop, since it comes last. This ability to change a recursive call into an iteration if it is the last operation in a recursive procedure is known as ***tail recursion***. The details are shown in Algorithm 8.11.

The depth of recursion and, hence, the amount of additional space used for the method stack in Algorithm 8.11 is $O(\log n)$. To see this, note that by doing a recursive call only for the smaller subproblem each time, we guarantee that the size of any recursive subproblem is at most half the size of the subproblem that is making that call. Thus, the depth of the recursion stack is never more than $O(\log n)$.

Algorithm CorrectInPlaceQuickSort(S, a, b):
Input: An array, S, of distinct elements; integers a and b
Output: The subarray $S[a \,..\, b]$ arranged in nondecreasing order

```
while a < b do
    l ← inPlacePartition(S, a, b)        // from Algorithm 8.9
    if l − a < b − l then   // first subarray is smaller
        CorrectInPlaceQuickSort(S, a, l − 1)
        a ← l + 1
    else
        CorrectInPlaceQuickSort(S, l + 1, b)
        b ← l − 1
```

Algorithm 8.11: Correct version of in-place randomized quick-sort for an array, S.

8.3 A Lower Bound on Comparison-Based Sorting

Recapping our discussions on sorting to this point, we have described several methods with either a worst-case or expected running time of $O(n \log n)$ on an input sequence of size n. These methods include merge-sort and quick-sort, described in this chapter, as well as heap-sort, described in Section 5.4. A natural question to ask, then, is whether it is possible to sort any faster than in $O(n \log n)$ time.

In this section, we show that if the computational primitive used by a sorting algorithm is the comparison of two elements, then this is the best we can do—comparison-based sorting has an $\Omega(n \log n)$ worst-case lower bound on its running time. (Recall the notation $\Omega(\cdot)$ from Section 1.1.5.) To focus on the main cost of comparison-based sorting, let us only count the comparisons that a sorting algorithm performs. Since we want to derive a lower bound, this will be sufficient.

Suppose we are given a sequence $S = (x_1, x_2, \ldots, x_n)$ that we wish to sort, and let us assume that all the elements of S are distinct (this is not a restriction since we are deriving a lower bound). Each time a sorting algorithm compares two elements x_i and x_j (that is, it asks, "is $x_i < x_j$?"), there are two outcomes: "yes" or "no." Based on the result of this comparison, the sorting algorithm may perform some internal calculations (which we are not counting here) and will eventually perform another comparison between two other elements of S, which again will have two outcomes. Therefore, we can represent a comparison-based sorting algorithm with a decision tree T. That is, each internal node v in T corresponds to a comparison and the edges from node v' to its children correspond to the computations resulting from either a "yes" or "no" answer (see Figure 8.12).

It is important to note that the hypothetical sorting algorithm in question probably has no explicit knowledge of the tree T. We simply use T to represent all the possible sequences of comparisons that a sorting algorithm might make, starting from the first comparison (associated with the root) and ending with the last comparison (associated with the parent of an external node) just before the algorithm terminates its execution.

Each possible initial ordering, or ***permutation***, of the elements in S will cause our hypothetical sorting algorithm to execute a series of comparisons, traversing a path in T from the root to some external node. Let us associate with each external node v in T, then, the set of permutations of S that cause our sorting algorithm to end up in v. The most important observation in our lower-bound argument is that each external node v in T can represent the sequence of comparisons for at most one permutation of S. The justification for this claim is simple: if two different permutations P_1 and P_2 of S are associated with the same external node, then there are at least two objects x_i and x_j, such that x_i is before x_j in P_1 but x_i is after x_j in P_2. At the same time, the output associated with v must be a specific reordering of S, with either x_i or x_j appearing before the other. But if P_1 and P_2 both cause the sorting algorithm to output the elements of S in this order, then that implies

there is a way to trick the algorithm into outputting x_i and x_j in the wrong order. Since this cannot be allowed by a correct sorting algorithm, each external node of T must be associated with exactly one permutation of S. We use this property of the decision tree associated with a sorting algorithm to prove the following result:

Theorem 8.5: *The running time of any comparison-based algorithm for sorting an n-element sequence is $\Omega(n \log n)$ in the worst case.*

Proof: The running time of a comparison-based sorting algorithm must be greater than or equal to the height of the decision tree T associated with this algorithm, as described above. (See Figure 8.12.) By the above argument, each external node in T must be associated with one permutation of S. Moreover, each permutation of S must result in a different external node of T. The number of permutations of n objects is

$$n! = n(n-1)(n-2)\cdots 2 \cdot 1.$$

Thus, T must have at least $n!$ external nodes. By Theorem 2.7, the height of T is at least $\log(n!)$. This immediately justifies the theorem, because there are at least $n/2$ terms that are greater than or equal to $n/2$ in the product $n!$; hence

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

which is $\Omega(n \log n)$. ■

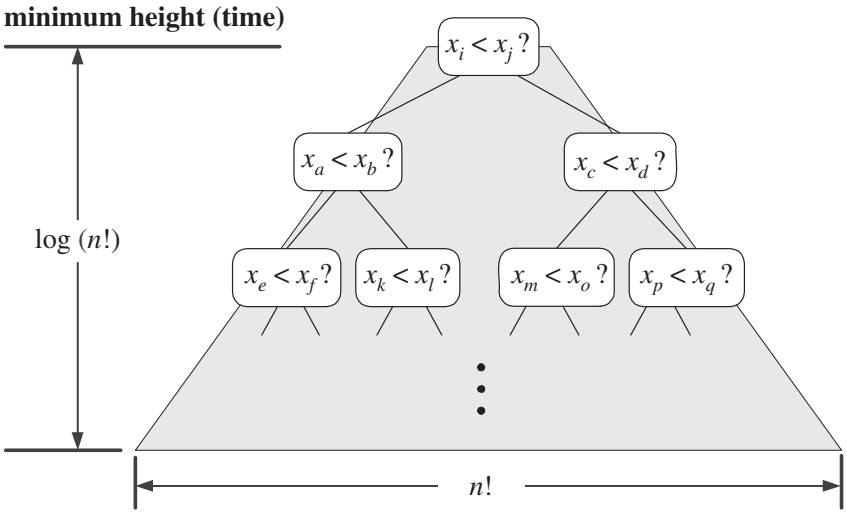


Figure 8.12: Visualizing the lower bound for comparison-based sorting.

8.4 Exercises

Reinforcement

R-8.1 Give a complete justification of Theorem 8.3.

R-8.2 Give a pseudocode description of the merge-sort algorithm assuming the input is given as a linked list.

R-8.3 Show that the running time of the merge-sort algorithm on an n-element sequence is $O(n \log n)$, even when n is not a power of 2.

R-8.4 Suppose we modify the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an n-element sequence as the pivot, we choose the element at index $\lfloor n/2 \rfloor$, that is, an element in the middle of the sequence. What is the running time of this version of quick-sort on a sequence that is already sorted?

R-8.5 Consider again the modification of the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an n-element sequence as the pivot, we choose the element at index $\lfloor n/2 \rfloor$. Describe the kind of sequence that would cause this version of quick-sort to run in $\Theta(n^2)$ time.

R-8.6 Show that the best-case running time of quick-sort on a sequence of size n with distinct elements is $O(n \log n)$.

R-8.7 Suppose that algorithm inPlaceQuickSort (Algorithm 8.9) is executed on a sequence with duplicate elements. Show that, in this case, the algorithm correctly sorts the input sequence, but the result of the divide step may differ from the high-level description given in Section 8.2 and may result in inefficiencies. In particular, what happens in the partition step when there are elements equal to the pivot? What is the running time of the algorithm if all the elements of the input sequence are equal?

Creativity

C-8.1 Describe a variation of the merge-sort algorithm that is given a single array, S, as input, and uses only an additional array, T, as a workspace. No other memory should be used other than a constant number of variables.

