Part III

Algorithm Design Techniques
Chapter 10

Divide and Conquer

In Part I of this text, we introduced several techniques for applying the top-down approach to algorithm design. We will now take a closer look at some of these techniques. In this chapter, we will look at the divide-and-conquer technique.

As we stated in Chapter 3, the divide-and-conquer technique involves reducing a large instance of a problem to one or more instances having a fixed fraction of the size of the original instance. For example, recall that the algorithm MaxSumDC, shown in Figure 3.3 on page 77, reduces large instances of the maximum subsequence sum problem to two smaller instances of roughly half the size.

Though we can sometimes convert divide-and-conquer algorithms to iterative algorithms, it is usually better to implement them using recursion. One reason is that typical divide-and-conquer algorithms implemented using recursion require very little stack space to support the recursion. If we divide an instance of size $n$ into instances of size $n/b$ whenever $n$ is divisible by $b$, we can express the total stack usage due to recursion with the recurrence

$$f(n) \in f(n/b) + \Theta(1).$$

Applying Theorem 3.32 to this recurrence, we see that $f(n) \in \Theta(lg n)$. The other reason for retaining the recursion is that when a large instance is reduced to more than one smaller instance, removing the recursion can be difficult and usually requires the use of a stack to simulate at least one recursive call.

Because divide-and-conquer algorithms are typically expressed using recursion, the analysis of their running times usually involves the asymptotic solution of a recurrence. Theorem 3.32 almost always applies to this recurrence. Not only does this give us a tool for analyzing running times, it also
can give us some insight into what must be done to make an algorithm more efficient. We will explain this concept further as we illustrate the technique by applying it to several problems.

## 10.1 Polynomial Multiplication

Suppose we are given two polynomials:

\[ p(x) = \sum_{i=0}^{n-1} a_i x^i \]
\[ q(x) = \sum_{i=0}^{n-1} b_i x^i. \]

We wish to compute the product polynomial:

\[ pq(x) = \sum_{i=0}^{2n-2} \left( \sum_{j=0}^{i} a_j b_{i-j} \right) x^i, \]

where we define \( a_j = b_j = 0 \) for \( n \leq j < 2n \). We can clearly compute each of the \( 2n - 1 \) coefficients in \( pq \) in \( \Theta(n) \) time; hence, by Theorem 3.28, we can compute the entire product in \( \Theta(n^2) \) time.

We wish to apply the divide-and-conquer technique to obtain a more efficient solution. Observe that if \( n > 1 \), then we can divide each polynomial into two smaller polynomials:

\[ p(x) = p_0(x) + x^m p_1(x) \]
\[ q(x) = q_0(x) + x^m q_1(x), \]

where

\[ p_0(x) = \sum_{i=0}^{m-1} a_i x^i \]
\[ p_1(x) = \sum_{i=0}^{n-m-1} a_{m+i} x^i \]
\[ q_0(x) = \sum_{i=0}^{m-1} b_i x^i \]
\[ q_1(x) = \sum_{i=0}^{n-m-1} b_{m+i} x^i. \]
If we set \( m = \lfloor n/2 \rfloor \), then each of the smaller polynomials has roughly \( n/2 \) terms.

The product polynomial is now

\[
pq(x) = p_0(x)q_0(x) + x^m(p_0(x)q_1(x) + p_1(x)q_0(x)) + x^{2m}p_1(x)q_1(x). \tag{10.1}
\]

To obtain the coefficients of \( pq \), we can first compute the four products of the smaller polynomials. We can then obtain any given coefficient of \( pq \) by performing at most two additions. We can therefore obtain all \( 2n - 1 \) coefficients in \( \Theta(n) \) time after the four smaller products are computed.

Setting \( m = \lfloor n/2 \rfloor \), we can describe the running time of this divide-and-conquer algorithm with the following recurrence:

\[
f(n) \in 4f(n/2) + \Theta(n) \tag{10.2}
\]

when \( n > 1 \) is a power of 2. Unfortunately, applying Theorem 3.32 yields \( f(n) \in \Theta(n^2) \), which is the same running time as the brute-force calculation.

This exercise illustrates an important point about the divide-and-conquer technique, namely, that the technique by itself does not guarantee improved running times. In order for the technique to be effective, it must save some work. Sometimes, as with MaxSumDC, the savings in work comes about naturally. In such cases, it may be rather hard to see how work was saved. In other cases, we must be more clever in order to save work.

As we suggested earlier, Theorem 3.32 can give insight into how a divide-and-conquer solution might be designed or improved. For example, consider recurrence (10.2) above. Because the third case of Theorem 3.32 applies, we cannot obtain a more efficient solution by reducing the \( \Theta(n) \) overhead outside of the recursive calls. In order to improve the performance, we need either to reduce the number of recursive calls or decrease the size of the smaller instances. We will focus on reducing the number of recursive calls. In the exercises, we explore alternative solutions involving decreasing the size of the smaller instances.

The following observation gives us the insight we need in order to reduce the number of recursive calls:

\[
(p_0(x) + p_1(x))(q_0(x) + q_1(x)) = 
p_0(x)q_0(x) + p_0(x)q_1(x) + p_1(x)q_0(x) + p_1(x)q_1(x).
\]

Note that all four of the terms in the right-hand-side above appear in the product \( pq \) (see (10.1) above). In order to make this fact useful, however, we need to be able to separate out the first and last terms. We can do this by
Figure 10.1 Divide-and-conquer polynomial multiplication algorithm

**Precondition:**  $p$ and $q$ are arrays of Numbers and $n$ is a positive Nat.

**Postcondition:**  Returns an array $P[0..2n-2]$ such that $P[i]$ is the coefficient of $x^i$ in the product $pq$, where $p[i]$ denotes the coefficient in $p$ of $x^i$ and $q[i]$ denotes the coefficient in $q$ of $x^i$.

\[
\text{POLYMULT}(p[0..n-1], q[0..n-1])
\]
\[
P \leftarrow \text{new ARRAY}[0..2(n-1)]
\]
\[
\text{if } n = 1
\]
\[
P[0] = p[0]q[0]
\]
\[
\text{else}
\]
\[
m \leftarrow \lceil n/2 \rceil; s \leftarrow \text{new ARRAY}[0..m-1]; t \leftarrow \text{new ARRAY}[0..m-1]
\]
\[
\text{COPY}(p[0..m-1], s[0..m-1]); \text{COPY}(q[0..m-1], t[0..m-1])
\]
\[
\text{for } i \leftarrow m \text{ to } n-1
\]
\[
s[i-m] \leftarrow s[i-m] + p[i]; t[i-m] \leftarrow t[i-m] + q[i]
\]
\[
P_1 \leftarrow \text{POLYMULT}(p[0..m-1], q[0..m-1])
\]
\[
P_2 \leftarrow \text{POLYMULT}(s[0..m-1], t[0..m-1])
\]
\[
P_3 \leftarrow \text{POLYMULT}(p[m..n-1], q[m..n-1])
\]
\[
\text{COPY}(P_1[0..2(m-1)], P[0..2(m-1)]); P[2m-1] \leftarrow 0
\]
\[
\text{COPY}(P_3[0..2(n-m-1)], P[2m..2(n-1)])
\]
\[
\text{for } i \leftarrow 0 \text{ to } 2(m-1)
\]
\[
P[m+i] \leftarrow P[m+i] + P_2[i] - P_1[i]
\]
\[
\text{for } i \leftarrow 0 \text{ to } 2(n-m-1)
\]
\[
P[m+i] \leftarrow P[m+i] - P_3[i]
\]
\[
\text{return } P
\]

correction 5/7/12.

computing the products $p_0(x)q_0(x)$ and $p_1(x)q_1(x)$, then subtracting. Thus, we can compute the product $pq$ using the following three products:

\[
P_1(x) = p_0(x)q_0(x)
\]
\[
P_2(x) = (p_0(x) + p_1(x))(q_0(x) + q_1(x))
\]
\[
P_3(x) = p_1(x)q_1(x)
\]

We can then compute any given coefficient of $pq$ with at most two subtractions and one addition.

The algorithm is shown in Figure 10.1. This implementation uses the \textsc{COPY} function specified in Figure 1.18 on page 22. Note that when we
CHAPTER 10. DIVIDE AND CONQUER

divide the polynomials, the low-order parts will be of degree \( m - 1 \), and the high-order parts will be of degree \( n - m - 1 \). We cannot select \( m \) so that these two degrees are the same if \( n \) is odd. Therefore, we need to be careful to note the degrees of each polynomial we construct. By choosing \( m = \lceil n/2 \rceil \), we ensure that \( m \geq n - m \). Thus, we can add the two halves of a polynomial by first recording the low-order half, then adding in the high-order half, yielding a polynomial of degree \( m - 1 \). After the recursive multiplications, \( P_1 \) and \( P_2 \) will both have degree \( 2(m - 1) \), but \( P_3 \) will have degree \( 2(n - m - 1) \). To construct \( P \), we can first copy \( P_1 \) and \( P_3 \) to the proper locations, and fill in 0 for the coefficient of \( x^{2m-1} \). We can then add \( P_2[i] - P_1[i] - P_3[i] \) to the coefficient of \( x^{m+i} \); however, because \( P_3 \) has a different degree than \( P_1 \) and \( P_2 \), we use a separate loop to subtract this polynomial.

From Figure 10.1 and Exercise 3.23 (page 102), it is evident that a total of \( \Theta(n) \) time is needed apart from the recursive calls. Thus, we can describe the running time with the following recurrence:

\[
f(n) \in 3f(n/2) + \Theta(n)
\]

whenever \( n > 1 \) is a power of 2. From Theorem 3.32, \( f(n) \in \Theta(n^{\lg 3}) \). Because \( \lg 3 \approx 1.59 \), this algorithm is an improvement over the brute-force calculation.

10.2 Merge Sort

In Section 5.6, we saw that we can sort using a worst-case running time in \( \Theta(n \lg n) \) using heap sort (Figure 5.20). However, as was suggested by Exercise 5.24, heap sort is not stable when sorting KEYED items; i.e., items with equal keys can be reordered by heap sort. In this section, we will apply divide-and-conquer to obtain a stable sorting algorithm that also runs in \( \Theta(n \lg n) \) time in the worst case.

We can apply divide-and-conquer to sorting by first dividing in half any array with more than one element and sorting the two halves. We then need to combine the two sorted halves into a single sorted array. We have therefore reduced the sorting problem to the problem of merging two sorted arrays into a single sorted array. Furthermore, in order to ensure stability, we require the following behavior for a merging algorithm:

- If two elements in the same input array have equal keys, they remain in the same order in the output array.
• If an element $x$ from the first input array has a key equal to some element $y$ in the second input array, then $x$ must precede $y$ in the output array.

Suppose we are given two sorted arrays. If either is empty, we can simply use the other. Otherwise, the element with minimum key in the two arrays needs to be first in the sorted result. The element with minimum key in each array is the first element in the array. We can therefore determine the overall minimum by comparing the keys of the first elements of the two arrays. If the keys are equal, in order to ensure stability, we must take the element from the first array. To obtain the remainder of the result, we merge the remainder of the two input arrays. We have therefore transformed a large instance of merging to a smaller instance.

