Chapter 3

Analysis of Algorithms

Software analysis differs from software design by the inputs and outputs of each process. With software analysis, the program and the input are given, and production of the output is determined. With software design, the input and the desired output are given, and production of the program is determined.

This book presents object-oriented (OO) software design by example. That is, it does not describe the process by which the algorithms in the dp4ds distribution were produced. The design tradeoffs in the dp4ds distribution favor the teaching of OO design patterns as classified in the Gamma book[1] at the expense of simplicity of the implementations.

Regardless of the tradeoffs in the design process, once an algorithm is written its quality should be assessed. Two important quality characteristics of an algorithm are its correctness, *i.e.* whether it conforms to its specification, and its efficiency, *i.e.* how fast it executes. This chapter provides tools for quantifying algorithm efficiency and for determining program correctness.

3.1 Iterative Algorithms

A program produces output by executing a sequence of statements. The time required for the program to produce its output is therefore directly related to the number of statements it executes. You cannot simply count the number of statements in a program listing to determine its efficiency because algorithms typically contain `if` statements and `while` loops.

The effect of an `if` statement is to omit the execution of a block of code depending on its test of a boolean condition. Thus, it is possible for a block of code inside an `if` statement to not execute even though it exists in a program listing. The effect of a loop is to repeat the execution of a block of code until the test of its boolean condition causes the loop to terminate. Thus, it is possible for a block of code inside a loop to execute many times even though it appears only once in a program listing.

The first step in determining the efficiency of an algorithm is to perform a statement execution count. The total number of statements that execute generally depends on the

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

size of the problem. The size can depend on one of two factors. One possibility is for the algorithm to process a set of data, in which case the size of the problem is equal to the number of data items n to be processed. For example, in a sorting algorithm n would be the number of values to sort. Another possibility is for the algorithm to compute the value of some function, in which case the size of the problem is equal to some parameter of the function. For example, in an algorithm to compute the nth Fibonacci number n would be the size of the problem.

Equational notation

This text uses a slightly unusual mathematical and logical notation that is defined in the text by Gries and Schneider.[2] The system unifies the quantification notation to be consistent between mathematics and logic. It uses a linear "equational" style and is especially suited for proving computation theorems and program correctness. From this point on, the text introduces equational notation as it is used.

The following sections use summation notation to represent statement execution counts. For example, the standard way to write the sum of the squares of the first n positive integers is

$$\sum_{i=1}^{n} i^2 .$$

The equational notation for the same expression is

$$(\Sigma\, i \mid 1 \leq i \leq n : i^2) .$$

All such quantifications have explicit scope for the dummy variable denoted by the outer parentheses. Within the parentheses the quantification consists of three parts:

- the operator and dummy variable,
- the range, which is a boolean expression, and
- the body, which is an expression that is type compatible with the operator.

A vertical bar separates the operator and dummy variable from the range, and a colon separates the range from the body. In the above example, the operation is addition, the dummy variable is i, the range is $1 \leq i \leq n$, and the body is i^2. Because Σ represents the addition operation, the above expression could just as accurately be written

$$(+i \mid 1 \leq i \leq n : i^2) .$$

Another equational notation convention is the use of a dot as the function application operator, which separates a function name from its argument. Hence, the function $g(x)$ is written in equational notation as $g.x$. The precedence of the function application operator is higher than all math and logic operators, so that $g.x+3$ is interpreted as $(g.x)+3$ and not $g.(x+3)$. If the latter case is intended, it is customary to omit the function application operator and write $g(x+3)$ as in standard math notation.

[2] David Gries and Fred B. Schneider, *A Logical Approach to Discrete Math*, Springer-Verlag, 1994.

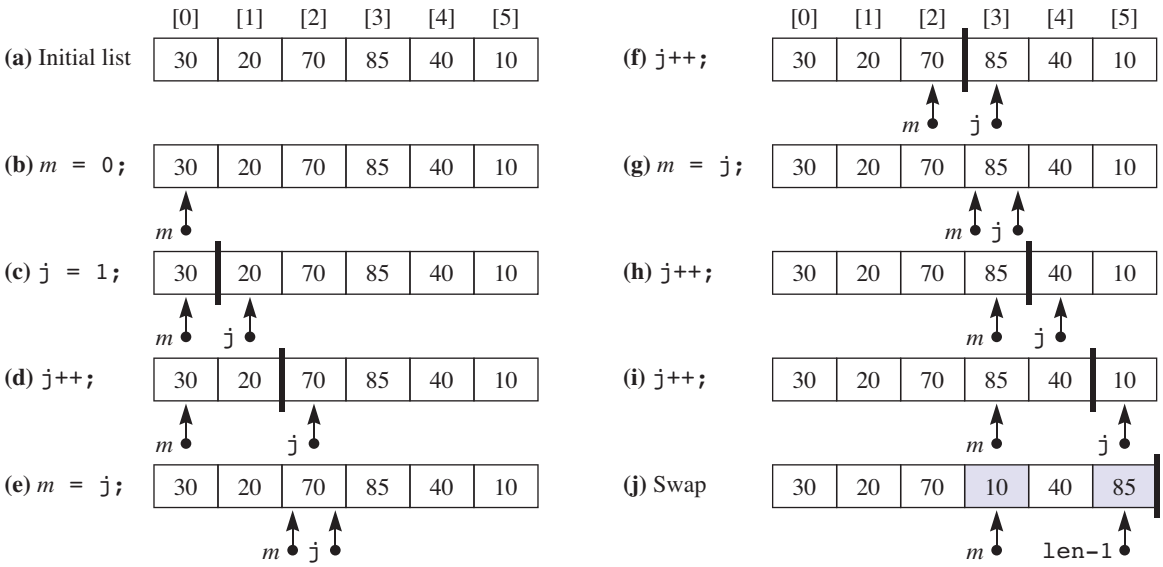


Figure 3.1 A trace of the execution of function `largestLast()`. The symbol *m* is the variable `indexOfMax` in the program listing.

Loops

Following is the program listing of an algorithm that takes a safe array containing a list of items of type `T`, determines the index in the list of the largest item, and switches the item with the last element of the list. Parameter `len` is the length of the list in the array. The executable statements are on lines 3, 4, 5, 6, 9, 10, and 11. None of the other lines contain executable code.

```
1  template<class T>
2  void largestLast(ASeq<T> &a, int len) {
3      int indexOfMax = 0;
4      for (int j = 1; j < len; j++) {
5          if (a[indexOfMax] < a[j]) {
6              indexOfMax = j;
7          }
8      }
9      T temp = a[len - 1];
10     a[len - 1] = a[indexOfMax];
11     a[indexOfMax] = temp;
12 }
```

Figure 3.1 is a trace of the execution for a list of length six. To conserve space in the figure, the symbol *m* is the variable `indexOfMax` in the function. The description Swap in part (j) of the figure represents the code

Line	`len` = 2	`len` = 3	`len` = n
3	1	1	1
4	2	3	n
5	1	2	$n-1$
6	0	0	0
9	1	1	1
10	1	1	1
11	1	1	1
Total:	7	9	$2n+3$

(a) Best case.

Line	`len` = 2	`len` = 3	`len` = n
3	1	1	1
4	2	3	n
5	1	2	$n-1$
6	1	2	$n-1$
9	1	1	1
10	1	1	1
11	1	1	1
Total:	8	11	$3n+2$

(b) Worst case.

Figure 3.2 Statement execution count for `largestLast()`.

```
T temp = a[len - 1];
a[len - 1] = a[indexOfMax];
a[indexOfMax] = temp;
```

which swaps `a[indexOfMax]` with the last element of the list. In this example, `len` has value 6 for the length of the list, `len-1` has value 5, and `a[5]` is the last element of the list.

You can see from the trace that line 5 in the body of the loop executes five times — once each for values of `j` at 1, 2, 3, 4, and 5. On the other hand, line 6, which is also in the body of the loop only executes two times, because it is an alternative of the `if` statement and only executes when the boolean guard is true. There are a range of statement execution counts for this algorithm depending on how many times line 6 executes. With algorithms like this, common practice is to analyze the two extreme cases:

- Best case — Line 6 never executes.
- Worst case — Line 6 always executes.

It is sometimes possible to analyze the average case as well, which would be the statement execution count averaged over a random distribution of list values. The average count will always fall somewhere between the two extreme cases.

Figure 3.2 shows the statement execution count for `largestLast()` for both cases when `len` has values 2, 3, and n in general. Note the difference between the `for` loop of line 4 and line 5 in its body. Every time the test in line 4 is true, line 5 executes. However, the count for line 4 is always one greater than the count for line 5, because the boolean test in line line 4 must execute one last time to trigger termination of the loop.

