3.9 Analyzing Recursive Functions

Determining the running time of a function that calls itself recursively requires more work than analyzing nonrecursive functions. The analysis for a recursive function requires that we associate with each function $F$ in a program an unknown running time $T_F(n)$. This unknown function represents $F$'s running time as a function of $n$, the size of $F$'s arguments. We then establish an inductive definition, called a recurrence relation for $T_F(n)$, that relates $T_F(n)$ to functions of the form $T_G(k)$ for the other functions $G$ in the program and their associated argument sizes $k$. If $F$ is directly recursive, then one or more of the $G$'s will be the same as $F$.

The value of $T_F(n)$ is normally established by an induction on the argument size $n$. Thus, it is necessary to pick a notion of argument size that guarantees functions are called with progressively smaller arguments as the recursion proceeds. The requirement is the same as what we encountered in Section 2.9, when we tried to prove statements about recursive programs. That should be no surprise, because a statement about the running time of a program is just one example of something that we might try to prove about a program.

Once we have a suitable notion of argument size, we can consider two cases:

1. The argument size is sufficiently small that no recursive calls will be made by $F$. This case corresponds to the basis in an inductive definition of $T_F(n)$.

2. For larger argument sizes, one or more recursive calls may occur. Note that whatever recursive calls $F$ makes, whether to itself or to some other function $G$, will be made with smaller arguments. This case corresponds to the inductive step in the definition of $T_F(n)$.

The recurrence relation defining $T_F(n)$ is derived by examining the code for function $F$ and doing the following:

a) For each call to a function $G$ or use of a function $G$ in an expression (note that $G$ may be $F$), use $T_G(k)$ as the running time of the call, where $k$ is the appropriate measure of the size of the arguments in the call.

b) Evaluate the running time of the body of function $F$, using the same techniques as in previous sections, but leaving terms like $T_G(k)$ as unknown functions, rather than concrete functions such as $n^2$. These terms cannot generally be combined with concrete functions using simplification tricks such as the summation rule. We must analyze $F$ twice — once on the assumption that $F$'s argument size $n$ is sufficiently small that no recursive function calls are made, and once assuming that $n$ is not that small. As a result, we obtain two expressions for the running time of $F$ — one (the basis expression) serving as the basis of the recurrence relation for $T_F(n)$, and the other (the induction expression) serving as the inductive part.

c) In the resulting basis and induction expressions for the running time of $F$, replace big-oh terms like $O(f(n))$ by a specific constant times the function involved — for example, $c f(n)$.

d) If $a$ is a basis value for the input size, set $T_F(a)$ equal to the basis expression resulting from step (c) on the assumption that there are no recursive calls. Also, set $T_F(n)$ equal to the induction expression from (c) for the case where $n$ is not a basis value.
int fact(int n)
{
    if (n <= 1)
        return 1; /* basis */
    else
        return n*fact(n-1); /* induction */
}

Fig. 3.23. Program to compute $n!$.

The running time of the entire function is determined by solving this recurrence relation. In Section 3.11, we shall give general techniques for solving recurrences of the kind that arise in the analysis of common recursive functions. For the moment, we solve these recurrences by ad hoc means.

**Example 3.24.** Let us reconsider the recursive program from Section 2.7 to compute the factorial function; the code is shown in Fig. 3.23. Since there is only one function, fact, involved, we shall use $T(n)$ for the unknown running time of this function. We shall use $n$, the value of the argument, as the size of the argument. Clearly, recursive calls made by fact when the argument is $n$ have a smaller argument, $n - 1$ to be precise.

For the basis of the inductive definition of $T(n)$ we shall take $n = 1$, since no recursive call is made by fact when its argument is 1. With $n = 1$, the condition of line (1) is true, and so the call to fact executes lines (1) and (2). Each takes $O(1)$ time, and so the running time of fact in the basis case is $O(1)$. That is, $T(1)$ is $O(1)$.

Now, consider what happens when $n > 1$. The condition of line (1) is false, and so we execute only lines (1) and (3). Line (1) takes $O(1)$ time, and line (3) takes $O(1)$ for the multiplication and assignment, plus $T(n - 1)$ for the recursive call to fact. That is, for $n > 1$, the running time of fact is $O(1) + T(n - 1)$. We can thus define $T(n)$ by the following recurrence relation:

**Basis.** $T(1) = O(1)$.

**Induction.** $T(n) = O(1) + T(n - 1)$, for $n > 1$.

We now invent constant symbols to stand for those constants hidden within the various big-oh expressions, as was suggested by rule (c) above. In this case, we can replace the $O(1)$ in the basis by some constant $a$, and the $O(1)$ in the induction by some constant $b$. These changes give us the following recurrence relation:

**Basis.** $T(1) = a$.

**Induction.** $T(n) = b + T(n - 1)$, for $n > 1$.

Now we must solve this recurrence for $T(n)$. We can calculate the first few values easily. $T(1) = a$ by the basis. Thus, by the inductive rule, we have
\[ T(2) = b + T(1) = a + b \]

Continuing to use the inductive rule, we get
\[ T(3) = b + T(2) = b + (a + b) = a + 2b \]

Then
\[ T(4) = b + T(3) = b + (a + 2b) = a + 3b \]

By this point, it should be no surprise if we guess that
\[ T(n) = a + (n - 1)b, \quad \text{for all } n \geq 1. \]

Indeed, computing some sample values, then guessing a solution, and finally proving our guess correct by an inductive proof is a common method of dealing with recurrences.

In this case, however, we can derive the solution directly by a method known as repeated substitution. First, let us make a substitution of variables, \( m \) for \( n \), in the recursive equation, which now becomes
\[ T(m) = b + T(m - 1), \quad \text{for } m > 1 \quad (3.3) \]

Now, we can substitute \( n, n - 1, n - 2, \ldots, 2 \) for \( m \) in equation (3.3) to get the sequence of equations
\[
\begin{align*}
1) \quad T(n) & = b + T(n - 1) \\
2) \quad T(n - 1) & = b + T(n - 2) \\
3) \quad T(n - 2) & = b + T(n - 3) \\
& \quad \vdots \\
n - 1) \quad T(2) & = b + T(1)
\end{align*}
\]

Next, we can use line (2) above to substitute for \( T(n - 1) \) in (1), to get the equation
\[ T(n) = b + (b + T(n - 2)) = 2b + T(n - 2) \]

Now, we use line (3) to substitute for \( T(n - 2) \) in the above to get
\[ T(n) = 2b + (b + T(n - 3)) = 3b + T(n - 3) \]

We proceed in this manner, each time replacing \( T(n - i) \) by \( b + T(n - i - 1) \), until we get down to \( T(1) \). At that point, we have the equation
\[ T(n) = (n - 1)b + T(1) \]

We can then use the basis to replace \( T(1) \) by \( a \), and conclude that \( T(n) = a + (n - 1)b \).

To make this analysis more formal, we need to prove by induction our intuitive observations about what happens when we repeatedly substitute for \( T(n - i) \). Thus we shall prove the following by induction on \( i \):

**STATEMENT** \( S(i) \): If \( 1 \leq i < n \), then \( T(n) = ib + T(n - i) \).

**BASIS.** The basis is \( i = 1 \). \( S(1) \) says that \( T(n) = b + T(n - 1) \). This is the inductive part of the definition of \( T(n) \) and therefore known to be true.