Putting it all together, we have the MergeSort algorithm shown in Figure 10.2. Note that in the Merge function, when the loop terminates, either $i > m$ or $j > n$. Hence, either $A[i..m]$ or $B[j..n]$ is empty. As a result, only one of the two calls to Copy (see Figure 1.18 on page 22 for its specification) will have any effect.

It is easily seen that Merge runs in $\Theta(m + n)$ time. Therefore, the time required for MergeSort excluding the recursive calls is in $\Theta(n)$. For $n > 1$ a power of 2, the following recurrence gives the worst-case running time of MergeSort:

$$f(n) \in 2f(n/2) + \Theta(n).$$

From Theorem 3.32, $f(n) \in \Theta(n \lg n)$.

### 10.3 Quick Sort

Though merge sort and heap sort both run in $\Theta(n \lg n)$ time in the worst case, another divide-and-conquer algorithm is more commonly used when stability is not required. As was suggested in Exercise 2.17, sorting can be reduced to the Dutch national flag problem, which was introduced in Section 2.4. We first select from the array to be sorted a pivot element $p$, which we use to determine the colors of the elements, as follows:

- If $x.\text{Key}() < p.\text{Key}()$, then $x$ is red.
- If $x.\text{Key}() = p.\text{Key}()$, then $x$ is white.
- If $x.\text{Key}() > p.\text{Key}()$, then $x$ is blue.
Figure 10.2 The MergerSort algorithm

**Precondition:** $A[1..n]$ is an array of Keyed items.

**Postcondition:** $A[1..n]$ is a permutation of its original values sorted in nondecreasing order. If initially, $x = A[i]$ and $y = A[j]$, where $x.Key() = y.Key()$ and $i < j$, then $x$ precedes $y$ in the result.

`MergerSort(A[1..n])`

if $n > 1$

$m \leftarrow \lfloor n/2 \rfloor$; `MergerSort(A[1..m])`; `MergerSort(A[m+1..n])`

$B[1..n] \leftarrow \text{Merge}(A[1..m], A[m+1..n])$

`Copy(B[1..n], A[1..n])`

**Precondition:** $A[1..m]$ and $B[1..n]$ are arrays of Keyed items sorted in nondecreasing order.

**Postcondition:** Returns an array $C[1..m+n]$ containing the elements of $A$ and $B$ in nondecreasing order, maintaining the same order as in the input arrays. If $A[i].Key() = B[j].Key()$, then $A[i]$ precedes $B[j]$ in $C$.

`Merge(A[1..m], B[1..n])`

$C \leftarrow \text{new} \ \text{Array}[1..m+n] \ ; \ i \leftarrow 1 ; \ j \leftarrow 1 ; \ k \leftarrow 1$

// **Invariant:** $C[1..k-1]$ contains $A[1..i-1]$ and $B[1..j-1]$ in correct order.

**while** $i \leq m$ **and** $j \leq n$

if $A[i].Key() \leq B[j].Key()$

$C[k] \leftarrow A[i] ; \ i \leftarrow i + 1$

else

$C[k] \leftarrow B[j] ; \ j \leftarrow j + 1$

$k \leftarrow k + 1$

`Copy(A[i..m], C[k..k+m-i]); Copy(B[j..n], C[k..k+n-j])`

return $C$
By solving the resulting Dutch national flag problem, we will have partitioned the array into three sections:

- The first section consists of all elements with keys less than $p$.\text{Key}().
- The second section consists of all elements with keys equal to $p$.\text{Key}().
- The third section consists of all elements with keys greater than $p$.\text{Key}().

By sorting the first and third sections, we will have sorted the array. This general strategy is known as quick sort.

Following the divide-and-conquer paradigm, we would like to select the pivot so that after the array has been partitioned, the first and third sections have roughly the same number of elements. Thus, the median element would be a good choice for $p$. As we will show in the next section, it is possible to find the median in $\Theta(n)$ time in the worst case. We saw in Section 3.6 that the Dutch national flag problem can be solved in $\Theta(n)$ time. Because each of the two subproblems is at most half the size of the original problem, we can bound the running time of this sorting algorithm with the recurrence

$$f(n) \in 2f(n/2) + \Theta(n).$$

From Theorem 3.32, $f(n) \in \Theta(n \lg n)$.

However, it turns out that the overhead of choosing the median as the pivot is too expensive in practice, so that heap sort, for example, outperforms this algorithm. On the other hand, choosing an arbitrary element, such as the first, degrades the worst case performance. For example, suppose that the input array is already sorted and that all keys are distinct. If we always choose the first element as the pivot, then we always choose the smallest element. As a result, one of the two subproblems is empty, and the other contains all but one of the original elements. Because the empty subproblem can be sorted in $\Theta(1)$ time, we can describe the running time for such a case with the following recurrence:

$$f(n) \in f(n - 1) + \Theta(n).$$

From Theorem 3.31, $f(n) \in \Theta(n^2)$, so that the running time for this algorithm is in $\Omega(n^2)$ in the worst case. Observing that each element is chosen as a pivot at most once, we can easily see that $O(n^2)$ is an upper bound on the running time, so that the algorithm runs in $\Theta(n^2)$ time in the worst case.

It turns out that the versions of quick sort used most frequently do, in fact, run in $\Theta(n^2)$ time in the worst case. However, choosing the first element
or the last element) as the pivot is a bad idea, because an already-sorted array yields the worst-case performance. Furthermore, the performance is nearly as bad on a nearly-sorted array. To make matters worse, it is not hard to see that when the running time is in $\Theta(n^2)$, the stack usage is in $\Theta(n)$. Because we often need to sort a nearly-sorted array, we don’t want an algorithm that performs badly in such cases.

The above analyses illustrate that it is better for the pivot element to be chosen to be near the median than to be near the smallest (or equivalently, the largest) element. More generally, it illustrates why divide-and-conquer is often an effective algorithm design strategy: when a problem is reduced to multiple subproblems, it is best if these subproblems are the same size. For quick sort, we need a way to choose the pivot element quickly in such a way that it tends to be near the median.

One way to accomplish this is to choose the pivot element randomly. This algorithm is shown in Figure 10.3. In order to make the presentation easier to follow, we have specified the algorithm so that the array is indexed with arbitrary endpoints.

Let us now analyze the expected running time of QuickSort on an array of size $n$. We first observe that for any call in which $lo < hi$, the loop will execute at least once. Furthermore, by an easy induction on $n$, we can show that at most $n + 1$ calls have $lo \geq hi$. Because each of these calls requires $\Theta(1)$ time, a total of at most $O(n)$ time is used in processing the base cases. Otherwise, the running time is proportional to the number of times the loop executes over the course of the algorithm.

Each iteration of the loop involves comparing one pair of elements. For a given call to QuickSort, the pivot is compared to all elements currently in the array, then is excluded from the subsequent recursive calls. Thus, once a pair of elements is compared, they are never compared again on subsequent loop iterations (though they may be compared twice in the same iteration — once in each if statement). The total running time is therefore proportional to the number of pairs of elements that are compared. We will only concern ourselves with pairs of distinct elements, as this will only exclude $O(n)$ pairs.

Let $F[1..n]$ be the final sorted array, and let $comp$ be a discrete random variable giving the number of pairs $(i, j)$ such that $1 \leq i < j \leq N$ and $F[i]$ is compared with $F[j]$. We wish to compute $E[comp]$. Let $c_{ij}$ denote the
Figure 10.3 The randomized QUICKSORT algorithm

**Precondition:** $A[lo..hi]$ is an array of Keyed items, and $lo$ and $hi$ are Ints.

**Postcondition:** $A[lo..hi]$ is a permutation of its original values in nondecreasing order.

```
QUICKSORT($A[lo..hi]$)
    if $lo < hi$
        $p ← A[\text{RANDOMINTEGER}(lo, hi)].\text{Key}()$
        $r ← 0; w ← 0; b ← 0$
        // Invariant: $r, w, b ∈ \mathbb{N}, r + w + b ≤ hi - lo + 1$, $A[i].\text{Key}() < p$
        // for $lo ≤ i < lo + r$, $A[i].\text{Key}() = p$ for $hi - b - w < i ≤ hi - b$, and
        // $A[i].\text{Key}() > p$ for $hi - b < i ≤ hi$.
        while $r + w + b < hi - lo + 1$
            $j ← hi - b - w$
            if $A[j].\text{Key}() < p$
                $A[j] ↔ A[lo + r]; r ← r + 1$
            else if $A[j].\text{Key}() = p$
                $w ← w + 1$
            else
                $A[j] ↔ A[hi - b]; b = b + 1$
        QUICKSORT($A[lo..lo + r - 1]$)
        QUICKSORT($A[hi - b + 1..hi]$)
```

**Precondition:** $i$ and $j$ are integers, $i ≤ j$.

**Postcondition:** Returns an integer $k$ such that $i ≤ k ≤ j$. Each value in the range has an equal probability, independent of previous calls.

```
RANDOMINTEGER($i, j$)
```
event that $F[i]$ is compared with $F[j]$. Then

$$E[\text{comp}] = E \left[ \sum_{i=1}^{n} \sum_{j=i+1}^{n} I(c_{ij}) \right]$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[I(c_{ij})]$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} P(c_{ij}).$$

We observe that $F[i]$ is compared with $F[j]$ iff one of them is in the subarray being sorted when the other is chosen as the pivot. Furthermore, two elements $F[i]$ and $F[j]$ are in the same subarray as long as no element $k$ such that $F[i].\text{Key}() \leq F[k].\text{Key}() \leq F[j].\text{Key}()$ is chosen as the pivot. Thus, the probability that $F[i]$ and $F[j]$ are compared is the probability that one of them is chosen as pivot before any other $F[k]$ satisfying the above inequality. Because there are at least $j - i + 1$ elements $F[k]$ satisfying the above inequality when $j > i$,

$$P(c_{ij}) \leq \frac{2}{j - i + 1}.$$

We therefore have

$$E[\text{comp}] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} P(c_{ij})$$

$$\leq \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}$$

$$= 2 \sum_{i=1}^{n} \sum_{j=2}^{n-i+1} \frac{1}{j}.$$  \hspace{1cm} (10.3)

The inner sum above is closely related to the harmonic series:

$$H_n = \sum_{i=1}^{n} \frac{1}{i}.$$  

Tight bounds for $H_n$ are given by the following theorem, whose proof is left as an exercise.
Theorem 10.1 For all \( n \geq 1 \):

\[
\ln(n + 1) \leq H_n \leq 1 + \ln n.
\]

Applying Theorem 10.1 to inequality (10.3), we have

\[
E[\text{comp}] \leq 2 \sum_{i=1}^{n} \sum_{j=2}^{n-i+1} \frac{1}{j}
\]

\[
= 2 \sum_{i=1}^{n} (H_{n-i+1} - 1)
\]

\[
\leq 2 \sum_{i=1}^{n} \ln(n - i + 1)
\]

\[
= 2 \sum_{i=1}^{n} \ln i
\]

\[
\in O(n \log n)
\]

from Theorem 3.28. For an array of distinct elements, a similar analysis shows that \( E[\text{comp}] \in \Omega(n \log n) \); hence, the expected running time of QuickSort on any array of \( n \) elements is in \( \Theta(n \log n) \).