The statement execution count for `largestLast()` in both cases is linear. The worst case function $f.n = 3n+2$ is a straight line on a plot of $f.n$ versus n. Linear

	[0]	[1]	[2]	[3]	[4]	[5]
(a) Initial list	30	20	70	85	40	10
(b) `k = 6;` Swap 85 and 10	30	20	70	10	40	85
(c) `k = 5;` Swap 70 and 40	30	20	40	10	70	85
(d) `k = 4;` Swap 40 and 10	30	20	10	40	70	85
(e) `k = 3;` Swap 30 and 10	10	20	30	40	70	85
(f) `k = 2;` Swap 20 and 20	10	20	30	40	70	85

Figure 3.3 A trace of the execution of function `selectionSort()`.

behavior is typical for algorithms with loops that are not nested. The next section shows that doubly-nested loops frequently have quadratic execution count functions.

Nested loops

Following is an algorithm for selection sort, which contains a nested loop. The inner loop is identical to the code for `largestLast()` with variable `k` taking the place of `len`. Figure 3.3 is a trace of `selectionSort()` for a list of length six. Figure 3.3(b) shows the effect of one execution of the outer loop and corresponds to the complete execution of the inner loop shown in Figure 3.2(a)–(j).

```
1  template<class T>
2  void selectionSort(ASeq<T> &a, int len) {
3      for (int k = len; k > 1; k--) {
4          int indexOfMax = 0;
5          for (int j = 1; j < k; j++) {
6              if (a[indexOfMax] < a[j]) {
7                  indexOfMax = j;
8              }
9          }
10         T temp = a[k - 1];
11         a[k - 1] = a[indexOfMax];
12         a[indexOfMax] = temp;
13     }
14 }
```

Figure 3.4 shows the best case and worse case execution counts for selection sort. When `len` equals 2, line 5 executes twice. In this case, `k` equals 2, and the inner loop executes once with a value of 1 for `j`. When `len` equals 3, line 5 executes twice when `k` equals 2 and three times when `k` equals 3. When `len` equals 4 (not shown in the figure), line 5 executes $2+3+4$ times which occurs when `k` has values 2, 3, and 4. You can see that in the general case for `len` equals n, line 5 executes $2+3+\cdots+n$ times. For a given value of `k`, line 5 executes one more time than 6 because of the extra test that terminates the inner loop.

Line	`len` = 2	`len` = 3	`len` = n
3	2	3	n
4	1	2	$n-1$
5	2	$2+3$	$2+3+\cdots+n$
6	1	$1+2$	$1+2+\cdots+n-1$
7	0	0	0
10	1	2	$n-1$
11	1	2	$n-1$
12	1	2	$n-1$
Total:	9	19	n^2+5n-5

(a) Best case.

Line	`len` = 2	`len` = 3	`len` = n
3	2	3	n
4	1	2	$n-1$
5	2	$2+3$	$2+3+\cdots+n$
6	1	$1+2$	$1+2+\cdots+n-1$
7	1	$1+2$	$1+2+\cdots+n-1$
10	1	2	$n-1$
11	1	2	$n-1$
12	1	2	$n-1$
Total:	10	22	$\frac{3}{2}n^2+\frac{9}{2}n-5$

(b) Worst case.

Figure 3.4 Statement execution count for `selectionSort()`.

To compute the total execution count in the general case for `len` equals n, you must compute the summations $2+3+\cdots+n$ for line 5 and $1+2+\cdots+(n-1)$ for line 6. These summations are computed from this formula for the sum of the first n positive integers.

$$(\Sigma\, i \mid 1 \le i \le n : i) = 1+2+3+\cdots+n = n(n+1)/2 \quad \text{for } n \ge 0.$$

Using this formula,

$$2+3+\cdots+n = n(n+1)/2-1 = \frac{1}{2}n^2+\frac{1}{2}n-1$$

and

$$1+2+\cdots+(n-1) = (n-1)n/2 = \frac{1}{2}n^2-\frac{1}{2}n.$$

The net result is that $f.n = n^2+5n-5$ in the best case and $f.n = \frac{3}{2}n^2+\frac{9}{2}n-5$ in the worst case. It is typical that singly nested loops are linear, doubly nested loops are quadratic, triply nested loops are cubic, and so on.

Applications

Statement execution counts are only the first step in the process of calculating the execution time. To get the total time of execution you would need to multiply

$$\text{Total time} = \text{Number of statements executed} \times \frac{\text{Time}}{\text{Statement}}.$$

But this expression is not accurate because it assumes that every statement takes the same time to execute, which is definitely not the case. A more accurate approach would be to determine, for each statement, how many central processing unit (CPU) cycles it takes to execute and combine that with the GHz rating of your CPU.

For example, consider the best case total from Figure 3.2, $2n+3$. It is computed from the sum

$$1+n+(n-1)+1+1+1$$

for executable statements on lines 3, 4, 5, 9, 10, and 11. If you can determine that the statement on line 3 takes c_3 cycles, the statement on line 4 takes c_4 cycles, and so on, then the total number of cycles that executes is

$$1c_3+nc_4+(n-1)c_5+1c_9+1c_{10}+1c_{11}\,.$$

You would then multiply this times the time per cycle determined by the GHz rating of your CPU. Suppose your CPU runs at 3GHz, which is 3×10^9 cycles/second. The total execution time in seconds would then be

$$\text{Total time} = (1c_3+nc_4+(n-1)c_5+1c_9+1c_{10}+1c_{11})\text{ cycles}\times\frac{\text{seconds}}{3\times10^9\text{cycles}}\,.$$

In general,

$$\text{Total time} = (\Sigma\, i \mid \text{Line } i \text{ is an executable statement} : e_ic_i)\times\frac{1}{\text{GHz rating}}$$

where e_i is the statement execution count for line i, and c_i is the number of CPU cycles it takes to execute the statement on line i.

This refinement in the estimate of the execution time still has two shortcomings. The first is that the operating system typically does not schedule a job to run from start to finish without interruption. Instead, your job may execute partially and then be interrupted by some unpredictable event, only to be resumed and completed at a later time. Because of operating system scheduling, the elapsed time is typically greater than the predicted execution time.

The second is that most computers now have more than one core processor that can execute statements concurrently. It might happen that the compiler for your program is able to detect different parts of your program that can execute at the same time on more than one core. Because of concurrent processing with multiple cores, the elapsed time can be less than the predicted execution time.

In spite of these shortcomings, using only the statement execution count to assess performance is still valuable, first as a rough estimate, and more importantly in an asymptotic analysis.

One use of statement execution counts is to measure the time it takes to execute an algorithm with a given set of data and from that measurement predict how long it would take to execute with a larger set of data. For example, suppose you run the selection sort for a hundred values and it takes $40\mu\text{s} = 40\times10^{-6}$ seconds. To calculate how long it would take to sort two hundred values, use the fact that in the best case $f.n = n^2+5n-5$, and set up the ratio

$$\frac{T}{(200)^2+5(200)-5} = \frac{40\times10^{-6}}{(100)^2+5(100)-5}$$

where T is the time to sort two hundred values. Solving for T yields a value of

$$T = \frac{(200)^2 + 5(200) - 5}{(100)^2 + 5(100) - 5}(40 \times 10^{-6}) = \frac{40995}{10495}(40 \times 10^{-6})$$
$$= 3.906(40 \times 10^{-6}) = 156 \times 10^{-6} = 156\mu\text{s}.$$

So, doubling the number of data values to sort does not double the execution time. It multiplies it by a factor of 3.9, because the statement execution count $f.n$ is not linear.

3.2 Asymptotic Analysis

The computation to estimate the execution time in the previous section is probably not a realistic scenario because of the small number of data values. Most software users are more interested in how their programs handle large problems, usually much more than one or two hundred data values. Asymptotic analysis evaluates algorithm efficiency for large values of n.

For example, suppose for a different processor that you measure 70ms to process a million values, and you want to estimate the execution time to process two million values. Using the same analysis as before, the computation is

$$T = \frac{(2 \times 10^6)^2 + 5(2 \times 10^6) - 5}{(10^6)^2 + 5(10^6) - 5}(70 \times 10^{-3}) = \frac{4000009999995}{1000004999995}(70 \times 10^{-3})$$
$$\approx 4(70 \times 10^{-3}) = 280 \times 10^{-3} = 280\text{ms}.$$

This example shows that the increase when you double the number of values is very nearly a factor of four. Furthermore, the computation with $n = 10^6$ is more closely an approximation to four than is the computation with $n = 100$. In general, the greater the value of n the better the approximation. To determine the approximate factor for large n, you can simply use the highest-order polynomial term and ignore the lower-order terms.

$$T = \frac{(2n)^2}{n^2}(\text{time for } n) = 4(\text{time for } n).$$

Asymptotic bounds

An asymptotic bound formalizes the idea that you can neglect the lower-order terms in a mathematical expression for the execution time. Here is the formal definition of the upper bound for a given function of the execution time $f.n$.