**INDUCTION.** If \( i \geq n - 1 \), there is nothing to prove. The reason is that statement \( S(i + 1) \) begins, “If \( 1 \leq i + 1 < n \cdot \cdot \cdot \),” and when the condition of an if-statement is false, the statement is true regardless of what follows the “then.” In this case, where \( i \geq n - 1 \), the condition \( i + 1 < n \) must be false, so \( S(i + 1) \) is true.
The hard part is when \( i \leq n-2 \). In that case, \( S(i) \) says that \( T(n) = ib + T(n-i) \). Since \( i \leq n-2 \), the argument of \( T(n-i) \) is at least 2. Thus we can apply the inductive rule for \( T \) — that is, (3.3) with \( n-i \) in place of \( m \) — to get the equation
\[
T(n-i) = b + T(n-i-1)
\]
When we substitute \( b + T(n-i-1) \) for \( T(n-i) \) in the equation \( T(n) = ib + T(n-i) \), we obtain \( T(n) = ib + (b + T(n-i-1)) \), or regrouping terms,
\[
T(n) = (i+1)b + T(n-i+1)
\]
This equation is the statement \( S(i+1) \), and we have now proved the induction step.

We have now shown that \( T(n) = a + (n-1)b \). However, \( a \) and \( b \) are unknown constants. Thus, there is no point in presenting the solution this way. Rather, we can express \( T(n) \) as a polynomial in \( n \), namely, \( bn + (a-b) \), and then replace terms by big-oh expressions, giving \( O(n) + O(1) \). Using the summation rule, we can eliminate \( O(1) \), which tells us that \( T(n) \) is \( O(n) \). That makes sense; it says that to compute \( n! \), we make on the order of \( n \) calls to \texttt{fact} (the actual number is exactly \( n \)), each of which requires \( O(1) \) time, excluding the time spent performing the recursive call to \texttt{fact}.

**EXERCISES**

3.9.1: Set up a recurrence relation for the running time of the function \texttt{sum} mentioned in Exercise 2.9.2, as a function of the length of the list that is input to the program. Replace big-oh’s by (unknown) constants, and try to solve your recurrence. What is the running time of \texttt{sum}?

3.9.2: Repeat Exercise 3.9.1 for the function \texttt{find0} from Exercise 2.9.3. What is a suitable size measure?

3.9.3*: Repeat Exercise 3.9.1 for the recursive selection sort program in Fig. 2.22 of Section 2.7. What is a suitable size measure?

3.9.4**: Repeat Exercise 3.9.1 for the function of Fig. 3.24, which computes the Fibonacci numbers. (The first two are 1, and each succeeding number is the sum of the previous two. The first seven Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13.) Note that the value of \( n \) is the appropriate size of an argument and that you need both 1 and 2 as basis cases.

```c
int fibonacci(int n)
{
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Fig. 3.24. C function computing the Fibonacci numbers.
3.9.5*: Write a recursive program to compute \( \text{gcd}(i, j) \), the greatest common divisor of two integers \( i \) and \( j \), as outlined in Exercise 2.7.8. Show that the running time of the program is \( O(\log i) \). \textit{Hint}: Show that after two calls we invoke \( \text{gcd}(m, n) \) where \( m \leq i/2 \).

\[ \]

3.10 Analysis of Merge Sort

We shall now analyze the merge sort algorithm that we presented in Section 2.8. First, we show that the \texttt{merge} and \texttt{split} functions each take \( O(n) \) time on lists of length \( n \), and then we use these bounds to show that the \texttt{MergeSort} function takes \( O(n \log n) \) time on lists of length \( n \).

Analysis of the Merge Function

We begin with the analysis of the recursive function \texttt{merge}, whose code we repeat as Fig. 3.25. The appropriate notion of size \( n \) for the argument of \texttt{merge} is the sum of the lengths of the lists \( \text{list1} \) and \( \text{list2} \). Thus, we let \( T(n) \) be the time taken by \texttt{merge} when the sum of the lengths of its argument lists is \( n \). We shall take \( n = 1 \) to be the basis case, and so we must analyze Fig. 3.25 on the assumption that one of \( \text{list1} \) and \( \text{list2} \) is empty and the other has only one element. There are two cases:

1. If the test of line (1) — that is, \( \text{list1} \) equals \texttt{NULL} — succeeds, then we return \( \text{list2} \), which takes \( O(1) \) time. Lines (2) through (7) are not executed. Thus, the entire function call takes \( O(1) \) time to test the selection of line (1) and \( O(1) \) time to perform the assignment on line (1), a total of \( O(1) \) time.

2. If the test of line (1) fails, then \( \text{list1} \) is not empty. Since we assume that the sum of the lengths of the lists is only 1, \( \text{list2} \) must therefore be empty. Thus, the test on line (2) — namely, \( \text{list2} \) equals \texttt{NULL} — must succeed. We then take \( O(1) \) time to perform the test of line (1), \( O(1) \) to perform the test of line (2), and \( O(1) \) to return \( \text{list1} \) on line (2). Lines (3) through (7) are not executed. Again, we take only \( O(1) \) time.

We conclude that in the basis case \texttt{merge} takes \( O(1) \) time.

Now let us consider the inductive case, where the sum of the list lengths is greater than 1. Of course, even if the sum of the lengths is 2 or more, one of the lists could still be empty. Thus, any of the four cases represented by the nested selection statements could be taken. The structure tree for the program of Fig. 3.25 is shown in Fig. 3.26. We can analyze the program by working from the bottom, up the structure tree.

The innermost selection begins with the “if” on line (3), where we test which list has the smaller first element and then either execute lines (4) and (5) or execute lines (6) and (7). The condition of line (3) takes \( O(1) \) time to evaluate. Line (5) takes \( O(1) \) time to evaluate, and line (4) takes \( O(1) \) time plus \( T(n - 1) \) time for the recursive call to \texttt{merge}. Note that \( n - 1 \) is the argument size for the recursive call, since we have eliminated exactly one element from one of the lists and left the other list as it was. Thus, the block of lines (4) and (5) takes \( O(1) + T(n - 1) \) time.

The analysis for the else-part in lines (6) and (7) is exactly the same: line (7) takes \( O(1) \) time and line (6) takes \( O(1) + T(n - 1) \) time. Thus, when we take
LIST merge(LIST list1, LIST list2)
{
    if (list1 == NULL) return list2;
    else if (list2 == NULL) return list1;
    else if (list1->element <= list2->element) {
        list1->next = merge(list1->next, list2);
        return list1;
    } else { /* list2 has smaller first element */
        list2->next = merge(list1, list2->next);
        return list2;
    }
}

Fig. 3.25. The function merge.

the maximum of the running times of the if- and else-parts, we find these times to be the same. The $O(1)$ for the test of the condition can be neglected, and so we conclude that the running time of the innermost selection is $O(1) + T(n - 1)$.

Now we proceed to the selection beginning on line (2), where we test whether list2 equals NULL. The time for testing the condition is $O(1)$, and the time for the if-part, which is the return on line (2), is also $O(1)$. However, the else-part is the selection statement of lines (3) through (7), which we just determined takes $O(1) + T(n - 1)$ time. Thus, the time for the selection of lines (2) through (7) is

$$O(1) + \max(O(1), O(1) + T(n - 1))$$

The second term of the maximum dominates the first and also dominates the $O(1)$ contributed by the test of the condition. Thus, the time for the if-statement beginning at line (2) is also $O(1) + T(n - 1)$.

Finally, we perform the same analysis for the outermost if-statement. Essentially, the dominant time is the else-part, which consists of lines (2) through (7).
A Common Form of Recursion

Many of the simplest recursive functions, such as fact and merge, perform some operation that takes $O(1)$ time and then make a recursive call to themselves on an argument one size smaller. Assuming the basis case takes $O(1)$ time, we see that such a function always leads to a recurrence relation $T(n) = O(1) + T(n-1)$. The solution for $T(n)$ is $O(n)$, or linear in the size of the argument. In Section 3.11 we shall see some generalizations of this principle.

That is, the time for the cases in which there is a recursive call, lines (4) and (5) or lines (6) and (7), dominates the time for the cases in which there is no recursive call, represented by lines (1) and (2), and also dominates the time for all three tests on lines (1), (2), and (3). Thus, the time for the function merge, when $n > 1$, is bounded above by $O(1) + T(n-1)$. We therefore have the following recurrence relation for defining $T(n)$:

**Basis.** $T(1) = O(1)$.

**Induction.** $T(n) = O(1) + T(n-1)$, for $n > 1$.

These equations are exactly the same as those derived for the function fact in Example 3.24. Thus, the solution is the same and we can conclude that $T(n)$ is $O(n)$. That result makes intuitive sense, since merge works by eliminating an element from one of the lists, taking $O(1)$ time to do so, and then calling itself recursively on the remaining lists. It follows that the number of recursive calls will be no greater than the sum of the lengths of the lists. Since each call takes $O(1)$ time, exclusive of the time taken by its recursive call, we expect the time for merge to be $O(n)$.