The expected-case analysis of QuickSort suggests that it would work well in practice, and indeed, there are versions that outperform both heap sort and merge sort. The most widely-used versions, however, are not randomized. Instead of choosing the pivot element randomly, they use heuristics, such as choosing the median of the first, middle, and last elements. Such heuristics tend to choose pivot elements nearer to the median than those chosen randomly. Furthermore, they typically involve less overhead than generating a random (or pseudorandom) number. Though these variations have worst-case running times in \( \Theta(n^2) \), the inputs that result in poor performance are so pathological that they rarely occur in practice.

10.4 Selection

In Section 1.1, we introduced the selection problem. Recall that this problem is to find the \( k \)th smallest element of an array of \( n \) elements. We showed that it can be reduced to sorting. Using either heap sort or merge sort, we therefore have an algorithm for this problem with a running time in \( \Theta(n \log n) \). In this section, we will improve upon this running time.
Section 2.4 shows that the selection problem can be reduced to the Dutch National Flag problem and a smaller instance of itself. This reduction is very similar to the reduction upon which quick sort is based. Specifically, we choose a pivot element \( p \) and solve the resulting Dutch national flag problem as we did for the quick sort reduction. Let \( r \) and \( w \) denote the numbers of red items and white items, respectively. We then have three cases:

- If \( r \geq k \), we return the \( k \)th smallest red item.
- If \( r < k \) and \( r + w \geq k \), we return \( p \).
- If \( r + w < k \), we return the \( (k - r - w) \)th smallest blue element.

Due to the similarity of this algorithm to quick sort, some of the same problems arise in choosing the pivot element appropriately. For example, if we always use the first element as the pivot, then selecting the \( n \)th smallest element in a sorted array of \( n \) distinct elements always results in a recursive call with all but one of the original elements. As we saw in Section 10.3, this yields a running time in \( \Theta(n^2) \). On the other hand, it is possible to show that selecting the pivot at random yields an expected running time in \( \Theta(n) \) — the details are left as an exercise.

Our goal is to construct a deterministic algorithm with worst-case running time in \( O(n) \). As we saw in Section 10.3, quick sort achieves a better asymptotic running time if the median is chosen as the pivot. It stands to reason that such a choice might be best for the selection algorithm. We mentioned in Section 10.3 that it is possible to find the median in \( O(n) \) time. The way to do this is to use our linear-time selection algorithm to find the \( \lceil n/2 \rceil \)nd smallest element. However, this doesn’t help us in designing the linear-time selection algorithm because the reduction is not to a smaller instance.

Instead, we need a way to approximate the median well enough so that the resulting algorithm runs in \( O(n) \) time. Consider the following strategy for approximating the median. First, we arrange the \( n \) elements into an \( M \times \lfloor n/M \rfloor \) array, where \( M \) is some fixed odd number. If \( n \) is not evenly divisible by \( M \), we will have up to \( M - 1 \) elements that will not fit in the array — we will ignore these elements for now. Suppose we sort each column in nondecreasing order. Further suppose that we order the columns (keeping each column intact) so that the middle row is in nondecreasing order. We then select an element in the center of the array as the pivot \( p \) (see Figure 10.4).
By choosing $p$ in this fashion, we ensure that all elements above and to the left of $p$ are no greater than $p$, and that all elements below and to the right of $p$ are no less than $p$. In other words, no more than about $3/4$ of the elements can be greater than $p$, and no more than about $3/4$ of the elements can be less than $p$. Thus, if we can find this $p$ in $O(n)$ time, we have the following recurrence giving the running time of the algorithm:

$$f(n) \in f(\lfloor 3n/4 \rfloor) + \Theta(n).$$

Unfortunately, Theorem 3.32 does not apply to this recurrence, as we would need $b = 4/3$, which is not a natural number. However, given Theorem 3.32, we might suspect that $f(n) \in \Theta(n)$, as this is the solution yielded by the theorem if we could use $b = 4/3$. We will soon show that this is, in fact, the case.

Let us now consider the time needed to find $p$. Sorting $M$ elements using either heap sort or merge sort uses $\Theta(M \lg M)$ time. Because $M$ is a fixed constant, however, $M \lg M$ is also a fixed constant, so that the time is in $\Theta(1)$. This must be done for each of the $\lceil n/M \rceil$ columns, so that the time to sort all of the columns is in $O(n)$. However, the time to sort the $\lfloor n/M \rfloor$ elements in the middle row is in $\Theta(n \lg n)$. On the other hand, we really don’t need to sort the middle row in order to find $p$ — we only need to find the median of this row. If $M > 1$, this row has strictly fewer than $n$ elements, so that finding its median is a smaller instance of the selection problem.

If we use this technique for finding the median, we need two recursive calls — one to find the median of $\lfloor n/M \rfloor$ elements, and one to solve the
smaller selection problem after the Dutch national flag algorithm has been applied. This latter recursive call has no more than about $3n/4$ elements. The recurrence describing the running time is therefore of the form

$$f(n) \in f(\lfloor 3n/4 \rfloor) + f(\lfloor n/M \rfloor) + \Theta(n), \quad (10.4)$$

for some odd number $M$.

We don’t have a theorem that applies to the above recurrence. We can gain some intuition by comparing this recurrence with recurrences of the form

$$g(n) \in ag(\lfloor n/b \rfloor) + \Theta(n),$$

where $b$ is an integer greater than 1. From Theorem 3.32, $g(n) \in \Theta(n)$ iff $a < b$, or equivalently, iff $a/b < 1$. If $a$ is a positive integer, this condition is equivalent to

$$\sum_{i=1}^{a} \frac{1}{b} < 1.$$

The following theorem generalizes this condition to a sum of several different positive real numbers.

**Theorem 10.2** Let $n_0 \geq 1$ and $m \geq 1$ be integers, and let $c_1, \ldots, c_m$ be positive real numbers such that

$$\sum_{i=1}^{m} c_i < 1.$$

Let $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ be an eventually nondecreasing function satisfying

$$f(n) \in \sum_{i=1}^{m} f(\lfloor c_i n \rfloor) + X(n)$$

whenever $n \geq n_0$, where $X$ is either $O$, $\Omega$, or $\Theta$. Then $f(n) \in X(n)$.

**Proof:** If $X$ is $\Omega$, clearly we have $f(n) \in \Omega(n)$. In what follows, we will show that if $X$ is $O$, then $f(n) \in O(n)$. It will then follow that if $X$ is $\Theta$, then $f(n) \in \Theta(n)$.

Suppose $X$ is $O$. Then for some natural number $n_1$ and some positive real number $a$, we have

$$f(n) \leq \sum_{i=1}^{m} f(\lfloor c_i n \rfloor) + an$$
whenever \( n \geq n_0 \) and \( n \geq n_1 \). By Exercise 3.8, it is sufficient to show that for some positive real number \( b \), \( f(n) \leq bn \) whenever \( n > 0 \).

The technique we will use to prove this fact is called \textit{constructive induction}. With this technique, we use the constant \( b \) as if it had already been fixed. As the proof progresses, we will need to assume certain constraints on \( b \). Once the proof is finished, if the assumed constraints are consistent, we can then fix a value for \( b \) that satisfied all of the constraints.

\textbf{Induction Hypothesis:} Assume that for some \( n \), if \( 0 < k < n \), then \( f(k) \leq bk \).

\textbf{Induction Step:} First, we will assume that \( n \) is large enough to apply the recurrence, then to apply the induction hypothesis to each term in the resulting summation. If \( n \geq \max(n_0, n_1) \),

\[
    f(n) \leq \sum_{i=1}^{m} f([c_i n]) + an.
\]

In order to be able to apply the induction hypothesis to each \( f([c_i n]) \), we need \( [c_i n] > 0 \). Therefore we assume that \( n \geq n_2 \), where

\[
    n_2 \geq \max\{n_0, n_1, 1/c_i \mid 1 \leq i \leq m\}.
\]

Let

\[
    c = \sum_{i=1}^{m} c_i.
\]

By the Induction Hypothesis,

\[
    f(n) \leq \sum_{i=1}^{m} b[c_i n] + an
\]

\[
    \leq bn \sum_{i=1}^{m} c_i + an
\]

\[
    = bcn + an
\]

\[
    = (bc + a)n.
\]

Thus, \( f(n) \leq bn \) if

\[
    bc + a \leq b
\]

\[
    a \leq b(1 - c)
\]

\[
    \frac{a}{1 - c} \leq b;
\]
as $1 - c > 0$.

**Base:** We must still show the claim for $0 < n < n_2$. In other words, we need $b \geq f(n)/n$ for $0 < n < n_2$. We can satisfy this constraint and the one above if $b = \max\{a/(1 - c), f(n)/n \mid 0 < n < n_2\}$ (note that because this set is finite and nonempty, it must have a maximum element). □

Returning to recurrence 10.4, we see that Theorem 10.2 applies if $M > 4$. Thus, if we set $M = 5$, we have $f(n) \in O(n)$. The entire algorithm is shown in Figure 10.5.

Now that we have described the algorithm precisely, let us analyze its running time more carefully to be sure that it is, in fact in $\Theta(n)$. It is easily seen that the running time is in $\Omega(n)$. We need for the recurrence

$$f(n) \in f([n/5]) + f([3n/4]) + O(n)$$

(10.5)

to give an upper bound on the running time of the algorithm for sufficiently large $n$. Clearly, the number of elements in the first recursive call is $[n/5]$. Furthermore, if we ignore the recursive calls, the time needed is in $O(n)$ (note that the time needed to sort 5 elements is bounded by a constant because the number of elements is constant). However, the number of elements in the second recursive call is not always bounded above by $[3n/4]$. Consider, for example, the array $A[1..13]$ with $A[i] = i$ for $1 \leq i \leq 13$. The values 3 and 8 will be placed in $T[1]$ and $T[2]$, respectively. The value assigned to $p$ will therefore be 3. If $k > 3$, ten elements will be passed to the second recursive call, but $[3 \cdot 13/4] = 9$.

Returning to Figure 10.4, and taking the number of rows to be 5, we see that for each two columns (ten elements) that we add, at least three elements are excluded from the second recursive call. For example, when $5 \leq n \leq 14$, at least three elements are excluded, and when $15 \leq n \leq 24$, at least six elements are excluded. In general, at least $3i$ elements are excluded if $10i - 5 \leq n \leq 10i + 4$. We need to find the largest value of $n$ such that $3i < n/4$, if such an $n$ exists. If we can do so, then for all larger values of $n$, the number of elements in the second recursive call will be no more than $3n/4$. Because the number of elements must be an integer, it will then be bounded above by $[3n/4]$.