Definition of asymptotic upper bound: For a given function $g.n$, $O(g.n)$, pronounced "big-oh of g of n", is the set of functions

$$\{f.n \mid (\exists c, n_0 \mid c > 0 \wedge n_0 > 0 : (\forall n \mid n \geq n_0 : 0 \leq f.n \leq c \cdot g.n))\}.$$

***O*-notation:** $f.n = O(g.n)$ means function $f.n$ is in the set $O(g.n)$.

In English, $f.n = O(g.n)$ if and only if there exist positive constants c and n_0 such that

$$f.n \leq c \cdot g.n \quad \text{for all } n \geq n_0.$$

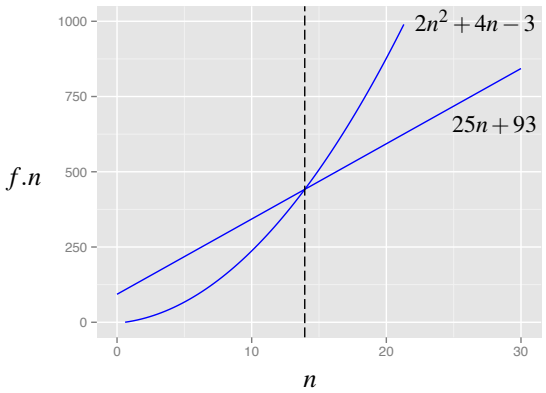


Figure 3.5 Statement execution counts for two algorithms. The crossover point indicated by the vertical dashed line is at $n = 13.94$.

In practice, $f.n$ is the expression for the execution time of an algorithm as a function of the size of the problem n, and $g.n$ is the leading term that predominates in determining the execution time of the algorithm. For example, suppose you compute a statement execution count for two algorithms as follows:

- Algorithm A: $f.n = 25n + 93$
- Algorithm B: $f.n = 2n^2 + 4n - 3$

The question is, Which algorithm is better, A or B? Figure 3.5 shows a plot of both functions. Even though the coefficients in Algorithm A (25 and 93) are greater than the coefficients for B, and even though Algorithm B takes less time to execute than A for small values of n, for all values of n greater than the crossover point Algorithm A is faster than B. Because performance for large problems is more important than performance for small problems, we conclude that Algorithm A is better.

For Algorithm A, the predominate term is n. Following is a proof from the definition that $25n + 93 = O(n)$, that is, $25n + 93$ is bounded above by n.

Proof: Must prove that there exist positive constants c, n_0 such that

$25n + 93 \leq cn \quad$ for $n \geq n_0$.

$$
\begin{aligned}
& 25n + 93 \\
\leq \quad & \langle \text{Replacing 93 with a larger value, provided } n \geq 93 \rangle \\
& 25n + n \\
= \quad & \langle \text{Math} \rangle \\
& 26n \\
= \quad & \langle \text{Provided } c = 26 \rangle \\
& cn
\end{aligned}
$$

So, $c = 26, n_0 = 93$. ∎

The first step in the proof replaces 93 with n. As long as n is at least 93, $25n + 93$ will be at most $25n + n$. The proviso, Provided $n \geq 93$, in the justification for the first step establishes the existence of a specific value for n_0. Similarly, the proviso in the last step of the proof establishes the existence of a specific value for c.

For Algorithm B, the predominate term is n^2. Following is a proof from the definition that $2n^2+4n-3 = O(n^2)$, that is, $2n^2+4n-3$ is bounded above by n^2.

Proof: Must prove that there exist positive constants c, n_0 such that
$2n^2+4n-3 \le cn^2 \quad$ for $n \ge n_0$.

$$
\begin{array}{ll}
 & 2n^2+4n-3 \\
\le & \langle \text{Eliminating a negative value} \rangle \\
 & 2n^2+4n \\
\le & \langle \text{Replacing } 4n \text{ with a larger value, provided } n \ge 1 \rangle \\
 & 2n^2+4n\cdot n \\
= & \langle \text{Math} \rangle \\
 & 6n^2 \\
= & \langle \text{Provided } c = 6 \rangle \\
 & cn^2
\end{array}
$$

So, $c = 6, n_0 = 1$. ∎

The proof establishes n_0 by the proviso in the second step, which replaces $4n$ with $4n \cdot n$. If n were less than 1, you would be replacing $4n$ with a smaller value instead of a larger one.

The proof that an execution count is bounded *above* by a predominant term demonstrates that the algorithm cannot be *worse* than the bound. On the other side, the proof that an execution count is bounded *below* by a predominant term demonstrates that the algorithm cannot be *better* than the bound. Here is the formal definition of the lower bound for a given function of the execution time $f.n$.

Definition of asymptotic lower bound: For a given function $g.n$, $\Omega(g.n)$, pronounced "big-omega of g of n", is the set of functions

$$\{f.n \mid (\exists c, n_0 \mid c > 0 \wedge n_0 > 0 : (\forall n \mid n \ge n_0 : 0 \le c \cdot g.n \le f.n))\}.$$

Ω-notation: $f.n = \Omega(g.n)$ means function $f.n$ is in the set $\Omega(g.n)$.

In English, $f.n = \Omega(g.n)$ if and only if there exist positive constants c and n_0 such that

$f.n \ge c \cdot g.n \quad$ for all $n \le n_0$.

For Algorithm A, it turns out that not only is the execution count bounded above by n, it is also bounded below by n. Following is a proof from the definition that $25n+93 = \Omega(n)$.

Proof: Must prove that there exist positive constants c, n_0 such that
$25n+93 \ge cn \quad$ for $n \ge n_0$.

$$
\begin{array}{ll}
 & 25n+93 \\
\ge & \langle \text{Eliminating a positive value} \rangle \\
 & 25n \\
= & \langle \text{Provided } c = 25 \rangle \\
 & cn
\end{array}
$$

So, $c = 25$, and any positive n_0 is possible, say $n_0 = 1$. ∎

None of the steps in this proof had restrictions on the value of n. Because n represents the size of the problem and is therefore positive, the convention is to arbitrarily choose $n_0 = 1$ in such proofs.

In the same way that n is both an asymptotic upper and lower bound of $25n+93$, n^2 is both an asymptotic upper and lower bound of $2n^2+4n-3$. Here is a proof that n^2 is a lower bound, that is, $2n^2+4n-3 = \Omega(n^2)$.

Proof: Must prove that there exist positive constants c, n_0 such that
$2n^2+4n-3 \geq cn^2 \quad$ for $n \geq n_0$.

$$
\begin{array}{ll}
 & 2n^2+4n-3 \\
\geq & \langle \text{Increasing a negative value, provided } n \geq 1 \rangle \\
 & 2n^2+4n-3n \\
= & \langle \text{Math} \rangle \\
 & 2n^2+n \\
\geq & \langle \text{Eliminating a positive value, provided } n \geq 0 \rangle \\
 & 2n^2 \\
= & \langle \text{Provided } c = 2 \rangle \\
 & cn^2
\end{array}
$$

So, $c = 2, n_0 = 1$. ∎

The above proof has two provisos for n. The proviso in the first step requires $n \geq 1$, establishing a value of 1 for n_0. The proviso in the third step requires $n \geq 0$, establishing a value of 0 for n_0. When there is more than one proviso establishing different values for n_0, logic dictates that you choose the *stronger* condition. In this example, $n \geq 1$ is stronger than $n \geq 0$ because $n \geq 1$ implies $n \geq 0$ and not the other way around. In other words, if $n \geq 1$ then $n \geq 0$ and both provisos are satisfied. That reasoning justifies the value of 1 for n_0 as opposed to 0.

The general strategy for proving bounds is to transform the expression, reasoning with inequalities, to produce a constant times the predominant term. When proving an upper bound O you can eliminate negative values and/or increase positive values. When proving an lower bound Ω you can eliminate positive values and/or increase negative values. Usually these techniques are sufficient to produce a constant times the predominant term.

One situation can arise when proving lower bounds that requires an additional technique. For example, suppose you want to prove that $6n^3 - 7n = \Omega(n^3)$ where the predominant term is followed by a lower order term that has a negative coefficient. You cannot eliminate the $-7n$ in one of the steps of the proof, because it is a negative value. That is, you cannot claim that $6n^3 - 7n \geq 6n^3$. Furthermore, you could increase a negative value and claim that $6n^3 - 7n \geq 6n^3 - 7n^3$. But then after a math step, you would have $c = -1$, and c must be a positive constant.