```c
LIST split(LIST list)
{
    LIST pSecondCell;
    if (list == NULL) return NULL;
    else if (list->next == NULL) return NULL;
    else { /* there are at least two cells */
        pSecondCell = list->next;
        list->next = pSecondCell->next;
        pSecondCell->next = split(pSecondCell->next);
        return pSecondCell;
    }
}
```

Fig. 3.27. The function split.
Analysis of the Split Function

Now let us consider the split function, which we reproduce as Fig. 3.27. The analysis is quite similar to that for merge. We let the size $n$ of the argument be the length of the list, and we here use $T(n)$ for the time taken by split on a list of length $n$.

For the basis, we take both $n = 0$ and $n = 1$. If $n = 0$ — that is, list is empty — the test of line (1) succeeds and we return NULL on line (1). Lines (2) through (6) are not executed, and we therefore take $O(1)$ time. If $n = 1$, that is, list is a single element, the test of line (1) fails, but the test of line (2) succeeds. We therefore return NULL on line (2) and do not execute lines (3) through (6). Again, only $O(1)$ time is needed for the two tests and one return statement.

For the induction, $n > 1$, there is a three-way selection branch, similar to the four-way branch we encountered in merge. To save time in analysis, we may observe — as we eventually concluded for merge — that we take $O(1)$ time to do one or both of the selection tests of lines (1) and (2). Also, in the cases in which one of these two tests is true, where we return on line (1) or (2), the additional time is only $O(1)$. The dominant time is the case in which both tests fail, that is, in which the list is of length at least 2; in this case we execute the statements of lines (3) through (6). All but the recursive call in line (5) contributes $O(1)$ time. The recursive call takes $T(n - 2)$ time, since the argument list is the original value of list, missing its first two elements (to see why, refer to the material in Section 2.8, especially the diagram of Fig. 2.28). Thus, in the inductive case, $T(n)$ is $O(1) + T(n - 2)$.

We may set up the following recurrence relation:

**BASIS.** $T(0) = O(1)$ and $T(1) = O(1)$.

**INDUCTION.** $T(n) = O(1) + T(n - 2)$, for $n > 1$.

As in Example 3.24, we must next invent constants to represent the constants of proportionality hidden by the $O(1)$’s. We shall let $a$ and $b$ be the constants represented by $O(1)$ in the basis for the values of $T(0)$ and $T(1)$, respectively, and we shall use $c$ for the constant represented by $O(1)$ in the inductive step. Thus, we may rewrite the recursive definition as

**BASIS.** $T(0) = a$ and $T(1) = b$.

**INDUCTION.** $T(n) = c + T(n - 2)$ for $n \geq 2$.

Let us evaluate the first few values of $T(n)$. Evidently $T(0) = a$ and $T(1) = b$ by the basis. We may use the inductive step to deduce

- $T(2) = c + T(0) = a + c$
- $T(3) = c + T(1) = b + c$
- $T(4) = c + T(2) = c + (a + c) = a + 2c$
- $T(5) = c + T(3) = c + (b + c) = b + 2c$
- $T(6) = c + T(4) = c + (a + 2c) = a + 3c$
The calculation of \( T(n) \) is really two separate calculations, one for odd \( n \) and the other for even \( n \). For even \( n \), we get \( T(n) = a + cn/2 \). That makes sense, since with an even-length list, we eliminate two elements, taking time \( c \) to do so, and after \( n/2 \) recursive calls, we are left with an empty list, on which we make no more recursive calls and take \( a \) time.

On an odd-length list, we again eliminate two elements, taking time \( c \) to do so. After \( (n - 1)/2 \) calls, we are down to a list of length 1, for which time \( b \) is required. Thus, the time for odd-length lists will be \( b + c(n - 1)/2 \).

The inductive proofs of these observations closely parallel the proof in Example 3.24. That is, we prove the following:

**STATEMENT** \( S(i) \): If \( 1 \leq i \leq n/2 \), then \( T(n) = ic + T(n - 2i) \).

In the proof, we use the inductive rule in the definition of \( T(n) \), which we can rewrite with argument \( m \) as

\[
T(m) = c + T(m - 2), \quad \text{for } m \geq 2
\]  

(3.4)

We may then prove \( S(i) \) by induction as follows:

**BASIS.** The basis, \( i = 1 \), is (3.4) with \( n \) in place of \( m \).

**INDUCTION.** Because \( S(i) \) has an if-then form, \( S(i + 1) \) is always true if \( i \geq n/2 \). Thus, the inductive step — that \( S(i) \) implies \( S(i+1) \) — requires no proof if \( i \geq n/2 \).

The hard case occurs when \( 1 \leq i < n/2 \). In this situation, suppose that the inductive hypothesis \( S(i) \) is true; \( T(n) = ic + T(n - 2i) \). We substitute \( n - 2i \) for \( m \) in (3.4), giving us

\[
T(n - 2i) = c + T(n - 2i - 2)
\]

If we substitute for \( T(n - 2i) \) in \( S(i) \), we get

\[
T(n) = ic + (c + T(n - 2i - 2))
\]

If we then group terms, we get

\[
T(n) = (i + 1)c + T(n - 2(i + 1))
\]

which is the statement \( S(i + 1) \). We have thus proved the inductive step, and we conclude \( T(n) = ic + T(n - 2i) \).

Now if \( n \) is even, let \( i = n/2 \). Then \( S(n/2) \) says that \( T(n) = cn/2 + T(0) \), which is \( a + cn/2 \). If \( n \) is odd, let \( i = (n - 1)/2 \). \( S((n - 1)/2) \) tells us that \( T(n) \) is \( c(n - 1)/2 + T(1) \) which equals \( b + c(n - 1)/2 \) since \( T(1) = b \).

Finally, we must convert to big-oh notation the constants \( a, b, \) and \( c \), which represent compiler- and machine-specific quantities. Both the polynomials \( a + cn/2 \) and \( b + c(n - 1)/2 \) have high-order terms proportional to \( n \). Thus, the question whether \( n \) is odd or even is actually irrelevant; the running time of \texttt{split} is \( O(n) \) in either case. Again, that is the intuitively correct answer, since on a list of length \( n \), \texttt{split} makes about \( n/2 \) recursive calls, each taking \( O(1) \) time.
LIST MergeSort(LIST list)
{
    LIST SecondList;
    (1)    if (list == NULL) return NULL;
    (2)    else if (list->next == NULL) return list;
    else {
        /* at least two elements on list */
        (3)    SecondList = split(list);
        (4)    return merge(MergeSort(list), MergeSort(SecondList));
    }
}

Fig. 3.28. The merge sort algorithm.

The Function MergeSort

Finally, we come to the function MergeSort, which is reproduced in Fig. 3.28. The appropriate measure \( n \) of argument size is again the length of the list to be sorted. Here, we shall use \( T(n) \) as the running time of MergeSort on a list of length \( n \).