We therefore need to find the largest $i$ such that

$$3i < (10i + 4)/4$$

$$12i < 10i + 4$$

$$i < 2.$$
Figure 10.5 A linear-time algorithm for the selection problem

\[
\text{\textsc{LinearSelect}}(A[1..n], k)
\]
\[
\begin{align*}
\text{if } n &\leq 4 \\
&\quad \text{Sort}(A[1..n]) \\
&\quad \text{return } A[k] \\
\text{else } &\quad m \leftarrow \lfloor n/5 \rfloor; T \leftarrow \text{new} \; \text{Array}[1..m]
\end{align*}
\]
\[
\begin{align*}
&\quad \text{for } i \leftarrow 1 \text{ to } m \\
&\quad \hspace{1em} \text{Sort}(A[5i - 4..5i]); T[i] \leftarrow A[5i - 2]
\end{align*}
\]
\[
\begin{align*}
p &\leftarrow \text{\textsc{LinearSelect}}(T[1..m], \lceil m/2 \rceil); r \leftarrow 0; w \leftarrow 0; b \leftarrow 0 \quad // \text{Invariant: } r, w, b \in \mathbb{N}, r + w + b \leq n, \text{ and } A[i] < p \text{ for } 1 \leq i \leq r, \\
&\quad // A[i] = p \text{ for } n - b - w < i \leq n - b, \text{ and } A[i] > p \text{ for } n - b < i \leq n.
\end{align*}
\]
\[
\begin{align*}
&\quad \text{while } r + w + b < n
\end{align*}
\]
\[
\begin{align*}
&\quad \hspace{1em} j \leftarrow n - b - w \\
&\quad \hspace{2em} \text{if } A[j] < p \\
&\quad \hspace{3em} r \leftarrow r + 1; A[j] \leftrightarrow A[r] \\
&\quad \hspace{1em} \text{else if } A[j] = p \\
&\quad \hspace{2em} w \leftarrow w + 1 \\
&\quad \hspace{1em} \text{else } \\
&\quad \hspace{2em} A[j] \leftarrow A[n - b]; b = b + 1
\end{align*}
\]
\[
\begin{align*}
&\quad \text{if } r \geq k \\
&\quad \hspace{1em} \text{return } \text{\textsc{LinearSelect}}(A[1..r], k) \\
&\quad \text{else if } r + w \geq k \\
&\quad \hspace{1em} \text{return } p \\
&\quad \text{else } \\
&\quad \hspace{1em} \text{return } \text{\textsc{LinearSelect}}(A[r + w + 1..n], k - r - w)
\end{align*}
\]
Precondition: \( u \) and \( v \) are BigNums.
Postcondition: Returns a BigNum representing the product of the values of \( u \) and \( v \).

**Multiply** \((u, v)\)

Precondition: \( u \) is a BigNum and \( v \) is a positive BigNum.
Postcondition: Returns a BigNum representing the value \( \lfloor u/v \rfloor \).

**Divide** \((u, v)\)

The largest such \( n \) is therefore \( 10 + 4 = 14 \). Then for all \( n \geq 15 \), recurrence (10.5) gives an upper bound on the running time of LinearSelect. From Theorem 10.2, the running time is in \( O(n) \), and hence in \( \Theta(n) \).

Various performance improvements can be made to LinearSelect. For example, if \( n = 5 \), there is no reason to apply the Dutch national flag algorithm after sorting the array — we can simply return \( A[k] \). In other words, it would be better if the base case included \( n = 5 \), and perhaps some larger values as well. Furthermore, sorting is not the most efficient way to solve the selection problem for small \( n \). We explore some alternatives in the exercises.

Even with these performance improvements, however, LinearSelect does not perform nearly as well as the randomized algorithm outlined at the beginning of this section. Better still is using a quick approximation of the median, such as finding the median of the first, middle, and last elements, as the value of \( p \). This approach yields an algorithm whose worst-case running time is in \( \Theta(n^2) \), but which typically performs better than even the randomized algorithm.

### 10.5 Integer Division

Exercise 4.14 on page 145 discussed an implementation of a BigNum ADT (specified in Figure 4.18, page 146) for arbitrary-precision natural numbers. This ADT is rather limited, in that its only arithmetic operations are addition and subtraction. Figure 10.6 specifies multiplication and division functions for operating on BigNums. We leave it as an exercise to show that
the polynomial multiplication algorithm of Section 10.1 can be adapted to form a BigNum multiplication algorithm that runs in $\Theta(n^{\lg 3})$ time, where $n$ is the number of bits in the product. In this section and the next, we consider implementations of Divide.

Let us first consider the familiar long division algorithm from elementary school. Suppose the dividend $u$ has $m$ digits and the divisor $v$ has $n$ digits. We begin by finding the smallest prefix of the significant digits of $u$ that gives us a number no smaller than $v$. If there is no such prefix, the quotient is 0. Otherwise, the next step is to obtain an approximation for the most significant digit of the result. Let $k$ be the number of digits in the prefix formed above. We obtain the approximation by dividing the $k - n + 1$ most significant digits of $u$ by the most significant digit of $v$. (Note that because $k$ will always be either $n$ or $n + 1$, $k - n + 1$ will be either 1 or 2.) This quotient is our approximation of the first significant digit of the result.

Because we only used one digit of $v$ in approximating the first significant digit of the result, this approximation may be too large. We determine whether this is the case by multiplying this approximation by $v$, then comparing the product with the prefix of $u$. If the product is larger, we repeatedly subtract 1 from the approximation and recompute the product until the product is no larger than the prefix of $u$. The resulting digit is the first significant digit of the result.

Having obtained the first significant digit of the result, we need to obtain the rest of the result. We begin by subtracting the last product obtained above from the prefix of $u$. We then append the rest of $u$ to this difference. Finally, we obtain the remaining digits of the quotient by dividing the resulting value by $v$. If we ensure that the result of this last division has exactly $m - k$ digits (by padding with zeros if necessary), then the digits we obtain complete the quotient. We have therefore reduced the division problem to a smaller instance of itself. The reduction is not quite a transformation, but it can easily be expressed as a loop.

This algorithm can easily be applied to the division of an $m$-bit binary number $u$ by an $n$-bit binary number $v$. In fact, we really don't need to approximate the first significant bit — it can't be a 0, so it must be a 1. We do, however, need to determine the prefix of $u$ as described above. We then subtract $v$ from the prefix and append the remainder of $u$. We obtain the remaining bits by dividing this value by $v$ and inserting zeros if necessary.

In the above algorithm, we perform a $\Theta(n)$-bit subtraction for each 1 bit of the quotient. Furthermore, each subtraction is preceded by at least one $\Theta(n)$-bit comparison. Because the number of 1 bits in the quotient may be as many as $m - n + 1$, the worst-case number of comparisons and
subtractions of $\Theta(n)$-bit numbers is $(m-n+1)$. The running time is therefore in $\Theta(n(m-n))$, which is worse than what we would like.

We can instead group the bits of $u$ and $v$ into digits of some radix $r$, where $r$ is a power of 2. However, for any fixed radix $r$, this decreases the number of comparisons by at most a constant factor, so that the asymptotic running time does not improve. Applying the divide-and-conquer principle, we might try breaking $v$ into two digits. For now, let’s assume the number of bits $n$ in $v$ is even. Our radix is therefore $2^{n/2}$. To obtain an approximation of the first digit of the quotient, we recursively divide the first one or two digits of $u$ by the first digit of $v$.

If $n$ is odd and greater than 1, we can multiply both $u$ and $v$ by 2 without changing the quotient. These multiplications have the effect of appending a zero as the least significant bit, so that the resulting length is even. Because $n$ is greater than 1, dividing the length of the resulting divisor by 2 yields a smaller subproblem. For $n = 1$, $v = 1$, so the quotient is simply $u$. We have therefore reduced the division problem to smaller instances of itself. The resulting algorithm is shown in Figure 10.7. We assume the existence of two constants zero and one, which refer to BigNums with values of 0 and 1, respectively.

We will now analyze the running time of DivideDC. We begin with the while loop. In order to analyze this loop, we need to know how large $\text{prod}$, $\text{rem}$, and $\text{approx}$ might be. From the invariant of the main loop, $\text{rem} \leq v - 1$ at the top of the main loop. Before the while loop is executed, the value of $\text{rem}$ is multiplied by $2^{n/2}$, and $\text{next}$ is added. Because $\text{next}$ contains at most $n/2$ significant bits, $\text{next} < 2^{n/2}$. Thus, when the while loop executes, $\text{rem} < v 2^{n/2}$. Because $v$ contains $n$ significant bits, $\text{rem}$ contains at most $3n/2$ significant bits. Likewise, it is not hard to show that at the beginning of the while loop, $\text{approx}$ contains at most $n/2 + 1$ significant bits, and that $\text{prod}$ contains at most $3n/2 + 1$ significant bits. The body of the while loop therefore runs in $\Theta(n)$ time in the worst case.

In order to get a tight bound on the number of iterations of the while loop, we need a tighter bound on $\text{approx}$. In particular, we need to know how close $\text{approx}$ is to $\lfloor \text{rem}/v \rfloor$. Let $r = \lfloor \text{rem} \times 2^{-n/2} \rfloor$. We first observe that

$$\left| \frac{r}{v \text{First} + 1} \right| = \left| \frac{r \times 2^{n/2}}{(v \text{First} + 1) \times 2^{n/2}} \right| = \left| \frac{\text{rem}}{(v \text{First} + 1) \times 2^{n/2}} \right|.$$
Figure 10.7 Divide-and-conquer implementation of Divide, specified in Figure 10.6