However, you *can* claim that $6n^3 - 7n \geq 5n^3$ and then compute within the proof the positive value of n_0 that makes it so. In this example, the coefficient of the highest order term $6n^3$ is 6. So, pick one less than 6, which is 5, for the coefficient of the term $5n^3$ that is less than the original expression.

Proof: Must prove that there exist positive constants c, n_0 such that
$6n^3 - 7n \geq cn^3 \quad$ for $n \geq n_0$.

$$
\begin{array}{ll}
 & 6n^3 - 7n \\
\geq & \quad \langle 6n^3 - 7n \geq 5n^3,\ n^3 \geq 7n,\ n^2 \geq 7,\ n \geq \sqrt{7}\ \rangle \\
 & 5n^3 \\
= & \quad \langle \text{Provided } c = 5 \rangle \\
 & cn^3
\end{array}
$$

So, $c = 5, n_0 = \sqrt{7}$. ∎

The algebraic steps are carried out in the justification of the first step of the proof. Note that one of the algebraic steps requires dividing both sides of the inequality by n without changing the inequality. The step is justified because n is guaranteed to be positive.

Constants c and n_0 must be positive. Constant n_0 is typically an integer and sometimes the statement execution count function is only valid for values of n that are greater than or equal to 1. However, the above technique may require that the value of c be less than 1, which can happen when you prove a lower bound with the predominant term having a coefficient of 1.

For example, suppose you want to prove that $n^3 - 5n = \Omega(n^3)$. The dilemma is that you must prove $n^3 - 5n \geq cn^3$ for some constant c. As before, you cannot claim that $n^3 - 5n \geq n^3$. However, you *can* claim that $n^3 - 5n \geq 0.5n^3$ and then compute within the proof the value of n_0 that makes it so.

Proof: Must prove that there exist positive constants c, n_0 such that
$n^3 - 5n \geq cn^3 \quad$ for $n \geq n_0$.

$$
\begin{array}{ll}
 & n^3 - 5n \\
\geq & \quad \langle n^3 - 5n \geq 0.5n^3,\ 0.5n^3 \geq 5n,\ n^2 \geq 10,\ n \geq \sqrt{10}\ \rangle \\
 & 0.5n^3 \\
= & \quad \langle \text{Provided } c = 0.5 \rangle \\
 & cn^3
\end{array}
$$

So, $c = 0.5, n_0 = \sqrt{10}$. ∎

You might ask where the 0.5 came from. Indeed, it is an arbitrary value less than 1. You could pick a different value less than 1, and you would simply get different a value for n_0. But, the proof would still be valid, as you only need to prove the existence of one pair of positive constants c and n_0.

When the same term is both an asymptotic upper bound and lower bound of some function, it is called an asymptotic tight bound. All the algorithms in this book have well known asymptotic tight bounds. Here is the formal definition.

Definition of asymptotic tight bound: For a given function $g.n$, $\Theta(g.n)$, pronounced "big-theta of g of n", is the set of functions

$$
\begin{array}{l}
\{f.n \mid (\exists c_1, c_2, n_0 \mid c_1 > 0 \wedge c_2 > 0 \wedge n_0 > 0 : \\
\qquad (\forall n \mid n \geq n_0 : 0 \ \leq \ c_1 \cdot g.n \ \leq \ f.n \ \leq \ c_2 \cdot g.n)\ )\}.
\end{array}
$$

Θ-notation: $f.n = \Theta(g.n)$ means function $f.n$ is in the set $\Theta(g.n)$.

In English, $f.n = \Theta(g.n)$ if and only if there exist positive constants c_1, c_2, and n_0 such that

$c_1 \cdot g.n \leq f.n \leq c_2 \cdot g.n \quad$ for all $n \geq n_0$.

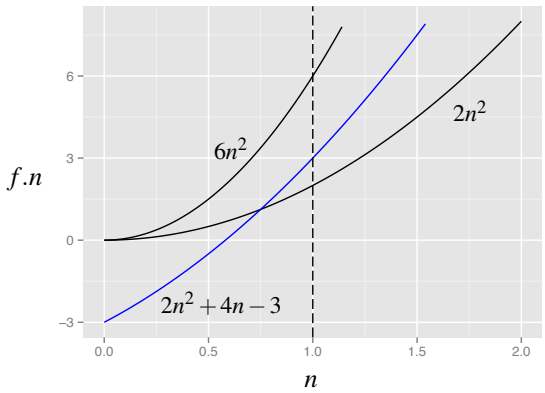


Figure 3.6 Upper and lower bounds for the function $f.n = 2n^2 + 4n - 3$, which is the statement execution count for Algroithm B. The vertical dashed line is at $n_0 = 1$.

The definition shows that Θ is a combination of Ω and O. Whereas Ω requires $c \cdot g.n \leq f.n$ for the lower bound, and O requires $f.n \leq c \cdot g.n$ for the upper bound, Θ requires both. The constants c_1 and c_2 are different for the two bounds, but there is only one constant n_0.

In practice, it is easier to prove asymptotic tight bounds using the following theorem.

Theorem: $f.n = \Theta(g.n)$ if and only if $f.n = O(g.n)$ and $f.n = \Omega(g.n)$.

It is easy to see how proving Ω and O is a proof of Θ. When you prove Ω you establish constants c_1 and n_1, and when you prove O you establish constants c_2 and n_2. The definition of Θ is satisfied if you take n_0 to be the larger of n_1 and n_2. If n is greater than the larger it must also be greater than the smaller and both criteria are satisfied.

Examples of asymptotic tight bounds are functions for Algorithms A and B, as the above proofs demonstrate. Specifically for Algorithm A, because $25n + 93 = O(n)$ and $25n + 93 = \Omega(n)$, it follows that $25n + 93 = \Theta(n)$. For Algorithm B, because $2n^2 + 4n - 3 = O(n^2)$ and $2n^2 + 4n - 3 = \Omega(n^2)$, it follows that $2n^2 + 4n - 3 = \Theta(n^2)$.

Figure 3.6 is a graphical interpretation of the tight bound of the execution function for Algorithm B. For the lower bound, the proof established $c = 2$, $n_0 = 1$, and for the upper bound it established $c = 6$, $n_0 = 1$. In this example, n_0 has the same value for the two cases and is shown by the vertical dashed line. You can see from the plot that as long as n is greater than 1, $2n^2 \leq 2n^2 + 4n - 3 \leq 6n^2$.

It is possible to apply the proof techniques for specific polynomial functions to a general polynomial with a positive coefficient in its highest term. The result is the following theorem.

Polynomial bound: Define an *asymptotically positive polynomial p.n of degree d* to be

$$p.n = (\Sigma\, i \mid 0 \leq i \leq d : a_i n^i)$$

where the constants $a_0, a_1, \ldots, a_d$ are the *coefficients* of the polynomial and $a_d > 0$. Then $p.n = \Theta(n^d)$.

This theorem tells you directly that the polynomial $843n^4 + 59n^3 - 923.6n^2 + 1000$ is

$\Theta(n^4)$. Proof of the theorem is based on the strategy for proving bounds O and Ω but applied to the coefficients a_i in general.

3.3 Recursive Algorithms

Statement execution counts for algorithms with doubly-nested loops are based on the formula for the sum of the first n integers.

$$(\Sigma\, i \mid 1 \le i \le n : i) = 1+2+3+\cdots+n = n(n+1)/2 \quad \text{for } n \ge 0.$$

Statement execution counts for algorithms with triply-nested loops, such as the matrix multiplication algorithm, are based on the formula for the sum of the squares of the first n integers.

$$(\Sigma\, i \mid 1 \le i \le n : i^2) = 1^2+2^2+3^2+\cdots+n^2 = n(n+1)(2n+1)/6 \quad \text{for } n \ge 0.$$

To determine the statement execution count of an iterative program with loops is a relatively straightforward application of formulas like these.

A recursive algorithm achieves repetition in its computation not by a loop, but by the programming technique of calling itself. The question immediately arises,

- How do you determine the statement execution count for a recursive algorithm?

Unfortunately, statement execution counts are significantly more difficult to determine with recursive algorithms compared to iterative ones. A rich mathematical theory has been developed to answer this question. Although a complete answer is beyond the scope of this book, the following section is an introduction to this field of study.

Writing recurrences

Recursive algorithms are based on a strategy called *divide and conquer*. A recursive algorithm must begin with a test for the smallest possible problem, called the base case. If the test succeeds, the solution is computed with no recursive calls. If the test fails, called the inductive case, the problem is divided into one or more smaller subproblems. A recursive call provides the solution for the smaller problem. Some processing might be necessary before and/or after the recursive call. The algorithm repeatedly calls itself, each time with a smaller problem, until it reaches the smallest problem, which requires no further calls.