We take \( n = 1 \) as the basis case and \( n > 1 \) as the inductive case, where recursive calls are made. If we examine MergeSort, we observe that, unless we call MergeSort from another function with an argument that is an empty list, then there is no way to get a call with an empty list as argument. The reason is that we execute line (4) only when list has at least two elements, in which case the lists that result from a split will have at least one element each. Thus, we can ignore the case \( n = 0 \) and start our induction at \( n = 1 \).

**Basis.** If list consists of a single element, then we execute lines (1) and (2), but none of the other code. Thus, in the basis case, \( T(1) \) is \( O(1) \).

**Induction.** In the inductive case, the tests of lines (1) and (2) both fail, and so we execute the block of lines (3) and (4). To make things simpler, let us assume that \( n \) is a power of 2. The reason it helps to make this assumption is that when \( n \) is even, the split of the list is into two pieces of length exactly \( n/2 \). Moreover, if \( n \) is a power of 2, then \( n/2 \) is also a power of 2, and the divisions by 2 are all into equal-sized pieces until we get down to pieces of 1 element each, at which time the recursion ends. The time spent by MergeSort when \( n > 1 \) is the sum of the following terms:

1. \( O(1) \) for the two tests
2. \( O(1) + O(n) \) for the assignment and call to split on line (3)
3. \( T(n/2) \) for the first recursive call to MergeSort on line (4)
4. \( T(n/2) \) for the second recursive call to MergeSort on line (4)
5. \( O(n) \) for the call to merge on line (4)
6. \( O(1) \) for the return on line (4).
Inductions that Skip Some Values

The reader should not be concerned by the new kind of induction that is involved in the analysis of MergeSort, where we skip over all but the powers of 2 in our proof. In general, if \(i_1, i_2, \ldots\) is a sequence of integers about which we want to prove a statement \(S\), we can show \(S(i_1)\) as a basis and then show for the induction that \(S(i_j)\) implies \(S(i_{j+1})\), for all \(j\). That is an ordinary induction if we think of it as an induction on \(j\). More precisely, define the statement \(S'\) by \(S'(j) = S(i_j)\). Then we prove \(S'(j)\) by induction on \(j\). For the case at hand, \(i_1 = 1, i_2 = 2, i_3 = 4, \) and in general, \(i_j = 2^{j-1}\).

Incidentally, note that \(T(n)\), the running time of MergeSort, surely does not decrease as \(n\) increases. Thus, showing that \(T(n)\) is \(O(n \log n)\) for \(n\) equal to a power of 2 also shows that \(T(n)\) is \(O(n \log n)\) for all \(n\).

If we add these terms, and drop the \(O(1)\)'s in favor of the larger \(O(n)\)'s that come from the calls to split and merge, we get the bound \(2T(n/2) + O(n)\) for the time spent by MergeSort in the inductive case. We thus have the recurrence relation:

**Basis.** \(T(1) = O(1)\).

**Induction.** \(T(n) = 2T(n/2) + O(n)\), where \(n\) is a power of 2 and greater than 1.

Our next step is to replace the big-oh expressions by functions with concrete constants. We shall replace the \(O(1)\) in the basis by constant \(a\), and the \(O(n)\) in the inductive step by \(bn\), for some constant \(b\). Our recurrence relation thus becomes

**Basis.** \(T(1) = a\).

**Induction.** \(T(n) = 2T(n/2) + bn\), where \(n\) is a power of 2 and greater than 1.

This recurrence is rather more complicated than the ones studied so far, but we can apply the same techniques. First, let us explore the values of \(T(n)\) for some small \(n\)'s. The basis tells us that \(T(1) = a\). Then the inductive step says

\[
T(2) = 2T(1) + 2b = 2a + 2b \\
T(4) = 2T(2) + 4b = 2(2a + 2b) + 4b = 4a + 8b \\
T(8) = 2T(4) + 8b = 2(4a + 8b) + 8b = 8a + 24b \\
T(16) = 2T(8) + 16b = 2(8a + 24b) + 16b = 16a + 64b
\]

It may not be easy to see what is going on. Evidently, the coefficient of \(a\) keeps pace with the value of \(n\); that is, \(T(n)\) is \(n\) times \(a\) plus some number of \(b\)'s. But the coefficient of \(b\) grows faster than any multiple of \(n\). The relationship between \(n\) and the coefficient of \(b\) is summarized as follows:

<table>
<thead>
<tr>
<th>Value of (n)</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coefficient of (b)</td>
<td>2</td>
<td>8</td>
<td>24</td>
<td>64</td>
</tr>
<tr>
<td>Ratio</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
The ratio is the coefficient of \( b \) divided by the value of \( n \). Thus, it appears that the coefficient of \( b \) is \( n \) times another factor that grows by 1 each time \( n \) doubles. In particular, we can see that this ratio is \( \log_2 n \), because \( \log_2 2 = 1 \), \( \log_2 4 = 2 \), \( \log_2 8 = 3 \), and \( \log_2 16 = 4 \). It is thus reasonable to conjecture that the solution to our recurrence relation is \( T(n) = an + bn \log_2 n \), at least for \( n \) a power of 2. We shall see that this formula is correct.

To get a solution to the recurrence, let us follow the same strategy as for previous examples. We write the inductive rule with argument \( m \), as

\[
T(m) = 2T(m/2) + bm, \quad \text{for } m \text{ a power of } 2 \quad (3.5)
\]

We then start with \( T(n) \) and use (3.5) to replace \( T(n) \) by an expression involving smaller values of the argument; in this case, the replacing expression involves \( T(n/2) \). That is, we begin with

\[
T(n) = 2T(n/2) + bn \quad (3.6)
\]

Next, we use (3.5), with \( n/2 \) in place of \( m \), to get a replacement for \( T(n/2) \) in (3.6). That is, (3.5) says that \( T(n/2) = 2T(n/4) + bn \), and we can replace (3.6) by

\[
T(n) = 2(2T(n/4) + bn/2) + bn = 4T(n/4) + 2bn
\]

Then, we can replace \( T(n/4) \) by \( 2T(n/8) + bn/4 \); the justification is (3.5) with \( n/4 \) in place of \( m \). That gives us

\[
T(n) = 4(2T(n/8) + bn/4) + 2bn = 8T(n/8) + 3bn
\]

The statement that we shall prove by induction on \( i \) is

\textbf{STATEMENT} \( S(i) \): If \( 1 \leq i \leq \log_2 n \), then \( T(n) = 2^i T(n/2^i) + ibn \).

\textbf{BASIS}. For \( i = 1 \), the statement \( S(1) \) says that \( T(n) = 2T(n/2) + bn \). This equality is the inductive rule in the definition of \( T(n) \), the running time of merge and sort, so we know that the basis holds.