\[
\text{DIVIDE\text{\textunderscore}DC}(u, v)
\]
\[
m \leftarrow u.\text{NUM\text{\textunderscore}BITS}(); \ n \leftarrow v.\text{NUM\text{\textunderscore}BITS}()
\]
\[
\text{if } \ n = 1
\]
\[
\quad \text{return } u
\]
\[
\text{else}
\]
\[
\quad \text{if } \ n \mod 2 = 1
\]
\[
\quad \quad u \leftarrow u.\text{SHIFT}(1); \ m \leftarrow m + 1
\]
\[
\quad \quad v \leftarrow v.\text{SHIFT}(1); \ n \leftarrow n + 1
\]
\[
\quad \quad \text{digLen} \leftarrow n/2; \ \text{numDig} \leftarrow \lceil m/\text{digLen} \rceil
\]
\[
\quad \quad \text{qLen} \leftarrow \text{digLen} \times (\text{numDig} - 1)
\]
\[
\quad \quad \text{qBits} \leftarrow \text{new ARRAY}[0..\text{qLen} - 1]
\]
\[
\quad \quad \text{vFirst} \leftarrow \text{new BigNum}(v.\text{GET\textunderscore BITS}(\text{digLen}, \text{digLen}))
\]
\[
\quad \quad \text{rem} \leftarrow \text{new BigNum}(u.\text{GET\textunderscore BITS}(\text{qLen}, \text{digLen}))
\]
\[
\quad \quad \text{// Invariant: } \text{rem} < v \quad \text{and}
\]
\[
\quad \quad \quad \text{v} \times \text{new BigNum}(\text{qBits}[i + \text{digLen}..\text{qLen} - 1]) + \text{rem} =
\]
\[
\quad \quad \quad \text{// new BigNum}(u.\text{GET\textunderscore BITS}(i + \text{digLen}, m - i - \text{digLen})).
\]
\[
\quad \text{for } i \leftarrow \text{qLen} - \text{digLen} \text{ to } 0 \text{ by } -\text{digLen}
\]
\[
\quad \quad \text{next} \leftarrow \text{new BigNum}(u.\text{GET\textunderscore BITS}(i, \text{digLen}))
\]
\[
\quad \quad \text{rem} \leftarrow \text{rem.SHIFT}(\text{digLen}).\text{ADD}(\text{next})
\]
\[
\quad \text{if } \text{rem.\text{COMPARE\textunderscore TO}(v)} < 0
\]
\[
\quad \quad \text{qDig} \leftarrow \text{zero.\text{GET\textunderscore BITS}(0, \text{digLen})}
\]
\[
\quad \text{else}
\]
\[
\quad \quad \text{approx} \leftarrow \text{DIVIDE\text{\textunderscore}DC}(\text{rem.\text{SHIFT}(-\text{digLen})}, \text{vFirst})
\]
\[
\quad \quad \text{prod} \leftarrow \text{MULTIPLY}(v, \text{approx})
\]
\[
\quad \quad \text{// Invariant: } \text{prod} = v \times \text{approx}, \ \text{approx} \geq \lfloor \text{rem}/v \rfloor, \text{ and the}
\]
\[
\quad \quad \quad \text{// invariant for the outer loop.}
\]
\[
\quad \quad \text{while } \text{prod.\text{COMPARE\textunderscore TO}(\text{rem})} > 0
\]
\[
\quad \quad \quad \text{approx} \leftarrow \text{approx.\text{SUBTRACT}(one)}
\]
\[
\quad \quad \quad \text{prod} \leftarrow \text{prod.\text{SUBTRACT}(v)}
\]
\[
\quad \quad \quad \text{qDig} \leftarrow \text{approx.\text{GET\textunderscore BITS}(0, \text{digLen})}
\]
\[
\quad \quad \quad \text{COPY}(\text{qDig}[0..\text{digLen} - 1], \text{qBits}[i..i + \text{digLen} - 1])
\]
\[
\text{return } \text{new BigNum}(\text{qBits})
\]
because the \( n/2 \) low-order bits of the numerator do not affect the value of the expression. Furthermore, the right-hand side above is no larger than \( \lfloor \text{rem}/v \rfloor \). Thus,

\[
\text{approx} - \lfloor \text{rem}/v \rfloor \leq \left[ \frac{r}{v_{\text{First}}} \right] - \left[ \frac{r}{v_{\text{First}} + 1} \right] \\
\leq \frac{r}{v_{\text{First}}} - \frac{r - v_{\text{First}}}{v_{\text{First}} + 1} \\
= \frac{r(v_{\text{First}} + 1) - v_{\text{First}}(r - v_{\text{First}})}{v_{\text{First}}(v_{\text{First}} + 1)} \\
= \frac{r + v_{\text{First}}^2}{v_{\text{First}}(v_{\text{First}} + 1)}.
\]

Now because \( \text{rem} < v_{\text{First}}^2 \), it follows that \( r < v_{\text{First}} \times 2^{n/2} \). We therefore have

\[
\text{approx} - \lfloor \text{rem}/v \rfloor \leq \frac{r + v_{\text{First}}^2}{v_{\text{First}}(v_{\text{First}} + 1)} \\
< \frac{v_{\text{First}} \times 2^{n/2} + v_{\text{First}}^2}{v_{\text{First}}(v_{\text{First}} + 1)} \\
= \frac{2^{n/2} + v_{\text{First}}}{v_{\text{First}} + 1} \\
= \frac{2^{n/2}}{v_{\text{First}} + 1} + \frac{v_{\text{First}}}{v_{\text{First}} + 1}.
\]

Because \( v_{\text{First}} \) contains \( n/2 \) significant bits, its value must be at least \( 2^{n/2 - 1} \). The value of the first term on the right-hand side above is therefore strictly less than 2. Clearly, the value of the second term is strictly less than 1, so that the right-hand side is strictly less than 3. Because the left-hand side is an integer, its value must therefore be at most 2. It follows from the \textbf{while} loop invariant that the loop terminates when \( \text{approx} = \lfloor \text{rem}/v \rfloor \). Because this loop decrements \( \text{approx} \) by 1 each iteration, it must iterate at most twice. Its running time is therefore \( \Theta(n) \).

It is now easily seen that, excluding the recursive call, the running time of the body of the main loop is dominated by the running time of the multiplication. Because the result of the multiplication contains at most \( 3n/2 + 1 \) significant bits, this multiplication can be done in \( \Theta(n^{\lg 3}) \) time using the multiplication algorithm suggested at the beginning of this section.
If \( m \geq n \), the number of iterations of the main loop is easily seen to be

\[
\text{numDig} - 1 = \left\lceil \frac{m}{\text{digLen}} \right\rceil - 1
\]

\[
= \left\lfloor \frac{m}{n/2} \right\rfloor - 1
\]

\[
= \left\lceil \frac{2m}{n} \right\rceil - 1.
\]

Thus, the running time of the main loop, excluding the recursive call, is in

\[
\Theta(n^{\lg 3}m/n) = \Theta(mn^{\lg 3-1}).
\]

We now observe that for even \( n \), there are in the worst case \([2m/n] - 1\) recursive calls. For odd \( n \), the worst-case number of recursive calls is \([2(m + 1)/(n + 1)] - 1\). The resulting recurrence is therefore quite complicated. However, consider the parameters of the recursive call. We have already shown that \( \text{rem} < v^{2n/2} \), and \( \text{vFirst} = \lceil v2^{-n/2} \rceil \). This recursive call therefore divides a value strictly less than \( v \) by \( \lceil v2^{-n/2} \rceil \). Thus, in any of these calls, the dividend is less than the divisor plus 1, multiplied by \( 2^n \), where \( n \) is the number of bits in the divisor. In addition, it is easily seen that the dividend is never less than the divisor. Furthermore, if these relationships initially hold for odd \( n \), they hold for the recursive call in this case as well. We therefore will first restrict our attention to this special case.

Let \( n \), the number of significant bits in \( v \), be even. If

\[
v \leq u < (v + 1)2^n,
\]

then \( m \), the number of significant bits in \( u \), is at most \( 2n \). The number of iterations of the outer loop is therefore at most

\[
\left\lfloor \frac{2m}{n} \right\rfloor - 1 \leq \left\lfloor \frac{4n}{n} \right\rfloor - 1
\]

\[
= 3.
\]

Because each iteration may contain a recursive call, this suggests that there are a total of at most 3 recursive calls. However, note that whenever a recursive call is made, the dividend is no less than the divisor, so that a nonzero digit results in the quotient. Suppose the first of the three digits of the quotient is nonzero. Because the first \( n \) bits of \( u \) are at most \( v \), the only possible nonzero result for the first digit is 1. The remainder of the quotient is then formed by dividing a value strictly less than \( 2^n \) by \( v \), which is at least \( 2^{n-1} \). This result is also at most 1, so that the second digit must be 0. We conclude that no more than two recursive calls are ever made. In each of these recursive calls, the divisor has \( n/2 \) bits.
If \( n \) is odd and greater than 1, we increase the number of bits in \( v \) by 1. The above reasoning then applies to \( n + 1 \), where \( n \) denotes the original number of bits in \( v \). We can therefore express the overall running time in terms of \( n \) via the recurrence

\[
f(n) \in 2f(\lceil n/2 \rceil) + \Theta(n^{\lg 3})
\]

for \( n > 1 \). Applying Theorem 3.32, we see that \( f(n) \in \Theta(n^{\lg 3}) \).

Let us now turn to the more general case. If \( m < n \), it is easily seen that the running time is in \( \Theta(n) \). If \( m \geq n \), as we have already shown, the worst-case number of recursive calls is in \( \Theta(m/n) \), and the overhead is in \( \Theta(mn^{\lg 3 - 1}) \). Because each of these recursive calls satisfies the special case analyzed above, each runs in \( \Theta(n^{\lg 3}) \) time. Thus, the overall running time is in \( \Theta(mn^{\lg 3 - 1}) \).

If we wish to express the running time in terms of the number of bits in the larger of the two operands, it is easily seen that the worst case occurs when \( m \) is larger, but \( n \) is a fixed fraction of \( m \). If \( N \) denotes the number of bits in the larger operand, we then see that the running time is in \( \Theta(N^{\lg 3}) \), which is asymptotically the same as multiplication. Furthermore, it is not hard to see that if we can improve the running time of multiplication to \( O(N^{1+\epsilon}) \) for any fixed positive \( \epsilon \), the running time of division will also be in \( O(N^{1+\epsilon}) \) (or \( O(m^{\epsilon}) \)).

In a later chapter, we will show that the running time of multiplication can be improved to \( O(N \lg N \lg \lg N) \). When we use this running time, the recurrence for the case in which

\[
v \leq u < (v + 1)2^n,
\]

becomes

\[
f(n) \in 2f(\lceil n/2 \rceil) + O(n \lg n \lg \lg n)
\]

for \( n > 1 \). Applying Theorem 3.32 to this recurrence yields

\[
f(n) \in O(n \lg^2 n \lg \lg n).
\]

The running time of the division algorithm is then in \( O(N \lg^2 N \lg \lg N) \), which is slightly worse than the running time of the multiplication algorithm used. In the next section, we will design a division algorithm whose asymptotic running time matches that of multiplication even for the asymptotically fastest known multiplication algorithm.
10.6 * Newton’s Method

We can reduce division to multiplication in a straightforward way if we can compute a reciprocal. A reciprocal of an arbitrary positive integer is a fraction that may or may not have a finite binary representation. We therefore will have to settle for an approximation. In order to simplify the discussion of fixed-point fractions, it helps to scale the value of the divisor \( v \) to an appropriate range. Specifically, suppose \( v \) consists of \( n \) bits; i.e., \( 2^{n-1} \leq v < 2^n \). Then

\[
\left\lfloor \frac{u}{v} \right\rfloor = \left\lfloor u 2^{-n} \left( \frac{1}{v 2^{-n}} \right) \right\rfloor.
\]

Note that \( 1/2 \leq v 2^{-n} < 1 \). Thus, if we can compute a close fixed-point approximation for the reciprocal of a value \( y \) such that \( 1/2 \leq y < 1 \), we can reduce the integer division problem to the integer multiplication problem. Note that due to the given range of \( y \), the binary encoding of \( y \) has no bits to the left of the radix point, and the first bit to the right is a 1. Furthermore, because \( 1 < 1/y \leq 2 \), we can approximate \( 1/y \) with a value \( z \) such that \( 1 \leq z < 2 \); i.e., \( z \) has a single 1 bit to the left of the radix point. We can therefore use BigNums to represent both \( y \) and \( 1/y \) with the interpretation that each contains a radix point at the appropriate position.