Here is the general outline of a divide-and-conquer algorithm.

```
divideAndConquer
    if (Smallest problem)
        Computation with no recursive calls
    else
        One or more recursive calls
        Possible additional computation
```

The problem is to determine how many statements execute with such an algorithm.

With iterative algorithms, you can relate mathematical formulas like the one for the sum of the first n integers directly to the code. With recursive algorithms the corresponding mathematical formulas are called *recurrences*. In the same way that a recursive algorithm calls itself, a mathematical recurrence is a function that is defined in terms of itself. A mathematical recurrence for the statement execution count of the above divide-and-conquer algorithm has the general form

$$T(n) = \begin{cases} \textit{Count for smallest problem} & \text{if smallest problem,} \\ \textit{Count for recursive calls} + & \\ \quad \textit{Count for additional computation} & \text{otherwise.} \end{cases}$$

where $T(n)$ represents the time to compute the problem with size n.

To determine the statement execution count is a two-step problem.

- Write down the recurrence from the recursive code for the algorithm.
- Solve the recurrence.

Unfortunately, writing the recurrence is only the first part of the problem, as the recurrence relation does not indicate directly the Θ of the algorithm.

Section 3.1 determines exact C++ statement execution counts for algorithms with loops. Section 3.2, however, defines asymptotic bounds that depend on the predominant term without regard to its specific coefficient. The analysis of recursive algorithms rarely takes into account exact statement execution counts, as the exact counts have no bearing on the asymptotic bounds.

Consider the recursive function to compute the factorial of integer n.

```
factorial(n)
   if n <= 0 then
      return 1
   else
      return n * factorial(n - 1)
```

The recurrence for this function is

$$T(n) = \begin{cases} 2 & \text{if } n \leq 0, \\ T(n-1)+3 & \text{if } n > 0. \end{cases}$$

The execution count is 2 for the base case, because the `if` statement executes followed by the `return` statement. The execution count is $T(n-1)+3$ for the inductive case because

- the `if` statement executes,
- the `factorial(n - 1)` statement executes,
- the multiplication by n executes, and
- the `return` statement executes.

The term 3 in the inductive case takes into account the additional computation other than the recursive call. The crucial part of the inductive case is the term $T(n-1)$. If $T(n)$ is the statement execution count required to compute $n!$, then $T(n-1)$ must be the statement execution count to compute $(n-1)!$.

Because the analysis is only concerned with asymptotic bounds, the recurrence is equivalent to

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

where the notation $\Theta(1)$ represents any function that has an asymptotic tight bound of 1, that is, any constant.

Consider the following recursive binary search algorithm that returns the index of value v if it is contained in sorted array a between $a[lo]$ and $a[hi]$, and -1 if it is not.

```
binarySearch(a, lo, hi, v)
   if lo > hi then
      return -1
   else
      mid = (lo + hi)/2
      if v < a[mid] then
         return binarySearch(a, lo, mid - 1, v)
      else if v > a[mid] then
         return binarySearch(a, mid + 1, hi, v)
      else
         return mid
```

The size of the problem is the number of elements in the sorted array. If there are n elements in the array indexed from 0 to $n-1$, the initial call to the function is

```
binarySearch(a, 0, n - 1, v)
```

with a value of 0 for lo and $n-1$ for hi.

Unlike the performance of the factorial function, the performance of this algorithm has a best case and a worst case. If you are lucky, the value searched will be at the midpoint of the array and its index will be returned with no recursive call. So, in the best case the execution time is $\Theta(1)$.

If v is not in the array the algorithm terminates by executing

```
return -1
```

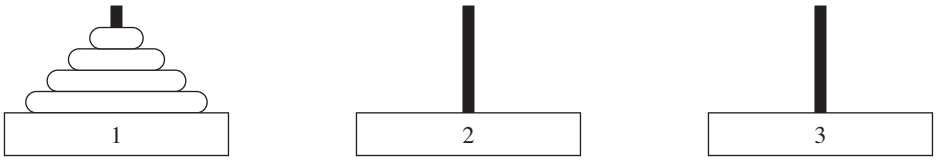
instead of

```
return mid
```

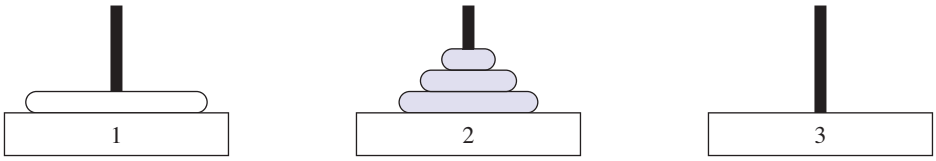
For each call, whether it discards the left half and searches the right half or discards the right half and searches the left half, it searches an array segment whose length is roughly half of the current segment. So, in the worst case the recurrence is

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

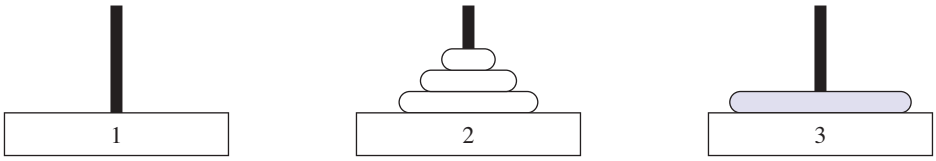
Sometimes a recursive algorithm makes more than one recursive call for the smaller subproblem. An example is the algorithm to print out the instructions for the classic towers of Hanoi puzzle. Figure 3.7 shows the puzzle with three pegs. The problem is to



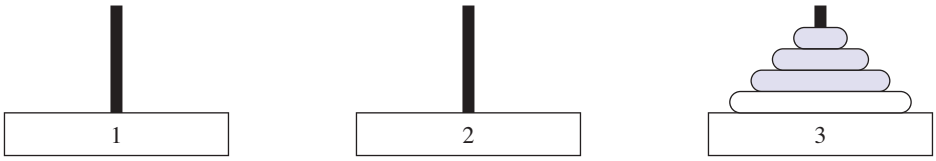
(a) n disks on peg 1.



(b) Move $n-1$ disks from peg 1 to peg 2.



(c) Move one disk from peg 1 to peg 3.



(d) Move $n-1$ disks from peg 2 to peg 3.

Figure 3.7 The towers of Hanoi puzzle. A requirement is to move the disks one at a time with no disk resting on a larger disk beneath it.

move the disks one at a time from peg 1 to peg 3 with no disk ever resting on a larger disk beneath it. The size of the problem n is the number of disks to move.

Suppose you want to move n disks from peg f to peg t where f is an integer variable that labels the "from" peg and t is a variable that labels the "to" peg. An expression for the intermediate peg, that is, the peg that is neither f nor t is $6-f-t$. For example, in Figure 3.7 f is 1, t is 3, and the intermediate peg is $6-f-t=6-1-3=2$.

The divide-and-conquer strategy for moving n disks is to divide the solution into three parts. The smaller subproblem is the solution for moving $n-1$ disks. The three parts of the solution to move n disks from peg f to peg t are

- move $n-1$ disks from peg f to the intermediate peg,
- move one disk from peg f to peg t, and
- move $n-1$ disks from the intermediate peg to peg t.

Here is the recursive algorithm.

```
towersOfHanoi(n, f, t)
```



```
if n == 1 then
   print "Move one disk from peg ", f, " to peg ", t, "."
else
   i = 6 - f - t
   towersOfHanoi(n - 1, f, i)
   towersOfHanoi(1, f, t)
   towersOfHanoi(n - 1, i, t)
```

The initial call to solve the problem in Figure 3.7(a) is

```
towersOfHanoi(4, 1, 3)
```

with a value of 4 for n, 1 for f, and 3 for t.

The recurrence expression for the base case is obviously $\Theta(1)$. But what is the recurrence for the inductive case? Even though the algorithm makes three recursive calls, the second call only contributes a constant count to the additional computation. The first call contributes $T(n-1)$ to the count and the third call also contributes $T(n-1)$. Because $T(n-1)+T(n-1)=2T(n-1)$, the recurrence is

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

The recurrence examples thus far have $\Theta(1)$ for the base case, which is true for virtually all important recursive algorithms in computer science. The coefficient in front of $T()$ in the inductive case is usually a small integer that indicates how many recursive calls the algorithm makes. In the vast majority of algorithms, the coefficient is either 1 or 2.

In the examples, the argument of T is either $n-1$, as in $2T(n-1)$ for towers of Hanoi, or $n/2$, as in $T(n/2)$ for the binary search. For the recursion to terminate the argument must be less than n, which is true for both $n-1$ and $n/2$. In the majority of algorithms the argument is one of these two expressions, although there are some important exceptions.