\textbf{INDUCTION}. As in similar inductions where the inductive hypothesis is of the if-then form, the inductive step holds whenever \( i \) is outside the assumed range; here, \( i \geq \log_2 n \) is the simple case, where \( S(i + 1) \) is seen to hold.

For the hard part, suppose that \( i < \log_2 n \). Also, assume the inductive hypothesis \( S(i) \); that is, \( T(n) = 2^i T(n/2^i) + ibn \). Substitute \( n/2^i \) for \( m \) in (3.5) to get

\[
T(n/2^i) = 2T(n/2^{i+1}) + bn/2^i \quad (3.7)
\]

Substitute the right side of (3.7) for \( T(n/2^i) \) in \( S(i) \) to get

\[
T(n) = 2^i (2T(n/2^{i+1}) + bn/2^i) + ibn
= 2^i T(n/2^{i+1}) + bn + ibn
= 2^{i+1} T(n/2^{i+1}) + (i + 1)bn
\]

The last equality is the statement \( S(i + 1) \), and so we have proved the inductive step.
We conclude that the equality $S(i)$ — that is, $T(n) = 2^i T(n/2^i) + ibn$ — holds for any $i$ between 1 and $\log_2 n$. Now consider the formula $S(\log_2 n)$, that is,

$$T(n) = 2^{\log_2 n} T(n/2^{\log_2 n}) + (\log_2 n)bn$$

We know that $2^{\log_2 n} = n$ (recall that the definition of $\log_2 n$ is the power to which we must raise 2 to equal $n$). Also, $n/2^{\log_2 n} = 1$. Thus, $S(\log_2 n)$ can be written

$$T(n) = T(1) + bn \log_2 n$$

We also know that $T(1) = a$, by the basis of the definition of $T$. Thus,

$$T(n) = an + bn \log_2 n$$

After this analysis, we must replace the constants $a$ and $b$ by big-oh expressions. That is, $T(n)$ is $O(n) + O(n \log n)$.

Since $n$ grows more slowly than $n \log n$, we may neglect the $O(n)$ term and say that $T(n)$ is $O(n \log n)$. That is, merge sort is an $O(n \log n)$-time algorithm. Remember that selection sort was shown to take $O(n^2)$ time. Although strictly speaking, $O(n^2)$ is only an upper bound, it is in fact the tightest simple bound for selection sort. Thus, we can be sure that, as $n$ gets large, merge sort will always run faster than selection sort. In practice, merge sort is faster than selection sort for $n$’s larger than a few dozen.

**EXERCISES**

3.10.1: Draw structure trees for the functions

a) split

b) MergeSort

Indicate the running time for each node of the trees.

3.10.2*: Define a function $k$-mergesort that splits a list into $k$ pieces, sorts each piece, and then merges the result.

a) What is the running time of $k$-mergesort as a function of $k$ and $n$?

b)** What value of $k$ gives the fastest algorithm (as a function of $n$)? This problem requires that you estimate the running times sufficiently precisely that you can distinguish constant factors. Since you cannot be that precise in practice, for the reasons we discussed at the beginning of the chapter, you really need to examine how the running time from (a) varies with $k$ and get an approximate minimum.

3.11 Solving Recurrence Relations

There are many techniques for solving recurrence relations. In this section, we shall discuss two such approaches. The first, which we have already seen, is repeatedly substituting the inductive rule into itself until we get a relationship between $T(n)$ and $T(1)$ or — if 1 is not the basis — between $T(n)$ and $T(i)$ for some $i$ that is covered by the basis. The second method we introduce is guessing a solution and checking its correctness by substituting into the basis and the inductive rules.

---

8 Remember that inside a big-oh expression, we do not have to specify the base of a logarithm, because logarithms to all bases are the same, to within a constant factor.
In the previous two sections, we have solved exactly for $T(n)$. However, since $T(n)$ is really a big-oh upper bound on the exact running time, it is sufficient to find a tight upper bound on $T(n)$. Thus, especially for the “guess-and-check” approach, we require only that the solution be an upper bound on the true solution to the recurrence.

**Solving Recurrences by Repeated Substitution**

Probably the simplest form of recurrence that we encounter in practice is that of Example 3.24:

**BASIS.** $T(1) = a$.

**INDUCTION.** $T(n) = T(n - 1) + b$, for $n > 1$.

We can generalize this form slightly if we allow the addition of some function $g(n)$ in place of the constant $b$ in the induction. We can write this form as

**BASIS.** $T(1) = a$.

**INDUCTION.** $T(n) = T(n - 1) + g(n)$, for $n > 1$.

This form arises whenever we have a recursive function that takes time $g(n)$ and then calls itself with an argument one smaller than the argument with which the current function call was made. Examples are the factorial function of Example 3.24, the function `merge` of Section 3.10, and the recursive selection sort of Section 2.7. In the first two of these functions, $g(n)$ is a constant, and in the third it is linear in $n$. The function `split` of Section 3.10 is almost of this form; it calls itself with an argument that is smaller by 2. We shall see that this difference is unimportant.

Let us solve this recurrence by repeated substitution. As in Example 3.24, we first write the inductive rule with the argument $m$, as

$$T(m) = T(m - 1) + g(m)$$

and then proceed to substitute for $T$ repeatedly in the right side of the original inductive rule. Doing this, we get the sequence of expressions

$$T(n) = T(n - 1) + g(n)
= T(n - 2) + g(n - 1) + g(n)
= T(n - 3) + g(n - 2) + g(n - 1) + g(n)
\vdots
= T(n - i) + g(n - i + 1) + g(n - i + 2) + \cdots + g(n - 1) + g(n)$$

Using the technique in Example 3.24, we can prove by induction on $i$, for $i = 1, 2, \ldots, n - 1$, that

$$T(n) = T(n - i) + \sum_{j=0}^{i-1} g(n - j)$$
We want to pick a value for $i$ so that $T(n - i)$ is covered by the basis; thus, we pick $i = n - 1$. Since $T(1) = a$, we have $T(n) = a + \sum_{j=0}^{n-2} g(n - j)$. Put another way, $T(n)$ is the constant $a$ plus the sum of all the values of $g$ from 2 to $n$, or $a + g(2) + g(3) + \cdots + g(n)$. Unless all the $g(j)$’s are 0, the $a$ term will not matter when we convert this expression to a big-oh expression, and so we generally just need the sum of the $g(j)$’s.

**Example 3.25.** Consider the recursive selection sort function of Fig. 2.22, the body of which we reproduce as Fig. 3.29. If we let $T(m)$ be the running time of the function SelectionSort when given an array of $m$ elements to sort (that is, when the value of its argument $i$ is $n - m$), we can develop a recurrence relation for $T(m)$ as follows. First, the basis case is $m = 1$. Here, only line (1) is executed, taking $O(1)$ time.