Because multiplying an approximation of the reciprocal of \( v 2^{-n} \) by \( u 2^{-n} \) gives only an approximation of \( \left\lfloor u/v \right\rfloor \), we may need to correct our result. Suppose we can approximate the reciprocal to enough accuracy that the result of the multiplication is a value \( q \) such that

\[
|q - \frac{u}{v}| \leq 1.
\]

Then it is not hard to see that

\[
\left| \left\lfloor q \right\rfloor - \left\lfloor \frac{u}{v} \right\rfloor \right| \leq 1.
\]

Then if \( v \lfloor q \rfloor \leq u - v \), we know that the actual quotient is \( \lfloor q \rfloor + 1 \). If \( v \lfloor q \rfloor > u \), we know that the quotient is \( \lfloor q \rfloor - 1 \). Otherwise, the quotient is \( \lfloor q \rfloor \).

Suppose an error of \( \epsilon \) is introduced in approximating the reciprocal. Then we need

\[
\left| u 2^{-n} \left( \frac{1}{v 2^{-n}} + \epsilon \right) - \frac{u}{v} \right| \leq 1
\]

\[
\left| \frac{u}{v} + \epsilon u 2^{-n} - \frac{u}{v} \right| \leq 1
\]

\[
|\epsilon| \leq 2^n / u.
\]
Figure 10.8 Partial implementation of Divide, specified in Figure 10.6, using an approximate reciprocal

\[
\text{DivideRecip}(u, v) \begin{align*}
\text{if } &u.\text{CompareTo}(v) < 0 \\
& \quad \text{return zero} \\
\text{else} & \\
&m \leftarrow u.\text{NumBits}(); n \leftarrow v.\text{NumBits}() \\
&r \leftarrow \text{Reciprocal}(v, m - n) \\
&q \leftarrow \text{Multiply}(u, r).\text{Shift}(1 - r.\text{NumBits}() - n); \\
&\text{prod} \leftarrow \text{Multiply}(v, q) \\
&\text{if } \text{prod.\text{CompareTo}}(u.\text{Subtract}(v)) \leq 0 \\
&\quad q \leftarrow q.\text{Add}(\text{one}) \\
&\text{else if } \text{prod.\text{CompareTo}}(u) > 0 \\
&\quad q \leftarrow q.\text{Subtract}(\text{one}) \\
&\text{return } q
\end{align*}
\]

\textbf{Precondition:} y refers to a nonzero \texttt{BigInt}, and \(k\) refers to a natural number.

\textbf{Postcondition:} Returns a \texttt{BigInt} \(z\) such that

\[
\left|\frac{z2^{1-z.\text{NumBits}()}}{y2^{-u.\text{NumBits}()}} - \frac{1}{y2^{-u.\text{NumBits}()}}\right| \leq 2^{-k}.
\]

\textbf{Reciprocal}(y, k)

Suppose \(u\) consists of \(m\) bits, so that \(u < 2^m\). Then we can ensure that the approximation of \(u/v\) differs from the actual value of \(u/v\) by no more than 1 if our approximation of the reciprocal of \(v2^{-n}\) differs from the actual reciprocal by at most \(2^{n-m}\).

The resulting algorithm is shown in Figure 10.8. We handle the case in which \(u < v\) separately in order to ensure that the precondition for \(u.\text{Subtract}(v)\) is met. As in the previous section, we use constants \texttt{zero} and \texttt{one}, which refer to \texttt{BigInt} representing 0 and 1, respectively. It is easily seen that the running time is simply the time for \textbf{Reciprocal} plus the time to do the two multiplications. The time to do the first multiplication depends on the size of the value returned by \textbf{Reciprocal}. Because the accuracy of the approximation is \(2^{n-m}\), we would expect the value to be not
much more than $m - n$ significant bits.

In the remainder of this section, we will consider how to implement the \texttt{Reciprocal} function specified in Figure 10.8. The technique we apply is \textit{Newton's method} for approximating a root of a function. Let $I$ be some interval of the real numbers, and suppose $f : I \rightarrow \mathbb{R}$ has at least one root — a value $x \in I$ such that $f(x) = 0$. For example, if $y$ is a fixed positive real number, the function $f(x) = 1/x - y$ over $\mathbb{R}^0$ has exactly one root, namely, $x = 1/y$. Newton’s method is an iterative approach to finding an approximation of a root of $f$.

Newton’s method begins with an initial estimate $x_0$ of the root. If $f(x_0)$ is not sufficiently close to 0, a better approximation is found using the derivative of $f$, which we will denote by $f'$. Recall that $f'(x_0)$ gives the slope of the line tangent to $f$ at $x_0$ (see Figure 10.9). We can easily find the intersection $x_1$ of this line with the $x$-axis, and for many functions, this intersection will be a better approximation to the root than the initial estimate. We then apply Newton’s method using $x_1$ as the initial estimate. For many functions, this approach is guaranteed to approach a root very quickly. As we will see, the function $f(x) = 1/x - y$ is such a function.

The line tangent to $f$ at $x_0$ has slope $f'(x_0)$ and includes the point $(x_0, f(x_0))$. To find its $x$-intercept, we need to go to the left of $x_0$ a distance of $f(x_0)/f'(x_0)$ (or if this value is negative, we go to the right a distance of
−f(x₀)/f′(x₀)). The new estimate \( x_1 \) is therefore given by

\[
x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.
\]

If \( f(x) = 1/x - y \), then \( f'(x) = -x^{-2} \). The new estimate is therefore

\[
x_1 = x_0 - \frac{1/x_0 - y}{-x_0^{-2}} = x_0 + x_0 - yx_0^2 = 2x_0 - yx_0^2.
\]

In order to see how quickly the Newtonian iteration converges to \( 1/y \), suppose we have an estimate \( x_0 = (1 + \epsilon)/y \), where \( \epsilon \) is some real number. Applying the iteration, we have

\[
x_1 = 2x_0 - yx_0^2 = \frac{2(1 + \epsilon)}{y} - y \left( \frac{1 + \epsilon}{y} \right)^2 = \frac{2 + 2\epsilon - 1 - 2\epsilon - \epsilon^2}{y} = \frac{1 - \epsilon^2}{y}.
\]

Thus, each iteration squares the error term \( \epsilon \). For \( 1/2 \leq y < 1 \), it is not hard to show that an initial estimate of \( 3/2 \) yields

\[
-1/4 \leq \epsilon < 1/2;
\]

hence, the number of bits of accuracy doubles with each iteration.

Although this technique does converge rapidly to \( 1/y \), the number of iterations depends on the degree of accuracy we require. Unfortunately, each iteration requires a multiplication by \( y \), so that the time required cannot be proportional to the time for a single multiplication of values having roughly the same size as \( y \). However, note that successive iterations give successively better approximations. For the earlier approximations, which will probably not be very accurate anyway, we need not use all of the bits of \( y \) in the computation.

This suggests the following approach. Suppose we need an approximation that differs from the actual reciprocal by no more than \( 2^{-k} \). We will use \( k \) as the size of this problem instance. If \( k \) is not too small, we first
solve a smaller instance in order to obtain a less-accurate approximation. The accuracy that we require of this approximation needs to be such that a single application of the Newtonian iteration will yield an accuracy of within $2^{-k}$. In applying this iteration, we only use as many bits of $y$ as we need in order to ensure the required accuracy. Finally, in order to keep the number of bits in the approximation from growing too rapidly, we return only as many bits as we need to ensure the required accuracy.

Let $\alpha \in \mathbb{R}$ denote the absolute error of some estimate; i.e., our estimate is $1/y + \alpha$. Let $\beta \in \mathbb{R}^{\geq 0}$ denote the absolute error introduced by truncating $y$, so that the value we use for $y$ in the iteration is $y - \beta$. Finally, let $\gamma \in \mathbb{R}^{\geq 0}$ denote the absolute error introduced by truncating the result. The value computed by the iteration is therefore

$$2 \left( \frac{1}{y + \alpha} \right) - (y - \beta) \left( \frac{1}{y + \alpha} \right)^2 - \gamma = \frac{1}{y} + \frac{\beta}{y^2} + \frac{2\alpha\beta}{y} + \alpha^2\beta - y\alpha^2 - \gamma.$$ 

We need for this value to differ from $1/y$ by at most $2^{-k}$; i.e., we need

$$\left| \frac{\beta}{y^2} + \frac{2\alpha\beta}{y} + \alpha^2\beta - y\alpha^2 - \gamma \right| \leq 2^{-k}. \tag{10.7}$$

Note that because $y > 0$, $\beta \geq 0$, and $\gamma \geq 0$, all terms except the second are always nonnegative. In order to ensure that the inequality holds when the value inside the absolute value bars is nonnegative, we can therefore ignore the last two terms. We therefore need

$$\frac{\beta}{y^2} + \frac{2\alpha\beta}{y} + \alpha^2\beta \leq 2^{-k}.$$ 

If we replace $\alpha$ by $|\alpha|$ in the above inequality, the left-hand side does not decrease. For fixed $\alpha$ and $\beta$, the resulting left-hand side is maximized when $y$ is minimized. Setting $y$ to its minimum possible value of $1/2$, it therefore suffices to ensure that

$$4\beta + 4|\alpha|\beta + \alpha^2\beta \leq 2^{-k}.$$ 

In order to keep the first term sufficiently small, we need $\beta < 2^{-k-2}$. In order to leave room for the other two terms, let us take $\beta \leq 2^{-k-3}$. In other words, we will use the first $k + 3$ bits of $y$ in applying the iteration. Then as long as $|\alpha| \leq 1/2$, we have

$$4\beta + 4\alpha\beta + \alpha^2\beta \leq 2^{-k-1} + 2^{-k-2} + 2^{-k-5} \leq 2^{-k}.$$
Let us now consider the case in which the value inside the absolute value bars in (10.7) is negative. We can now ignore the first and third terms. We therefore need
\[ y\alpha^2 + \gamma - \frac{2\alpha\beta}{y} \leq 2^{-k}. \]
Here, we can safely replace \( \alpha \) by \(-|\alpha|\). For fixed \( \alpha, \beta, \) and \( \gamma \) in the resulting inequality, the first term is maximized when \( y \) is maximized, but the third term is maximized when \( y \) is minimized. It therefore suffices to ensure that
\[ \alpha^2 + \gamma + 4|\alpha|\beta \leq 2^{-k}. \]
Again taking \( \beta \leq 2^{-k-3} \), we only need \(|\alpha| \leq 2^{-(k+1)/2} \) and \( \gamma \leq 2^{-k-2} \). We then have
\[
\begin{align*}
\alpha^2 + \gamma + 4|\alpha|\beta & \leq 2^{-k-1} + 2^{-k-2} + 2^{k-1-(k+1)/2} \\
& \leq 2^{-k},
\end{align*}
\]
provided \( k \geq 1 \).