C-8.2 Let A be a collection of objects. Describe an efficient method for converting A into a set. That is, remove all duplicates from A. What is the running time of this method?

C-8.3 Suppose we are given two n-element sorted sequences A and B that should not be viewed as sets (that is, A and B may contain duplicate entries). Describe an $O(n)$-time method for computing a sequence representing the set $A \cup B$ (with no duplicates).

C-8.4 Show that randomized quick-sort runs in $O(n \log n)$ time with probability $1 - 1/n^2$.

Hint: Use the ***Chernoff bound*** that states that if we flip a coin k times, then the probability that we get fewer than $k/16$ heads is less than $2^{-k/8}$.

C-8.5 Suppose we are given a sequence S of n elements, each of which is colored red or blue. Assuming S is represented as an array, give an in-place method for ordering S so that all the blue elements are listed before all the red elements. Can you extend your approach to three colors?

C-8.6 Suppose we are given two sequences A and B of n elements, possibly containing duplicates, on which a total order relation is defined. Describe an efficient algorithm for determining if A and B contain the same set of elements (possibly in different orders). What is the running time of this method?

C-8.7 Suppose we are given a sequence S of n elements, on which a total order relation is defined. Describe an efficient method for determining whether there are two equal elements in S. What is the running time of your method?

C-8.8 Modify Algorithm inPlaceQuickSort (Algorithm 8.9) to handle the general case efficiently when the input array, S, may have duplicate keys.

C-8.9 Let S be an array of n elements on which a total order relation is defined. An ***inversion*** in S is a pair of indices i and j such that $i < j$ but $S[i] > S[j]$. Describe an algorithm running in $O(n \log n)$ time for determining the ***number*** of inversions in S (which can be as large as $O(n^2)$).

Hint: Try to modify the merge-sort algorithm to solve this problem.

C-8.10 Give an example of a sequence of n integers with $\Omega(n^2)$ inversions. (Recall the definition of inversion from Exercise C-8.9.)

C-8.11 Let A and B be two sequences of n integers each. Given an integer x, describe an $O(n \log n)$-time algorithm for determining if there is an integer a in A and an integer b in B such that $x = a + b$.

C-8.12 Given a sequence of numbers, $(x_1, x_2, \ldots, x_n)$, the ***mode*** is the value that appears the most number of times in this sequence. Give an efficient algorithm to compute the mode for a sequence of n numbers. What is the running time of your method?

C-8.13 Suppose you would like to sort n music files, but you only have an old, unreliable computer, which you have nicknamed "Rustbucket." Every time Rustbucket compares two music files, x and y, there is an independent 50-50 chance that it has an internal disk fault and returns the value 0, instead of the correct result, 1, for "**true**" or -1, for "**false**," to the question, "$x \leq y$?" That is, for each comparison of music files that Rustbucket is asked to perform, it is as if it flips a fair coin and answers the comparison correctly if the coin turns up "heads" and answers with 0 if the coin turns up "tails." Moreover, this behavior occurs independent of previous comparison requests, even for the same pair of music files. Otherwise, Rustbucket correctly performs every other kind of operation (not involving the comparison of two music files), including if-statements, for-loops, and while-loops based on comparisons of integers. Describe an efficient algorithm that can use Rustbucket to sort n music files correctly and show that your algorithm has an expected running time that is $O(n \log n)$.

Applications

A-8.1 Suppose you are given a new hardware device that can merge $k > 2$ different sorted lists of total size n into a single sorted list in $O(n)$ time, independent of the value of k. Such a device could, for example, be based on a hardware streaming system or could be based on a network protocol. Show that you can use this device to sort n elements in $O(n \log n / \log k)$ time. That is, if k is $\Theta(\sqrt{n})$, then you can use this device to sort in linear time.

A-8.2 Suppose we are given an n-element sequence S such that each element in S represents a different vote in an election, where each vote is given as an integer representing the ID of the chosen candidate. Without making any assumptions about who is running or even how many candidates there are, design an $O(n \log n)$-time algorithm to see who wins the election S represents, assuming the candidate with the most votes wins.