```plaintext
(1) if (i < n-1) {
(2)   small = i;
(3)   for (j = i+1; j < n; j++)
(4)     if (A[j] < A[small])
(5)       small = j;
(6)   temp = A[small];
(8)   A[i] = temp;
(9)   recSS(A, i+1, n);
}
```

Fig. 3.29. Recursive selection sort.

For the inductive case, $m > 1$, we execute the test of line (1) and the assignments of lines (2), (6), (7), and (8), all of which take $O(1)$ time. The for-loop of lines (3) to (5) takes $O(n - i)$ time, or $O(m)$ time, as we discussed in connection with the iterative selection sort program in Example 3.17. To review why, note that the body, lines (4) and (5), takes $O(1)$ time, and we go $m - 1$ times around the loop. Thus, the time of the for-loop dominates lines (1) through (8), and we can write $T(m)$, the time of the entire function, as $T(m - 1) + O(m)$. The second term, $O(m)$, covers lines (1) through (8), and the $T(m - 1)$ term is the time for line (9), the recursive call. If we replace the hidden constant factors in the big-oh expressions by concrete constants, we get the recurrence relation

**BASIS.** $T(1) = a$.

**INDUCTION.** $T(m) = T(m - 1) + bm$, for $m > 1$.

This recurrence is of the form we studied, with $g(m) = bm$. That is, the solution is
\[ T(m) = a + \sum_{j=0}^{m-2} b(m-j) \]
\[ = a + 2b + 3b + \cdots + mb \]
\[ = a + b(m-1)(m+2)/2 \]

Thus, \( T(m) \) is \( O(m^2) \). Since we are interested in the running time of function \texttt{SelectionSort} on the entire array of length \( n \), that is, when called with \( i = 1 \), we need the expression for \( T(n) \) and find that it is \( O(n^2) \). Thus, the recursive version of selection sort is quadratic, just like the iterative version.

Another common form of recurrence generalizes the recurrence we derived for \texttt{MergeSort} in the previous section:

**Basis.** \( T(1) = a \).

**Induction.** \( T(n) = 2T(n/2) + g(n) \), for \( n \) a power of 2 and greater than 1.

This is the recurrence for a recursive algorithm that solves a problem of size \( n \) by subdividing it into two subproblems, each of size \( n/2 \). Here \( g(n) \) is the amount of time taken to create the subproblems and combine the solutions. For example, \texttt{MergeSort} divides a problem of size \( n \) into two problems of size \( n/2 \). The function \( g(n) \) is \( bn \) for some constant \( b \), since the time taken by \texttt{MergeSort} exclusive of recursive calls to itself is \( O(n) \), principally for the \texttt{split} and \texttt{merge} algorithms.

To solve this recurrence, we substitute for \( T \) on the right side. Here we assume that \( n = 2^k \) for some \( k \). The recursive equation can be written with \( m \) as its argument: \( T(m) = 2T(m/2) + g(m) \). If we substitute \( n/2^i \) for \( m \), we get

\[ T(n/2^i) = 2T(n/2^{i+1}) + g(n/2^i) \quad (3.8) \]

If we start with the inductive rule and proceed to substitute for \( T \) using (3.8) with progressively greater values of \( i \), we find

\[ T(n) = 2T(n/2) + g(n) \]
\[ = 2(2T(n/2^2) + g(n/2)) + g(n) \]
\[ = 2^2T(n/2^2) + 2g(n/2) + g(n) \]
\[ = 2^2(2T(n/2^3) + g(n/2^2)) + 2g(n/2) + g(n) \]
\[ = 2^3T(n/2^3) + 2^2g(n/2^2) + 2g(n/2) + g(n) \]
\[ \cdots \]
\[ = 2^iT(n/2^i) + \sum_{j=0}^{i-1} 2^jg(n/2^j) \]

If \( n = 2^k \), we know that \( T(n/2^k) = T(1) = a \). Thus, when \( i = k \), that is, when \( i = \log_2 n \), we obtain the solution

\[ T(n) = an + \sum_{j=0}^{(\log_2 n)-1} 2^jg(n/2^j) \quad (3.9) \]
to our recurrence.

Intuitively, the first term of (3.9) represents the contribution of the basis value \( a \). That is, there are \( n \) calls to the recursive function with an argument of size 1. The summation is the contribution of the recursion, and it represents the work performed by all the calls with argument size greater than 1.

Figure 3.30 suggests the accumulation of time during the execution of MergeSort. It represents the time to sort eight elements. The first row represents the work on the outermost call, involving all eight elements; the second row represents the work of the two calls with four elements each; and the third row is the four calls with two elements each. Finally, the bottom row represents the eight calls to MergeSort with a list of length one. In general, if there are \( n \) elements in the original unsorted list, there will be \( \log_2 n \) levels at which \( bn \) work is done by calls to MergeSort that result in other calls. The accumulated time of all these calls is thus \( bn \log_2 n \). There will be one level at which calls are made that do not result in further calls, and \( an \) is the total time spent by these calls. Note that the first \( \log_2 n \) levels represent the terms of the summation in (3.9) and the lowest level represents the term \( an \).

<table>
<thead>
<tr>
<th></th>
<th>( bn )</th>
<th>( bn )</th>
<th>( bn )</th>
<th>( an )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3.30. Time spent by the calls to mergesort.

\[ \begin{align*}
T(n) & = an + \sum_{j=0}^{\log_2 n-1} 2^j bn / 2^j \\
& = an + bn \sum_{j=0}^{\log_2 n-1} 1 \\
& = an + bn \log n
\end{align*} \]

The last equality comes from the fact there are \( \log_2 n \) terms in the sum and each term is 1. Thus, when \( g(n) \) is linear, the solution to (3.9) is \( O(n \log n) \).

**Example 3.26.** In the case of MergeSort, the function \( g(n) \) is \( bn \) for some constant \( b \). The solution (3.9) with these parameters is therefore

**Solving Recurrences by Guessing a Solution**

Another useful approach to solving recurrences is to guess a solution \( f(n) \) and then use the recurrence to show that \( T(n) \leq f(n) \). That may not give the exact value of \( T(n) \), but if it gives a tight upper bound, we are satisfied. Often we guess only the functional form of \( f(n) \), leaving some parameters unspecified; for example, we may guess that \( f(n) = an^b \), for some \( a \) and \( b \). The values of the parameters will be forced, as we try to prove \( T(n) \leq f(n) \) for all \( n \).
Although we may consider it bizarre to imagine that we can guess solutions accurately, we can frequently deduce the high-order term by looking at the values of $T(n)$ for a few small values of $n$. We then throw in some lower-order terms as well, and see whether their coefficients turn out to be nonzero.\footnote{If it is any comfort, the theory of differential equations, which in many ways resembles the theory of recurrence equations, also relies on known solutions to equations of a few common forms and then educated guessing to solve other equations.}

\begin{itemize}
  \item \textbf{Example 3.27.} Let us again examine the \texttt{MergeSort} recurrence relation from the previous section, which we wrote as

  \textbf{Basis.} $T(1) = a$.

  \textbf{Induction.} $T(n) = 2T(n/2) + bn$, for $n$ a power of 2 and greater than 1.

  We are going to guess that an upper bound to $T(n)$ is $f(n) = cn \log_2 n + d$ for some constants $c$ and $d$. Recall that this form is not exactly right; in the previous example we derived the solution and saw that it had an $O(n \log n)$ term with an $O(n)$ term, rather than with a constant term. However, this guess turns out to be good enough to prove an $O(n \log n)$ upper bound on $T(n)$.