We can satisfy the constraints on \( \alpha \) and \( \gamma \) by finding an approximation within \( 2^{-\lceil (k+1)/2 \rceil} \), and returning \( k + 3 \) bits of the result of applying the iteration (recall that the result has one bit to the left of the radix point). Note that if we take \( k \) as the size of the problem instance, we are reducing the problem to an instance roughly half the original size. We therefore have a divide-and-conquer algorithm.

In order to complete the algorithm, we need to handle the base cases. Because \( \lceil (k+1)/2 \rceil < k \) only when \( k > 2 \), these cases occur for \( k \leq 2 \). It turns out that these cases are important for ensuring that the approximation is at least 1 and strictly less than 2. From (10.6), the result of the iteration is never more than \( 1/y \) (here \( y \) denotes the portion we are actually using in computing the iteration). Thus, if \( y > 1/2 \), the estimate is less than 2. Furthermore, if \( y = 1/2 \), an initial estimate less than 2 will ensure that some error remains, so that the result is still strictly less than 2. Finally, provided \( \epsilon < 1 \), the result is always closer to \( 1/y \) than the initial estimate. Thus, if we make sure that our base case gives a value that is less than 2 and no worse an estimate than 1 would be, the approximation will always be in the proper range.

We leave it as an exercise to show that the estimate
\[
\frac{11 - \lfloor 8y \rfloor}{4}
\]
satisfies the specification and the requirements discussed above for \( k \leq 2 \). \( \lfloor 8y \rfloor \) is simply the first 3 bits of \( y \). Because \( 1/2 \leq y < 1 \), \( 4 \leq \lfloor 8y \rfloor < 8 \). The
CHAPTER 10. DIVIDE AND CONQUER

Figure 10.10 Implementation of Reciprocal, specified in Figure 10.8, using Newton’s method with divide-and-conquer

RecipNewton(y, k)
  n ← y.NumBits(); len ← k + 3
  if k ≤ 2
    return eleven.Subtract(y.Shift(3 − n))
  else
    x0 ← RecipNewton(y, ⌈(k + 1)/2⌉)
    y ← y.Shift(len − n)
    t ← x0.Shift(len + x0.NumBits())
    x1 ← t.Subtract(Multiply(y, Multiply(x0, x0)))
  return x1.Shift(len − x1.NumBits())

numerator is therefore always a 3-bit natural number. The final division by 4 simply puts the radix point in the proper place.

The algorithm is shown in Figure 10.10. We use the variable len to store the value k + 3, which, except in the base case, is both the number of bits we use from y and the number of bits we return. We assume the existence of a constant eleven referring to a BigNum with value 11. Before we do the subtraction, we must make sure the radix points in the operands line up. The approximation x0 has one bit to the left of the implicit radix point. The multiplication of x0 by 2 simply moves the radix point to the right one place. As a result, the implicit radix point in 2x0 is x0.NumBits() − 2 from the right in x0. The implicit radix point in the product yx0^2 is len + 2(x0.NumBits() − 1) bits from the right. In order for the radix points to line up, we therefore need to pad the value stored in x0 with len + x0.NumBits() zeros prior to subtracting.

Let us now analyze the running time of RecipNewton. Suppose we use a multiplication algorithm that runs in O(M(n)) time, where n is the number of bits in the product. For now, we will assume that M(n) is a smooth function in Ω(n), but we will strengthen this assumption as the analysis proceeds. It is easily seen that for k ≥ 3, the number of bits returned by RecipNewton is k + 3. Therefore, for k ≥ 4, the worst-case number of bits in the first product is 2k + 6. Because we use k + 3 bits of y, the worst-case number of bits in the second product is 3k + 9. Because M
is smooth, from Exercise 3.18, the time required for the two multiplications is in $O(M(k))$.

Because the remainder of the operations, excluding the recursive call, run in linear time, the total time excluding the recursive call is in $O(M(k))$. The total running time is therefore given by the recurrence

$$f(k) \in f\left(\left\lceil \frac{k+1}{2} \right\rceil \right) + O(M(k))$$

for $k \geq 4$. We can simplify this recurrence by defining $f_1(k) = f(k+1)$. Thus, for $k \geq 4$,

$$f_1(k) = f(k+1)$$

$$\in f\left(\left\lceil \frac{k+2}{2} \right\rceil \right) + O(M(k+1))$$

$$= f\left(\left\lceil \frac{k}{2} \right\rceil + 1 \right) + O(M(k+1))$$

$$= f_1(\lceil k/2 \rceil) + O(M(k+1))$$

$$= f_1(\lceil k/2 \rceil) + O(M(k)),$$

because $M$ is smooth.

In order to be able to apply Theorem 3.32 to $f_1$, we need additional assumptions on $M$. We therefore assume that $M(k) = k^q g(k)$, where $q \geq 1$ and $g_1(k) = g(2^{k+2})$ is smooth. (Note that the functions $k \lg^3 k$ and $k \lg k \lg \lg k \lg \lg k$ both satisfy these assumptions on $M$.) Then from Theorem 3.32, $f_1(k) \in O(M(k))$. Because $M$ is smooth, $f(k) = f_1(k-1) \in O(M(k))$.

We can now analyze the running time of \textsc{DivideRecip}. If $m < n$, the running time is clearly in $\Theta(1)$. Suppose $m \geq n$. Then the value $r$ returned by \textsc{Reciprocal}(v, m - n) has $m - n + 3$ bits in the worst case. Hence, the result of the first multiplication has $2m - n + 3$ bits in the worst case. The worst-case running time of this multiplication is therefore in $O(M(m))$. $q$ then has $m - n + 1$ bits in the worst case. The result of the second multiplication therefore has $m + 1$ bits in the worst case, and hence runs in $O(M(m))$ time. Because \textsc{Reciprocal} runs in $O(M(m-n))$ time, and the remaining operations run in $O(m)$ time, the overall running time is in $O(M(m))$. The running time for \textsc{DivideRecip} is therefore the same as for multiplication, even if our multiplication algorithm runs in $O(n \lg n \lg \lg n)$ time.
10.7 Summary

The divide-and-conquer technique involves reducing large instances of a problem to one or more smaller instances, each of which is a fraction of the size of the original problem. The running time of the resulting algorithm can typically be analyzed by deriving a recurrence to which Theorem 3.32 applies. Theorem 3.32 can also suggest how to improve a divide-and-conquer algorithm.

Some variations of the divide-and-conquer technique don’t completely fit the above description. For example, quick sort does not necessarily produce subproblems whose sizes are a fraction of the size of the original array. As a result, Theorem 3.32 does not apply. However, we still consider quick sort to be a divide-and-conquer algorithm because its goal is to partition an array into two arrays of approximately half the size of the input array, and to sort these arrays recursively. Likewise, in LinearSelect, the sizes of the two recursive calls are very different, but because they are both fractions of the original size, the analysis ends up being related to that of a more standard divide-and-conquer algorithm. Finally, DivideDC does not divide the problem into a bounded number of subproblems; however, all of the recursive calls in turn yield at most two recursive calls, so we can analyze these calls using standard divide-and-conquer techniques.

10.8 Exercises

Exercise 10.1 Prove that PolyMult, shown in Figure 10.1, meets its specification.

Exercise 10.2 PolyMult is not particularly efficient when one polynomial has a degree much larger than that of the other. For example, if \( p \) has degree \( n \) and \( q \) has degree 1, a straightforward implementation of the definition of the product yields \( \Theta(n) \) running time. Devise an algorithm that runs in \( \Theta(mn^{\log_3 3-1}) \) time on polynomials of degree \( m \) and \( n \) with \( m \geq n \). Your algorithm may use PolyMult. Analyze the running time of your algorithm. [Hint: If \( m > n \), divide the larger polynomial into polynomials of degree at most \( n \).]

* Exercise 10.3 Construct a divide-and-conquer polynomial multiplication algorithm that performs 5 recursive calls on polynomials of 1/3 the size of the original polynomials. Show that your algorithm has a running time in \( \Theta(n^{\log_3 5}) \). (Note that \( \log_3 5 < \log_3 3 \).)
** Exercise 10.4** Generalize Exercise 10.3 by showing that for sufficiently large \( n \) and any \( k \geq 2 \), the product of two degree-\((n - 1)\) polynomials can be computed from the products of \( 2k - 1 \) polynomials of degree approximately \((n/k) - 1\). Using this result, show that for any \( \epsilon \in \mathbb{R}^{>0} \), there is an algorithm to multiply two degree-\((n - 1)\) polynomials in \( O(n^{1+\epsilon}) \) time.

**Exercise 10.5** Adapt POLYMULT to implement MULTIPLY, as specified in Figure 10.6, in \( \Theta(n^{\log_2 3}) \) time, where \( n \) is the number of bits in the product.

**Exercise 10.6** Prove that MERGESORT, shown in Figure 10.2, meets its specification.

**Exercise 10.7** Suppose we are given a tape containing a large number of KEYED items to be sorted. The number of items is too large to fit into main memory, but we have three additional tapes we can use, and we can rewrite the input tape. Give a bottom-up version of merge sort that produces the sorted output on one of the tapes. You may not assume that data items on the tapes can be accessed “randomly” — they must be accessed in sequence. Your algorithm must make at most \( O(\log n) \) passes through each tape.

**Exercise 10.8** Prove that QUICKSORT, shown in Figure 10.3, meets its specification.

**Exercise 10.9** Notice that one of the recursive calls in QUICKSORT is tail recursion. Taking advantage of this fact, convert one of the recursive calls to iteration. Notice that the calls can be made in either order, and so either may be converted to iteration. Make the proper choice so that the resulting algorithm uses \( \Theta(\log n) \) stack space in the worst case on an array of \( n \) elements.

* **Exercise 10.10** The goal of this exercise is to prove Theorem 10.1.

a. For \( x \leq y \), let \([x, y]\) denote the set of all real numbers \( a \) such that \( x \leq a \leq y \). For natural numbers \( m < n \), let \( f : [m, n] \to \mathbb{R}^{\geq 0} \) be a continuous function such that whenever \( m \leq x < y \leq n \), \( f(x) \geq f(y) \) (i.e., \( f \) is nonincreasing). Prove that

\[
\sum_{i=m+1}^{n} f(i) \leq \int_{m}^{n} f(x)dx \leq \sum_{i=m}^{n-1} f(i).
\]

b. Use the result of part a. to prove Theorem 10.1.
Exercise 10.11 Prove that for an array of size $n$, QuickSort (shown in Figure 10.3) makes a total of at most $n + 1$ calls (including the initial call and all recursive calls, as appropriate) in which $lo \geq hi$.

Exercise 10.12 A randomized algorithm for the selection problem can be obtained by replacing the first assignment statement of SelectBy-Median (Figure 2.7 on page 43) with the statement:

$$p \leftarrow A[\text{RANDOMINTEGER}(1, n)]$$

Show that the expected running time of this algorithm is in $\Theta(n)$. [Hint: Your analysis should be similar to the analysis of QuickSort in Section 10.3.]