A-8.3 Consider the voting problem from the previous exercise, but now suppose that we know the number $k < n$ of candidates running. Describe an $O(n \log k)$-time algorithm for determining who wins the election.

A-8.4 Bob has a set, A, of n nuts and a set, B, of n bolts, such that each nut has a unique matching bolt. Unfortunately, the nuts in A all look the same, and the bolts in B all look the same as well. The only comparison that Bob can make is to take a nut-bolt pair (a, b), such that $a \in A$ and $b \in B$, and test if the threads of a are larger, smaller, or a perfect match with the threads of b. Describe an efficient algorithm for Bob to match up all of his nuts and bolts. What is the running time of this algorithm?

A-8.5 As mentioned above, for each word, w, in a collection of documents, an inverted file stores a list of documents that contain the word, w. In addition, search engines typically order the list for each word by a ranking score. Modern search engines must be able to answer more than just single-word queries, however. Describe an efficient method for computing a list of the documents that contain two words, w and u, ordered by the ranking scores assigned to the documents. What is the running time of your method in terms of n_w and n_u, the respective sizes of the lists for w and u?

A-8.6 In cloud computing, it is common for a client, "Alice," to store her data on an external server owned by a cloud storage provider, "Bob." Because Bob is likely to be honest, but curious, Alice wants to keep the contents of her data private from Bob. Of course, she can encrypt her data using a secret key that only she knows, but that is not enough, since she may reveal information about her data simply based on the pattern in which she accesses her data. Since Bob can see the access pattern for Alice's data, even if she encrypts it, Alice should consider using an algorithm that has a data access pattern that does not depend on any of its input values. Such an algorithm is said to be ***data-oblivious***. Suppose, then, that Alice wants to sort an array, A, of elements stored on Bob's computer, but do so in a data-oblivious fashion, so that she sorts privately, even though each time she accesses some item, $A[i]$, in the array, A, Bob learns that she is accessing the item at index i. She can use a constant amount of local private memory, e.g., to store indices, pointers, or to perform a comparison and swap, if the elements

Algorithm OddEvenMerge(A, B, C):
Input: Two sorted arrays, $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_n]$, and an empty array, C, of size $2n$
Output: C, containing the elements from A and B in sorted order
Let $O_1 \leftarrow [a_1, a_3, a_5, \ldots, a_{n-1}]$
Let $O_2 \leftarrow [b_1, b_3, b_5, \ldots, b_{n-1}]$
Let $E_1 \leftarrow [a_2, a_4, a_6, \ldots, a_n]$
Let $E_2 \leftarrow [b_2, b_4, b_6, \ldots, b_n]$
Call OddEvenMerge(O_1, O_2, O), where $O = [o_1, o_2, \ldots, o_n]$
Call OddEvenMerge(E_1, E_2, E), where $E = [e_1, e_2, \ldots, e_n]$
Let $C \leftarrow [o_1, e_1, o_2, e_2, \ldots, o_n, e_n]$
for $i \leftarrow 1$ **to** n **do**
    Do a compare-exchange of $C[2i-1]$ and $C[2i]$
return C

Algorithm 8.13: Odd-even merge.

are out of order, as an atomic action called a ***compare-exchange***. For example, she could use bubble-sort (Algorithm 5.20) to sort A, since this algorithms is data-oblivious when implemented using the compare-exchange primitive. But this would require $O(n^2)$ time, which is quite inefficient for solving the sorting problem. An alternative is to use the ***odd-even merge-sort*** algorithm, which is the same as the merge-sort algorithm given above, except that the merge step is replaced with the merge method shown in Algorithm 8.13. Argue why the odd-even merge-sort algorithm is data-oblivious, and analyze the running time of the resulting sorting algorithm.