  We shall use complete induction on $n$ to prove the following, for some constants $c$ and $d$:

  \textbf{Statement} $S(n)$: If $n$ is a power of 2 and $n \geq 1$, then $T(n) \leq f(n)$, where $f(n)$ is the function $cn \log_2 n + d$.

  \textbf{Basis.} When $n = 1$, $T(1) \leq f(1)$ provided $a \leq d$. The reason is that the $cn \log_2 n$ term of $f(n)$ is 0 when $n = 1$, so that $f(1) = d$, and it is given that $T(1) = a$.

  \textbf{Induction.} Let us assume $S(i)$ for all $i < n$, and prove $S(n)$ for some $n > 1$.\footnote{In most complete inductions we assume $S(i)$ for $i$ up to $n$ and prove $S(n + 1)$. In this case it is notationally simpler to prove $S(n)$ from $S(i)$, for $i < n$, which amounts to the same thing.}

  If $n$ is not a power of 2, there is nothing to prove, since the if-portion of the if-then statement $S(n)$ is not true. Thus, consider the hard case, in which $n$ is a power of 2. We may assume $S(n/2)$, that is,

  $$T(n/2) \leq (cn/2) \log_2 (n/2) + d$$

  because it is part of the inductive hypothesis. For the inductive step, we need to show that

  $$T(n) \leq f(n) = cn \log_2 n + d$$

  When $n \geq 2$, the inductive part of the definition of $T(n)$ tells us that

  $$T(n) \leq 2T(n/2) + bn$$

  Using the inductive hypothesis to bound $T(n/2)$, we have

  $$T(n) \leq 2[c(n/2) \log_2 (n/2) + d] + bn$$

\end{itemize}
Manipulating Inequalities

In Example 3.27 we derive one inequality, \( T(n) \leq cn \log_2 n + d \), from another,
\[
T(n) \leq cn \log_2 n + (b - c)n + 2d
\]
Our method was to find an “excess” and requiring it to be at most 0. The general principle is that if we have an inequality \( A \leq B + E \), and if we want to show that \( A \leq B \), then it is sufficient to show that \( E \leq 0 \). In Example 3.27, \( A \) is \( T(n) \), \( B \) is \( cn \log_2 n + d \), and \( E \), the excess, is \( (b - c)n + d \).

Since \( \log_2 (n/2) = \log_2 n - \log_2 2 = \log_2 n - 1 \), we may simplify this expression to
\[
T(n) \leq cn \log_2 n + (b - c)n + 2d
\]
(3.10)

We now show that \( T(n) \leq cn \log_2 n + d \), provided that the excess over \( cn \log_2 n + d \) on the right side of (3.10) is at most 0; that is, \( (b - c)n + d \leq 0 \). Since \( n > 1 \), this inequality is true when \( d \geq 0 \) and \( b - c \leq -d \).

We now have three constraints for \( f(n) = cn \log_2 n + d \) to be an upper bound on \( T(n) \):
1. The constraint \( a \leq d \) comes from the basis.
2. \( d \geq 0 \) comes from the induction, but since we know that \( a > 0 \), this inequality is superseded by (1).
3. \( b - c \leq -d \), or \( c \geq b + d \), also comes from the induction.

These constraints are obviously satisfied if we let \( d = a \) and \( c = a + b \). We have now shown by induction on \( n \) that for all \( n \geq 1 \) and a power of 2,
\[
T(n) \leq (a + b)n \log_2 n + a
\]
This argument shows that \( T(n) \) is \( O(n \log n) \), that is, that \( T(n) \) does not grow any faster than \( n \log n \). However, the bound \( (a + b)n \log_2 n + a \) that we obtained is slightly larger than the exact answer that we obtained in Example 3.26, which was \( bn \log_2 n + an \). At least we were successful in obtaining a bound. Had we taken the simpler guess of \( f(n) = cn \log_2 n \), we would have failed, because there is no value of \( c \) that can make \( f(1) \geq a \). The reason is that \( c \times 1 \times \log_2 1 = 0 \), so that \( f(1) = 0 \). If \( a > 0 \), we evidently cannot make \( f(1) \geq a \). ✦

Example 3.28. Now let us consider a recurrence relation that we shall encounter later in the book:

BASIS. \( G(1) = 3 \).

INDUCTION. \( G(n) = (2^{n/2} + 1)G(n/2) \), for \( n > 1 \).

This recurrence has actual numbers, like 3, instead of symbolic constants like \( a \). In Chapter 13, we shall use recurrences such as this to count the number of gates in a circuit, and gates can be counted exactly, without needing the big-oh notation to
Summary of Solutions

In the table below, we list the solutions to some of the most common recurrence relations, including some we have not covered in this section. In each case, we assume that the basis equation is \( T(1) = a \) and that \( k \geq 0 \).

<table>
<thead>
<tr>
<th>Inductive Equation</th>
<th>( T(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T(n) = T(n-1) + bn^k )</td>
<td>( O(n^{k+1}) )</td>
</tr>
<tr>
<td>( T(n) = cT(n-1) + bn^k ), for ( c &gt; 1 )</td>
<td>( O(c^n) )</td>
</tr>
<tr>
<td>( T(n) = cT(n/d) + bn^k ), for ( c &gt; d^k )</td>
<td>( O(n^{\log_d c}) )</td>
</tr>
<tr>
<td>( T(n) = cT(n/d) + bn^k ), for ( c &lt; d^k )</td>
<td>( O(n^k) )</td>
</tr>
<tr>
<td>( T(n) = cT(n/d) + bn^k ), for ( c = d^k )</td>
<td>( O(n^k \log n) )</td>
</tr>
</tbody>
</table>

All the above also hold if \( bn^k \) is replaced by any \( k \)th-degree polynomial.

hide unknowable constant factors.

If we think about a solution by repeated substitution, we might see that we are going to make \( \log_2 n - 1 \) substitutions before \( G(n) \) is expressed in terms of \( G(1) \). As we make the substitutions, we generate factors

\[
(2^{n/2} + 1)(2^{n/4} + 1)(2^{n/8} + 1) \cdots (2^1 + 1)
\]

If we neglect the “+1” term in each factor, we have approximately the product

\[
2^{n/2}2^{n/4}2^{n/8} \cdots 2^1,
\]

which is

\[
2^{n/2+n/4+n/8+\cdots+1}
\]

or \( 2^{n-1} \) if we sum the geometric series in the exponent. That is half of \( 2^n \), and so we might guess that \( 2^n \) is a term in the solution \( G(n) \). However, if we guess that \( f(n) = c2^n \) is an upper bound on \( G(n) \), we shall fail, as the reader may check. That is, we get two inequalities involving \( c \) that have no solution.

We shall thus guess the next simplest form, \( f(n) = c2^n + d \), and here we shall be successful. That is, we can prove the following statement by complete induction on \( n \) for some constants \( c \) and \( d \):

**STATEMENT** \( S(n) \): If \( n \) is a power of 2 and \( n \geq 1 \), then \( G(n) \leq c2^n + d \).

**BASIS.** If \( n = 1 \), then we must show that \( G(1) \leq c2^1 + d \), that is, that \( 3 \leq 2c + d \). This inequality becomes one of the constraints on \( c \) and \( d \).

**INDUCTION.** As in Example 3.27, the only hard part occurs when \( n \) is a power of 2 and we want to prove \( S(n) \) from \( S(n/2) \). The equation in this case is

\[
G(n/2) \leq c2^{n/2} + d
\]

We must prove \( S(n) \), which is \( G(n) \leq c2^n + d \). We start with the inductive definition of \( G \),

\[
G(n) = (2^{n/2} + 1)G(n/2)
\]
and then substitute our upper bound for $G(n/2)$, converting this expression to

$$G(n) \leq (2^{n/2} + 1)(c2^{n/2} + d)$$

Simplifying, we get

$$G(n) \leq c2^n + (c + d)2^{n/2} + d$$

That will give the desired upper bound, $c2^n + d$, on $G(n)$, provided that the excess on the right, $(c + d)2^{n/2}$, is no more than 0. It is thus sufficient that $c + d \leq 0$.

We need to select $c$ and $d$ to satisfy the two inequalities