Every example thus far has $\Theta(1)$ for the additional computation in the inductive case. There are a number of important problems that have $\Theta(n)$ for the additional computation. The chapter on sorting shows some algorithms whose recurrences are

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

Solving recurrences

After writing the recurrence for the execution time from the algorithm, the next step is to solve the recurrence for $T(n)$ as a function of n. Four possible techniques are

- backward substitution,
- the recursion-tree method,
- guess and verify, and
- the master method.

Backward substitution is a generalization technique. You start with $T(n)$ and successively substitute the expression for the inductive case until you get to the base case. For example, the recurrence for the recursive factorial is

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

Because we are only interested in the asymptotic bounds, standard practice is to solve the simplest problem with specific constants for the Θ terms.

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

Here is the backward substitution.

$$\begin{aligned} T(n) &= T(n-1) + 1 \\ &= T(n-2) + 1 + 1 \\ &= T(n-3) + 1 + 1 + 1 \\ &\vdots \\ &= T(n-n) + \overbrace{1 + 1 + \cdots + 1}^{n \text{ occurrences}} \\ &= T(0) + n \\ &= 1 + n \end{aligned}$$

So, $T(n) = n + 1$, and therefore $T(n) = \Theta(n)$.

The penultimate step requires you to count how many substitutions are required to reach the base case. In the factorial problem, there are n substitutions, and so there are n occurrences of 1 in the summation. It is not always so easy to count the number of substitutions. Consider this recurrence for the worst case binary search.

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

When the inductive case has $n/2$ in the argument of T, it simplifies the analysis to assume that n is an exact power of 2, say $n = 2^k$. Here is the backward substitution.

$$\begin{aligned} T(n) &= T(n/2) + 1 && = T(n/2^1) + 1 \\ &= T(n/4) + 1 + 1 && = T(n/2^2) + 1 + 1 \\ &= T(n/8) + 1 + 1 + 1 && = T(n/2^3) + 1 + 1 + 1 \\ &\vdots \\ &= T(n/n) + 1 + 1 + \cdots + 1 && = T(n/2^k) + \overbrace{1 + 1 + \cdots + 1}^{k \text{ occurrences}} \\ &= T(1) + k \\ &= 1 + k \end{aligned}$$

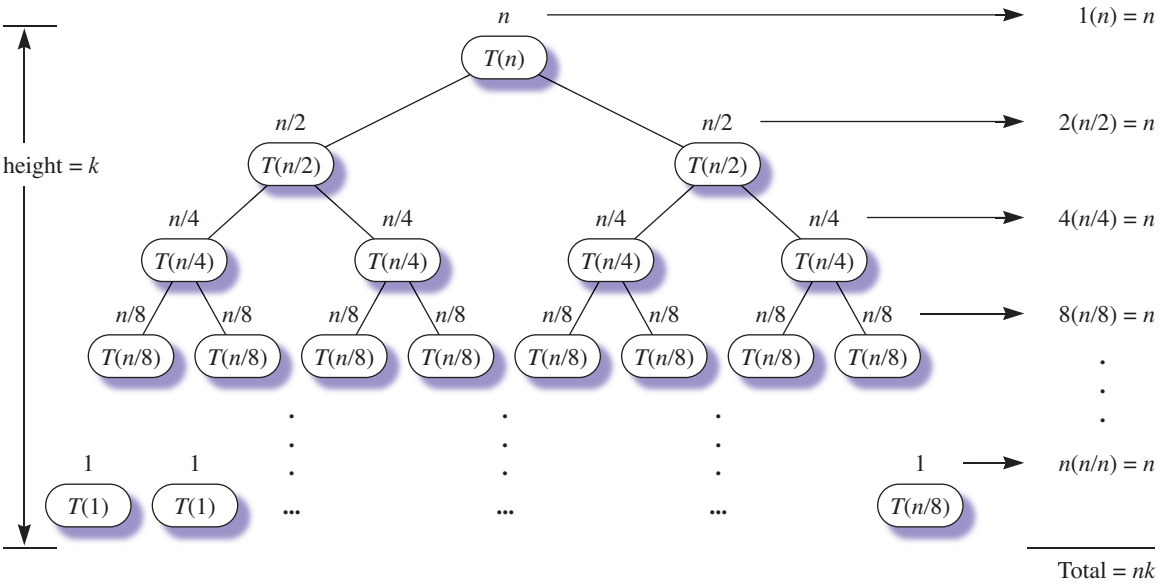


Figure 3.8 The recurrence tree for an inductive case of $T(n) = 2T(n/2) + n$.

Solving $n = 2^k$ for k yields $k = \lg n$ where $\lg n$ is the notation for $\log_2 n$. So, $T(n) = \lg n + 1$, and therefore $T(n) = \Theta(\lg n)$.

The recursion tree method is a way to visualize graphically the call tree and calculate the asymptotic bound. A recursion tree normally gives you the bound as opposed to an exact solution of the recursion. The technique is to write the call tree for the recursive computation. Inside each node, write the parameter of the recursive call. Next to each node, write an expression for how much work is done at that level. On the right side of the tree, total the work that is done at that level. Finally, total the work done at all levels to get the asymptotic bound.

As an example, consider the following recurrence assuming again for simplicity that n is an exact power of 2 with $n = 2^k$.

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

Figure 3.8 shows the recurrence tree. Because the inductive case has $2T(n/2)$ there are two recursive calls at each level, each call with a parameter of $n/2$. The work done at the top level is n because of the n in $2T(n/2) + n$. The inductive case at the second level is $T(n/2) = 2T(n/4) + n/2$. Hence, the work done for a call at that level is $n/2$. However, there are two calls to the second level, so the total work done at that level is $n/2 + n/2 = n$. Similarly, the total work done at each level is n. The grand total is n times the number of levels, which is $nk = n \lg n$. The asymptotic bound on the total time is, therefore, $T(n) = \Theta(n \lg n)$.

Even though it can be difficult to derive the solution to a recurrence, it is straightforward to prove that a given function is a solution to a given recurrence. Such proofs are

naturally proofs by mathematical induction with the base case of the proof corresponding to the base case of the recurrence and the inductive case of the proof corresponding to the inductive case of the recurrence. When using the guess and verify method the guess can come from the backward substitution method or the recursion-tree method.

Here is a proof that $T(n) = 2^n - 1$ is a solution to the recurrence

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

for the towers of Hanoi algorithm. In the inductive case, it is legal to use textual substitution in the recurrence because the recurrence is the definition of $T(n)$. It is illegal to use textual substitution in the proposed solution $T(n) = 2^n - 1$.

Proof: Base case.

$$\begin{aligned} & T(n) = 2^n - 1 \\ = & \quad \langle \text{Base case is } n = 1 \rangle \\ & T(1) = 2^1 - 1 \\ = & \quad \langle \text{Math} \rangle \\ & T(1) = 1 \\ = & \quad \langle \text{Definition of } T(n) \rangle \\ & 1 = 1 \end{aligned}$$

Inductive case. Must prove that $T(n+1) = 2^{n+1} - 1$ assuming $T(n) = 2^n - 1$ as the inductive hypothesis.

$$\begin{aligned} & T(n+1) \\ = & \quad \langle \text{Definition of } T(n) \text{ with } [n := n+1] \rangle \\ & 2T(n+1-1)+1 \\ = & \quad \langle \text{Math} \rangle \\ & 2T(n)+1 \\ = & \quad \langle \text{Inductive hypothesis} \rangle \\ & 2(2^n - 1)+1 \\ = & \quad \langle \text{Math} \rangle \\ & 2^{n+1} - 1 \quad \blacksquare \end{aligned}$$

So, $T(n) = 2^n - 1$, and therefore $T(n) = \Theta(2^n)$.

Another example of the guess and verify method is the proof that $T(n) = \lg n + 1$ is a solution to the recurrence

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

for the worst case binary search. To keep the analysis simple, the proof assumes that n is an exact power of 2. In contrast with the previous proof, which proved the inductive case for solution $T(n+1)$ assuming the solution for $T(n)$, this one proves the solution for $T(2n)$ assuming the solution for $T(n)$. Technically, the proof is only valid for those those values of n that are exact powers of 2. There is a mathematical justification for why

the resulting asymptotic bounds also hold for the other values of n, but its description is beyond the scope of this book.