A-8.7 In computer games and also in simulations of card-playing scenarios, we sometimes need to use a computer to simulate the way that person would shuffle a deck of cards. Given two decks of n cards each, the ***riffle shuffle*** algorithm involves repeatedly choosing one of the two decks at random and then removing the bottom card from that deck and placing that card on the top of an output deck. This card-choosing step is repeated until all the original cards are placed in the output deck. Define a ***recursive-riffle*** algorithm, which cuts a deck of n cards into two decks of $n/2$ each, where n is a power of 2, and then calls the recursive-riffle algorithm on each half. When these recursive calls return, the recursive-riffle algorithm then performs a riffle shuffle of the two decks to produce the shuffled result. Show that every card in the original deck has an equal probability of being the top card after a recursive-riffle is performed on the deck, and analyze the running time of the recursive-riffle algorithm using a recurrence equation.

A-8.8 Many states require that candidate names appear on a ballot in random order, so as to minimize biases that can arise from the order in which candidate names appear on a ballot for a given election. For instance, in the 2012 general election, the Secretary of State of California performed a random drawing that determined that candidate names for that election must appear in alphabetical order based on the following ordering of letters:

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

For example, if three candidates in that election had the last names, "BROWN," "BLACK," and "WHITE," then they would appear on the ballot in the order, (WHITE, BROWN, BLACK). Describe an efficient algorithm that takes, as input, an array, A, specifying an ordering of letters such as this, and a collection of names, and sorts the collection of names using a lexicographic order based on the alternative ordering of letters given in A. What is the running time of your method in terms of m, the size of the array, A, and n, the number of names to be sorted? (You may assume the length of each name is bounded by some constant, but you may not assume that m or n is a constant.)

A-8.9 A ***floating-point number*** is a pair, (m, d), of integers, which represents the number $m \times b^d$, where b is either 2 or 10. In any real-world programming environment, the sizes of m and d are limited; hence, each arithmetic operation involving two floating-point numbers may have some ***roundoff error***. The traditional way to account for this error is to introduce a ***machine precision*** parameter, $\epsilon < 1$, for the given programming environment, and bound roundoff errors in terms of this parameter. For instance, in the case of floating-point addition, $fl(x + y)$, for summing two floating-point numbers, x and y, we can write

$$fl(x + y) = (x + y) \cdot (1 + \delta_{x,y}),$$

where $|\delta_{x,y}| \leq \epsilon$. Consider, then, using an accumulation algorithm for summing a sequence, $(x_1, x_2, \ldots, x_n)$, of positive floating-point numbers, as shown in Algorithm 8.14. Assuming that ϵ is small enough so that ϵ^2 times any floating-point number is negligibly small, then we can use a term, e_n, to estimate an upper bound for the roundoff error for summing these numbers in this way as

$$e_n = \epsilon \sum_{i=1}^{n} (n - i + 1)x_i.$$

Prove that the optimal order for summing any set of n positive floating-point number according to the standard accumulation algorithm, so as to minimize the error term, e_n, is to sum them in nondecreasing order. Also, give an $O(n \log n)$-time method for arranging the numbers in the sequence $(x_1, x_2, \ldots, x_n)$ so that the standard accumulation summation algorithm minimizes the error term, e_n.

Algorithm FloatSum$(x_1, x_2, \ldots, x_n)$:
Input: A sequence, $(x_1, x_2 \ldots, x_n)$, of n positive floating-point numbers
Output: A floating-point representation of the sum of the n numbers
$s \leftarrow 0.0$
for $i \leftarrow 1$ **to** n **do**
$s \leftarrow fl(s + x_i)$
return s

Algorithm 8.14: An accumulation algorithm for summing n floating-point numbers.

Chapter Notes

Knuth's classic text on *Sorting and Searching* [131] contains an extensive history of the sorting problem and algorithms for solving it, starting with the census card sorting machines of the late 19th century. Huang and Langston [107] describe how to merge two sorted lists in-place in linear time. The standard quick-sort algorithm is due to Hoare [100]. A tighter analysis of randomized quick-sort can be found in the book by Motwani and Raghavan [162]. Gonnet and Baeza-Yates [85] provide experimental comparisons and theoretical analyses of a number of different sorting algorithms. Kao and Wang [115] study the problem of minimizing the numerical error in summing n floating-point numbers.