1. $2c + d \geq 3$, from the basis, and
2. $c + d \leq 0$, from the induction.

For example, these inequalities are satisfied if $c = 3$ and $d = -3$. Then we know that $G(n) \leq 3(2^n - 1)$. Thus, $G(n)$ grows exponentially with $n$. It happens that this function is the exact solution, that is, that $G(n) = 3(2^n - 1)$, as the reader may prove by induction on $n$. ✪

**EXERCISES**

3.11.1: Let $T(n)$ be defined by the recurrence

$$T(n) = T(n-1) + g(n), \text{ for } n > 1$$

Prove by induction on $i$ that if $1 \leq i < n$, then

$$T(n) = T(n-i) + \sum_{j=0}^{i-1} g(n-j)$$

3.11.2: Suppose we have a recurrence of the form

$$T(1) = a$$
$$T(n) = T(n-1) + g(n), \text{ for } n > 1$$

Give tight big-oh upper bounds on the solution if $g(n)$ is

a) $n^2$
b) $n^2 + 3n$
c) $n^{3/2}$
d) $n \log n$
e) $2^n$

3.11.3: Suppose we have a recurrence of the form

$$T(1) = a$$
$$T(n) = T(n/2) + g(n), \text{ for } n \text{ a power of } 2 \text{ and } n > 1$$

Give tight big-oh upper bounds on the solution if $g(n)$ is

a) $n^2$
b) $2n$
c) 10
d) $n \log n$
e) $2^n$
3.11.4*: Guess each of the following as the solution to the recurrence
\[ T(1) = a \]
\[ T(n) = 2T(n/2) + bn, \text{ for } n \text{ a power of } 2 \text{ and } n > 1 \]
\[ a) \ cn \log_2 n + dn + e \\
\[ b) \ cn + d \\
\[ c) \ cn^2 \]

What constraints on the unknown constants \( c, d, \) and \( e \) are implied? For which of these forms does there exist an upper bound on \( T(n) \)?

3.11.5: Show that if we guess \( G(n) \leq c2^n \) for the recurrence of Example 3.28, we fail to find a solution.

3.11.6*: Show that if \( T(1) = a \)
\[ T(n) = T(n - 1) + n^k, \text{ for } n > 1 \]
then \( T(n) \) is \( O(n^{k+1}) \). You may assume \( k \geq 0 \). Also, show that this is the tightest simple big-oh upper bound, that is, that \( T(n) \) is not \( O(n^m) \) if \( m < k + 1 \). \text{Hint: Expand } T(n) \text{ in terms of } T(n-i), \text{ for } i = 1, 2, \ldots, \text{ to get the upper bound. For the lower bound, show that } T(n) \text{ is at least } cn^{k+1} \text{ for some particular } c > 0.

3.11.7**: Show that if \( T(1) = a \)
\[ T(n) = cT(n/d) + bn^k, \text{ for } n \text{ a power of } d \]
Iteratively expand \( T(n) \) in terms of \( T(n/d^i) \) for \( i = 1, 2, \ldots \). Show that
\[ a) \text{ If } c > d^k, \text{ then } T(n) \text{ is } O(n^{k \log_d c}) \\
\[ b) \text{ If } c = d^k, \text{ then } T(n) \text{ is } O(n^k \log n) \\
\[ c) \text{ If } c < d^k, \text{ then } T(n) \text{ is } O(n^k) \]

3.11.8**: Consider the recurrence
\[ T(1) = a \]
\[ T(n) = cT(n/d) + bn^k, \text{ for } n \text{ a power of } d \]
You may use the solutions of Exercise 3.11.6.

3.11.9: Solve the following recurrences, each of which has \( T(1) = a \):
\[ a) \text{ T(n) = } 3T(n/2) + n^2, \text{ for } n \text{ a power of } 2 \text{ and } n > 1 \\
\[ b) \text{ T(n) = } 10T(n/3) + n^2, \text{ for } n \text{ a power of } 3 \text{ and } n > 1 \\
\[ c) \text{ T(n) = } 16T(n/4) + n^2, \text{ for } n \text{ a power of } 4 \text{ and } n > 1 \]

3.11.10: Solve the recurrence
\[ T(1) = 1 \]
\[ T(n) = 3^nT(n/2), \text{ for } n \text{ a power of } 2 \text{ and } n > 1 \]

3.11.11: The Fibonacci recurrence is \( F(0) = F(1) = 1 \), and
The running time of programs

\[ F(n) = F(n - 1) + F(n - 2), \text{ for } n > 1 \]

The values \( F(0), F(1), F(2), \ldots \) form the sequence of Fibonacci numbers, in which each number after the first two is the sum of the two previous numbers. (See Exercise 3.9.4.) Let \( r = (1 + \sqrt{5})/2 \). This constant \( r \) is called the golden ratio and its value is about 1.62. Show that \( F(n) \) is \( O(r^n) \). Hint: For the induction, it helps to guess that \( F(n) \leq ar^n \) for some \( n \), and attempt to prove that inequality by induction on \( n \). The basis must incorporate the two values \( n = 0 \) and \( n = 1 \). In the inductive step, it helps to notice that \( r \) satisfies the equation \( r^2 = r + 1 \).

\section*{3.12 Summary of Chapter 3}

Here are the important concepts covered in Chapter 3.

\begin{itemize}
  \item Many factors go into the selection of an algorithm for a program, but simplicity, ease of implementation, and efficiency often dominate.
  \item Big-oh expressions provide a convenient notation for upper bounds on the running times of programs.
  \item There are recursive rules for evaluating the running time of the various compound statements of C, such as for-loops and conditions, in terms of the running times of their constituent parts.
  \item We can evaluate the running time of a function by drawing a structure tree that represents the nesting structure of statements and evaluating the running time of the various pieces in a bottom-up order.
  \item Recurrence relations are a natural way to model the running time of recursive programs.
  \item We can solve recurrence relations either by iterated substitution or by guessing a solution and checking our guess is correct.
\end{itemize}

Divide and conquer is an important algorithm-design technique in which a problem is partitioned into subproblems, the solutions to which can be combined to provide a solution to the whole problem. A few rules of thumb can be used to evaluate the running time of the resulting algorithm: A function that takes time \( O(1) \) and then calls itself on a subproblem of size \( n - 1 \) takes time \( O(n) \). Examples are the factorial function and the merge function.

\begin{itemize}
  \item More generally, a function that takes time \( O(n^k) \) and then calls itself on a subproblem of size \( n - 1 \) takes time \( O(n^{k+1}) \).
  \item If a function calls itself twice but the recursion goes on for \( \log_2 n \) levels (as in merge sort), then the total running time is \( O(n \log n) \) times the work done at each call, plus \( O(n) \) times the work done at the basis. In the case of merge sort, the work at each call, including basis calls, is \( O(1) \), so the total running time is \( O(n \log n) + O(n) \), or just \( O(n \log n) \).
\end{itemize}
If a function calls itself twice and the recursion goes on for \( n \) levels (as in the Fibonacci program of Exercise 3.9.4), then the running time is exponential in \( n \).

3.13 Bibliographic Notes for Chapter 3

The study of the running time of programs and the computational complexity of problems was pioneered by Hartmanis and Stearns [1964]. Knuth [1968] was the book that established the study of the running time of algorithms as an essential ingredient in computer science.

Since that time, a rich theory for the difficulty of problems has been developed. Many of the key ideas are found in Aho, Hopcroft, and Ullman [1974, 1983].

In this chapter, we have concentrated on upper bounds for the running times of programs. Knuth [1976] describes analogous notations for lower bounds and exact bounds on running times.

For more discussion of divide and conquer as an algorithm-design technique, see Aho, Hopcroft, and Ullman [1974] or Borodin and Munro [1975]. Additional information on techniques for solving recurrence relations can be found in Graham, Knuth, and Patashnik [1989].