Proof: Base case.

$$
\begin{aligned}
& T(n) = \lg n + 1 \\
= \quad & \langle \text{Base case is } n = 1 \rangle \\
& T(1) = \lg 1 + 1 \\
= \quad & \langle \text{Math} \rangle \\
& T(1) = 1 \\
= \quad & \langle \text{Definition of } T(n) \rangle \\
& 1 = 1
\end{aligned}
$$

Inductive case. Must prove that $T(2n) = \lg(2n) + 1$ assuming $T(n) = \lg n + 1$ as the inductive hypothesis.

$$
\begin{aligned}
& T(2n) \\
= \quad & \langle \text{Definition of } T(n) \text{ with } [n := 2n] \rangle \\
& T(2n/2) + 1 \\
= \quad & \langle \text{Math} \rangle \\
& T(n) + 1 \\
= \quad & \langle \text{Inductive hypothesis} \rangle \\
& \lg n + 1 + 1 \\
= \quad & \langle \text{Math} \rangle \\
& \lg n + \lg 2 + 1 \\
= \quad & \langle \text{Math}, \lg a + \lg b = \lg(ab) \rangle \\
& \lg(2n) + 1 \quad \blacksquare
\end{aligned}
$$

So, $T(n) = \lg n + 1$, and therefore $T(n) = \Theta(\lg n)$.

The master method is a cookbook method based on the master theorem. There are several versions of the master theorem. The following version is the simplest one to understand and use.

Master theorem: Let a be an integer $a \geq 1$, b be a real number $b > 1$, and c be a real number $c > 0$. Given a recurrence of the form

$$
T(n) = \begin{cases} 1 & \text{if } n = 1, \\ aT(n/b) + n^c & \text{if } n > 1. \end{cases}
$$

then, for n an exact power of b,

- if $\log_b a < c$, $T(n) = \Theta(n^c)$,
- if $\log_b a = c$, $T(n) = \Theta(n^c \log n)$, and
- if $\log_b a > c$, $T(n) = \Theta(n^{\log_b a})$.

A proof of the master theorem is based on analyzing the recurrence tree. In general, the height of the tree is $\log_b n$ and there are a^i nodes at each level, because the number of nodes at each level is a times the number of nodes at the previous level. In the first case, the work per level decreases with increasing depth, and the effect of the term for the

additional computation n^c predominates. In the third case, the work per level increases with increasing depth, and the effect of the high multiplicity of recursive calls $aT(n/b)$ predominates.

More advanced versions of the master theorem apply to cases where the additional computation in the inductive case is $f.n$ in general instead of being restricted to the form n^c. All versions, however, have n/b for the argument of T in the inductive case. So, the master theorem does not apply to recurrences with $n-1$ for the argument.

An example of the master method is with

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

whose recurrence tree is shown in Figure 3.8. From the recurrence, identify $a = 2, b = 2$, and $c = 1$. Then compute $\log_b a = \log_2 2 = 1$. Because $c = 1$, we have the second case of the master theorem. Therefore, $T(n) = \Theta(n^c \log n) = \Theta(n^1 \log n) = \Theta(n \lg n)$. (The last step comes from the fact that $\log_a n$ and $\log_b n$ differ by a constant factor for any positive constants a and b.)

Two more examples of the master method are for the recurrence

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

which has $T(n) = \Theta(n^2)$, and

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

which has $T(n) = \Theta(n^{\log_2 2}) = \Theta(n)$.

Properties of asymptotic bounds

Recursive algorithms frequently have asymptotic bounds that contain logarithmic functions. Proofs of such asymptotic bounds rely on the same techniques as those in the proofs of polynomial bounds. Figure 3.9 is a plot of the three functions $f.n = 1$, $f.n = \lg n$, and $f.n = n$. Here are some mathematical facts from the figure that are useful in proofs of bounds that contain logarithmic functions.

- If $n \geq 1$, then $\lg n \geq 0$.
- If $n \geq 2$, then $\lg n \geq 1$.
- For all positive values of n, $n > \lg n$.

Asymptotically, $\lg n$ is situated between 1 and n. That is, $\lg n = \Omega(1)$, and $\lg n = O(n)$.

An example of how to use these facts is the proof that $f.n = 3n^2 + 2n \lg n = \Theta(n^2)$. Here is the proof that $f.n = 3n^2 + 2n \lg n = O(n^2)$.

Proof: Must prove that there exist positive constants c, n_0 such that

$f.n = 3n^2 + 2n \lg n \leq cn^2 \quad$ for $n \geq n_0$.

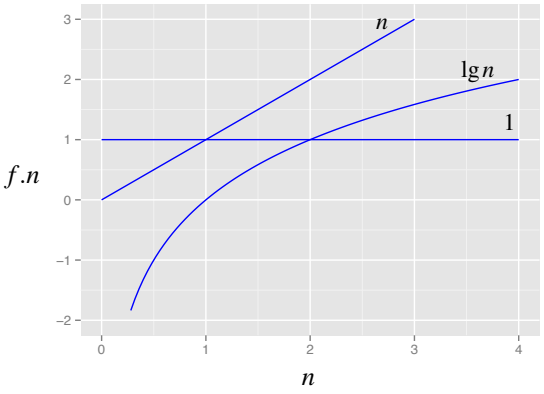


Figure 3.9 Three functions of n. The function $\lg n$ is $\log_2 n$ and is equal to 1 when n equals 2.

$$
\begin{aligned}
& 3n^2 + 2n\lg n \\
\leq \quad & \langle \text{Replacing } \lg n \text{ with a larger value} \rangle \\
& 3n^2 + 2n \cdot n \\
= \quad & \langle \text{Math} \rangle \\
& 5n^2 \\
= \quad & \langle \text{Provided } c = 5 \rangle \\
& cn^2
\end{aligned}
$$

So, $c = 5$, and any positive n_0 is possible, say $n_0 = 1$. ∎

And here is the proof that $f.n = 3n^2 + 2n\lg n = \Omega(n^2)$.

Proof: Must prove that there exist positive constants c, n_0 such that $f.n = 3n^2 + 2n\lg n \geq cn^2 \quad \text{for } n \geq n_0$.

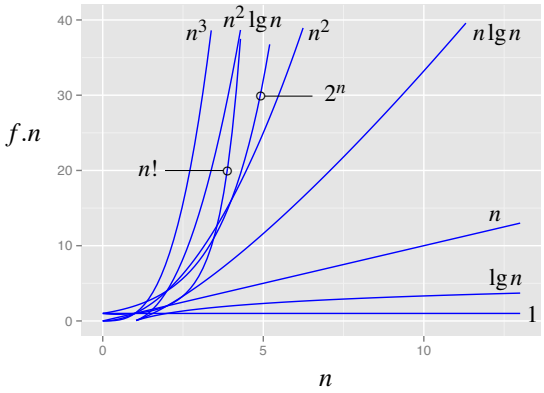
$$
\begin{aligned}
& 3n^2 + 2n\lg n \\
\geq \quad & \langle \text{Eliminating a positive value, provided } n \geq 1 \rangle \\
& 3n^2 \\
= \quad & \langle \text{Provided } c = 3 \rangle \\
& cn^2
\end{aligned}
$$

So, $c = 3$, and $n_0 = 1$. ∎

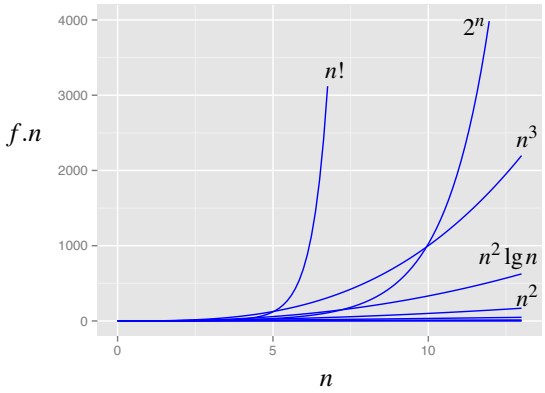
Note in the first step that n must be at least 1 for $\lg n$ to be positive.

The above example shows that any n^2 term asymptotically predominates over any $n\lg n$ term, because n predominates over $\lg n$. Similarly, any $n^2\lg n$ term asymptotically predominates over any n^2 term, because $\lg n$ predominates over 1. Figure 3.10 is a plot of nine functions of n that shows these relationships. In Figure 3.10(a), you can see that $\lg n$ is between n and 1, n is between $n\lg n$ and $\lg n$, $n\lg n$ is between n^2 and n, and n^2 is between $n^2\lg n$ and $n\lg n$.

The algorithm for towers of Hanoi is $\Theta(2^n)$. It is not clear from Figure 3.10(a) how 2^n compares with $n^2\lg n$ and n^3. From the plot, it might even seem that n^3 predominates over 2^n. However, that is decidedly *not* the case. There is a crossover point, which



(a) Vertical scale from 0 to 40.



(b) Vertical scale from 0 to 4000.

Figure 3.10 Nine functions of n. The same nine functions appear in both plots, but the vertical scale of the plot in part **(b)** is zoomed out by a factor of 100 over the vertical scale in part **(a)**.