Exercise 10.13 Let $n_0 \geq 1$ and $m \geq 1$ be integers, let $q$ be a positive real number, and let $c_1, \ldots, c_m$ be positive real numbers such that

$$\sum_{i=1}^{m} c_i^q < 1.$$

Let $f : \mathbb{N} \rightarrow \mathbb{R}^\geq 0$ be an eventually nondecreasing function satisfying

$$f(n) \in \sum_{i=1}^{m} f([c_i n]) + X(n^q)$$

whenever $n \geq n_0$, where $X$ is either $O$, $\Omega$, or $\Theta$. Prove that $f(n) \in X(n^q)$.

* Exercise 10.14 Let $n_0 \geq 1$ and $m \geq 2$ be integers, let $q$ be a positive real number, and let $c_1, \ldots, c_m$ be positive real numbers such that

$$\sum_{i=1}^{m} c_i^q = 1.$$

Let $f : \mathbb{N} \rightarrow \mathbb{R}^\geq 0$ be an eventually nondecreasing function satisfying

$$f(n) \in \sum_{i=1}^{m} f([c_i n]) + O(n^q)$$

whenever $n \geq n_0$. Prove that $f(n) \in O(n^q \log n)$.

Exercise 10.15 Prove that LinearSelect, shown in Figure 10.5, meets the specification given in Figure 1.2 (p. 6).
Exercise 10.16 Determine the number of comparisons used by each of the following algorithms when sorting 4 elements.

a. INSERTIONSORT, shown in Figure 1.7 on page 11.

b. MERGESORT, shown in Figure 10.2.

Exercise 10.17 Repeat Exercise 10.16 for 5 elements.

Exercise 10.18 Show that it is possible to find either the smallest or largest of \( n \) elements using at most \( n - 1 \) comparisons.

* Exercise 10.19 Show that it is possible to find either the second largest or second smallest of \( n \) elements using at most \( n + \lceil \lg n \rceil - 2 \) comparisons.

* Exercise 10.20 Show that it is possible to find the median of five elements using at most six comparisons.

* Exercise 10.21 Prove that DIVIDE-DC, shown in Figure 10.7, meets its specification as given in Figure 10.6.

Exercise 10.22 Prove that DIVIDE-RECIP, shown in Figure 10.8, meets its specification as given in Figure 10.6.

* Exercise 10.23 Let \( 1/2 \leq y < 1 \).

a. Show that if
\[
x_0 = \frac{11 - \lfloor 8y \rfloor}{4},
\]
then
\[
\left| \frac{1}{y} - x_0 \right| \leq \frac{1}{4}.
\]

b. Show that if \( x_0 \) is as defined in part a, then
\[
\left| \frac{1}{y} - x_0 \right| \leq \left| \frac{1}{y} - 1 \right|.
\]

c. Prove that RECIP-NEWTON meets its specification as given in Figure 10.8.
CHAPTER 10. DIVIDE AND CONQUER

Figure 10.11 Additional functions for BigNums

Precondition: $u$ is a BigNum and $k$ is a Nat.
Postcondition: Returns a BigNum representing the value of $u^k$. We assume that $0^0 = 1$.

Power($u$, $k$)

Precondition: $u$ is a BigNum.
Postcondition: Returns a String (see Figure 4.17 and Exercise 4.13) containing the decimal representation $u$.

ToString($u$)

Exercise 10.24 Design a divide-and-conquer algorithm that implements Power as specified in Figure 10.11. Your algorithm should run in $O(M(n))$ time, where $n$ is the number of bits in the result and $M(n)$ is the time needed for Multiply when the product contains $n$ bits. You may make reasonable assumptions about $M(n)$, provided $n^{\log_3 3}$ and $n \log n \log \log n$ satisfy these assumptions.

* Exercise 10.25 Design a divide-and-conquer algorithm that implements ToString as specified in Figure 10.11. Your algorithm should run in $O(n^q)$ time, where $n$ is the number of bits in $u$, assuming Multiply needs $O(n^q)$ time to produce an $n$-bit product. You may assume $q$ is a real number strictly larger than 1.

* Exercise 10.26 Given two natural numbers $u$ and $v$ which are not both 0, the greatest common divisor of $u$ and $v$ (or gcd($u$, $v$)) is the largest integer that evenly divides both $u$ and $v$.

a. Prove that for any positive integers $u$ and $v$, $\gcd(u, v) = \gcd(v, u \mod v)$.

b. Design a divide-and-conquer algorithm that takes as input two positive integers $u$ and $v$ and returns $\gcd(u, v)$. Your algorithm should run in $O(\log \max(u, v))$ time.
* Exercise 10.27  Given two positive integers $u$ and $m$ such that $u < m$, a multiplicative inverse of $u$ mod $m$ is any positive integer $v$ such that $1 \leq v < m$ and $(uv) \mod m = 1$.

a. Prove that for any positive integers $u$ and $v$, there exist integers $a$ and $b$ such that $au + bv = \gcd(u, v)$.

b. Prove that $u$ has a multiplicative inverse mod $m$ iff $\gcd(u, m) = 1$. [Hint: See Theorem 7.4 on page 265.]

c. Prove that for $1 \leq u < m$, $u$ has at most one multiplicative inverse mod $m$.

d. Give an efficient divide-and-conquer algorithm that takes as input positive integers $u$ and $m$ such that $u < m$ and returns the multiplicative inverse of $u$ mod $m$, or nil if no inverse exists. Your algorithm should run in $O(\lg m)$ time. [Hint: Modify the algorithm for Exercise 10.26 to find $a$ and $b$ as described in part a.]

Exercise 10.28  The Manhattan Skyline Problem can be stated as follows. We are given a description of $n$ rectangular buildings on the horizon. Each description is a triple, $\langle l_i, w_i, h_i \rangle$, where $l_i$ is the $x$-coordinate of the building’s left-hand edge, $w_i$ is the width of the building, and $h_i$ is the height above the horizon of the building’s roof. (Note that the buildings may overlap.) We wish to construct the skyline produced by these buildings. The skyline is represented by a sequence of points $\langle (x_1, y_1), \ldots, (x_k, y_k) \rangle$, ordered by $x$-coordinate, representing the locations where a vertical segment of the skyline meets a horizontal segment leading to the right (see Figure 10.12). Note that the value of $y_k$ must always be 0. Give a divide-and-conquer algorithm to compute the Manhattan skyline, and show that your algorithm runs in $\Theta(n \lg n)$ time.

Exercise 10.29  A majority element of an array $A[1..n]$ is an element that occurs more than $n/2$ times in the array. Construct an efficient divide-and-conquer algorithm to find the majority element of $A$ if one exists. Your algorithm may only compare elements for equality (hence, it may not sort the elements). Analyze the worst-case running time of your algorithm. ($\Theta(n)$ is possible.)

Exercise 10.30  Give a divide-and-conquer algorithm to construct a round-robin tournament involving $n$ competitors. The tournament consists of a series of rounds. In each round, every competitor plays one other competitor
if \( n \) is even; if \( n \) is odd, exactly one competitor is idle each round. Every competitor must play every other competitor exactly once in the tournament. Therefore, the number of rounds is \( n - 1 \) if \( n \) is even, or \( n \) if \( n \) is odd. Let the competitors be identified by the natural numbers 0, \ldots, \( n - 1 \). Your algorithm should produce a 2-dimensional array \( A \) of natural numbers such that \( A[i, j] \) indicates \( j \)'s opponent in round \( i \); if \( j \) is idle in round \( i \), \( A[i, j] \) should be \( n \). Your algorithm should run in \( \Theta(n^2) \) time. A possible output for \( n = 5 \) is shown below.

\[
\begin{array}{cccccc}
0 & 1 & 2 & 3 & 4 \\
0 & 1 & 0 & 5 & 4 & 3 \\
1 & 2 & 4 & 0 & 5 & 1 \\
2 & 3 & 2 & 1 & 0 & 5 \\
3 & 4 & 5 & 3 & 2 & 0 \\
4 & 5 & 3 & 4 & 1 & 2 \\
\end{array}
\]

* Exercise 10.31* Give a \( \Theta(n \lg n) \) divide-and-conquer algorithm for determining the closest pair of points in a given collection. You may assume that the points are given via two arrays: \( x[1..n] \) and \( y[1..n] \), where \( n \geq 2 \). The \( n \) points are then \((x[1], y[1]), (x[2], y[2]), \ldots, (x[n], y[n])\). The distance
between a pair of points \((x, y)\) and \((x', y')\) is given by
\[
\sqrt{(x - x')^2 + (y - y')^2}.
\]
Your algorithm should return the minimum distance separating any two distinct points.

* Exercise 10.32 Give a divide-and-conquer algorithm for computing \(\lfloor \sqrt{n} \rfloor\), where \(n\) is a BigNum. Your algorithm’s running time should be in \(O(M(\lg n))\), where \(M(n)\) is as defined in Section 10.6.

** Exercise 10.33 Give a \(\Theta(n\lg^7)\) divide-and-conquer algorithm for multiplying two \(n \times n\) matrices of real numbers. [Hint: Find a way to multiply two \(2 \times 2\) matrices using only 7 scalar multiplications. Use this technique as the basis for a divide-and-conquer algorithm.]

* Exercise 10.34 A Hamiltonian path in a (directed or undirected) graph is a path that contains each vertex exactly once. A directed graph is said to be complete if for each pair of distinct vertices \(i\) and \(j\), either \((i, j)\) or \((j, i)\) is an edge in the graph. It turns out that every complete directed graph has a Hamiltonian path. Give a divide-and-conquer algorithm that finds a Hamiltonian path in a given complete directed graph. Your algorithm should run in \(O(n \lg n)\) time in the worst case, where \(n\) is the number of vertices in the graph, assuming the graph is implemented as a MatrixGraph.

10.9 Chapter Notes

The POLYMULT algorithm is based on a \(\Theta(n\lg^3)\) large-integer multiplication algorithm by Karatsuba and Ofman [73]. The DIVIDEDC algorithm is due to Burnikel and Ziegler [19]. The RECIPNEWTON algorithm is a topdown adaptation of an algorithm given by Knuth [77]; he credits the idea to Cook. Solutions to Exercises 10.3, 10.4, 10.24, and 10.25, can be found in Knuth [77]. Implementations of variations on the DIVIDEREcip and RECIPNEWTON, as well as the algorithms suggested by Exercises 10.24 and 10.25, can be found on this textbook’s web site. Furthermore, the calculator demo posted there provides a framework for testing implementations of arbitrary-precision arithmetic algorithms.

Merge sort was one of the earliest algorithms developed for electronic computers, being developed by von Neumann in 1945 [106, 78]. Exercise 10.7 is based on work by Eckert and Mauchly [32]. Quick sort was developed by
Hoare [59]. See Bentley and McIlroy [13] for a good practical implementation of quick sort.

Algorithm LinearSelect is due to Blum, et al. [15]. The solution to Exercise 10.19 is due to Aigner [4].

The solution to Exercise 10.31 is due to Bentley [11]. The solution to Exercise 10.33 is due to Strassen [101].