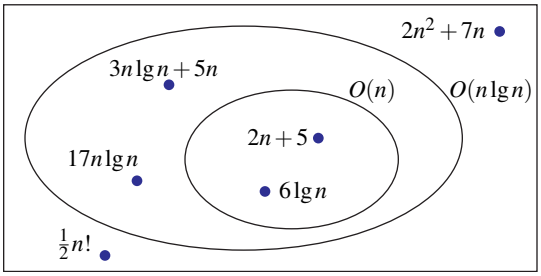
is visible in part (b) of the figure. The crossover point is slightly less than $n = 10$, as $10^3 = 1000$ and $2^{10} = 1024$. It is true in general that $2^n = \Omega(n^b)$ for any constant b. For example, 2^n predominates over n^{100}. The crossover point is further out on the n axis, but the crossover point does exist.

Figure 3.10(b) shows that function $f.n = n!$ is even worse than $f.n = 2^n$. Algorithms that have an execution time of $\Omega(n^b)$ for some constant b are called *polynomial-time algorithms*. There is a huge difference in asymptotic execution time between polynomial-time algorithms and all the worse ones, such as those with times $\Theta(2^n)$ and $\Theta(n!)$. The polynomial-time algorithms are called *easy* or *tractable*, while all the other ones are called *hard* or *intractable*.

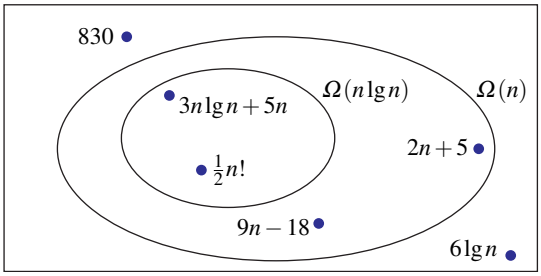
There is a famous class of computational problems that have been solved with algorithms that are intractable. It would be a huge benefit if any one of these problems could be solved with algorithms that are tractable. Its execution time would dramatically decrease if such an algorithm could be discovered. These problems, called *NP-complete* problems, are all related to each other in a special way. Even though computer scientists have not discovered a tractable algorithm for any of these problems, they have been able to prove that if any *one* of them can be solved with a tractable algorithm, then they can *all* be solved with tractable algorithms. Because so many people have tried but failed to find tractable algorithms for this special class of problems, nearly everyone believes that such algorithms are impossible. The interesting dilemma, however, is that no one has been able to *prove* that such algorithms do not exist. The class of problems that can be solved in polynomial time is called P. The big question is usually formulated, Does $P = NP$?

Even though it is widely used in the literature, asymptotic notation is not mathematically precise. For example, the expression

$$f.n = O(n \lg n)$$



(a) Some functions in the sets $O(n)$ and $O(n\lg n)$.



(b) Some functions in the sets $\Omega(n)$ and $\Omega(n\lg n)$.

Figure 3.11 A Venn diagram of some functions in the sets O and Ω.

does not really mean that the expression on the left of $=$ is equal to the expression on the right. For two expressions to be equal they must have the same type. The expression on the left is a function, while the expression on the right is a *set* of functions. A more precise notation would be

$$f.n \in O(n\lg n)$$

where the symbol $\in$ represents the set membership operator, and the expression is read as, "$f.n$ is an element of the set $O(n\lg n)$."

Figure 3.11 is a Venn diagram of some of some typical functions. Part (a) shows that $O(n)$ is a proper subset of $O(n\lg n)$. Function $f.n = 2n+5$ is in both $O(n)$ and $O(n\lg n)$, because it is bounded above by both n and $n\lg n$. Function $f.n = 3n\lg n+5n$ is not in $O(n)$, because it is not bounded above by n. However, it is in $O(n\lg n)$, because it is bounded above by $n\lg n$. Part (b) shows that $\Omega(n\lg n)$ is a proper subset of $\Omega(n)$. Function $f.n = 3n\lg n+5n$ is in both, because it is bounded below by both $n\lg n$ and n.

Following is a summary of the asymptotic orders that describe typical algorithm performance.

Asymptotic orders:

$$O(1) \subset O(\lg n) \subset O(n) \subset O(n\lg n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n!)$$

$$\Omega(1) \supset \Omega(\lg n) \supset \Omega(n) \supset \Omega(n\lg n) \supset \Omega(n^2) \supset \Omega(n^3) \supset \Omega(2^n) \supset \Omega(n!)$$

Another imprecision with asymptotic notation is the combination of expressions by addition or multiplication. For example, suppose you analyze an algorithm that consists of two parts that execute one after the other. Say you determine that the first part has tight bound $\Theta(n\lg n)$ and the second part has tight bound $\Theta(n)$. The total execution time is the sum of these times. It is customary to write the expression for the total time as

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

even though you cannot add sets with the $+$ operator this way. The interpretation of the addition is that $\Theta(n\lg n)$ represents a specific function with a tight bound of $n\lg n$, and $\Theta(n)$ represents another specific function with a tight bound of n. If you add two such

functions, you can reason that the $n \lg n$ term from the first function will predominate over the n term from the second function and the result will be a function with a tight bound of $\Theta(n \lg n)$.

As another example, suppose an algorithm has a loop that executes n times. In the body of the loop is code that executes with a tight bound of $n \lg n$. It is customary to write the total time as

$$\Theta(n) \cdot \Theta(n \lg n) = \Theta(n^2 \lg n)$$

even though you cannot multiply sets. The interpretation of the multiplication is that $\Theta(n \lg n)$ represents a specific function with a tight bound of $n \lg n$, and $\Theta(n)$ represents another specific function with a tight bound of n. If the body of the loop executes on the order of n times, and each execution the body requires on the order of $n \lg n$ executions, then the total number of executions will be $n \cdot n \lg n$, which is $\Theta(n^2 \lg n)$.

The mathematical relation of equality is reflexive because $x = x$. Similarly, the asymptotic bounds are reflexive in the following sense.

Reflexivity:

$$f.n = O(f.n).$$
$$f.n = \Omega(f.n).$$
$$f.n = \Theta(f.n).$$

The mathematical relation of equality is symmetric because $x = y \equiv y = x$. Similarly, the asymptotic bounds are symmetric in the following sense.

Symmetry: $f.n = \Theta(g.n) \equiv g.n = \Theta(f.n)$.
Transpose symmetry: $f.n = O(g.n) \equiv g.n = \Omega(f.n)$.

The mathematical relation of greater than is transitive because $x > y \wedge y > z \Rightarrow x > z$. Similarly, the asymptotic bounds are transitive in the following sense.

Transitivity:

$$f.n = O(g.n) \wedge g.n = O(h.n) \Rightarrow f.n = O(h.n).$$
$$f.n = \Omega(g.n) \wedge g.n = \Omega(h.n) \Rightarrow f.n = \Omega(h.n).$$
$$f.n = \Theta(g.n) \wedge g.n = \Theta(h.n) \Rightarrow f.n = \Theta(h.n).$$

These properties can all be proved from the definitions of the asymptotic bounds.

3.4 Program Correctness

To prove that a program is correct is to prove that it follows its specification. A specification consists of a precondition, which describes any initial requirements that must be met before the program executes, and a postcondition, which the program guarantees is true after it executes. Most algorithms contain repetition produced by an iterative loop or by a recursive call. The proof heuristics are different for these two programming techniques.

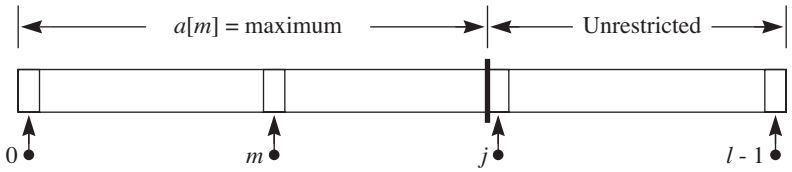


Figure 3.12 The loop invariant for the function `largestLast()`.

Proving iterative algorithms

An iterative algorithm achieves its repetition with a loop statement such as the `while` statement or the `for` statement. The key idea is to identify the relevant *loop invariant*, which is a boolean expression related to the postcondition. Roughly speaking, the loop invariant is an expression that is true for part of the problem. As the loop progresses, that part of the problem gets bigger until, at the end of the loop, it encompasses the entire problem. To prove that an iterative loop is correct is a four-step process.

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

An example of the proof technique is the following proof of correctness for the function `largestLast()`. Figure 3.1 is a trace of the algorithm. Parts (a) – (i) show the effect of the loop, which determines the index of the maximum value in the array. Here is the code for the loop.

```
int indexOfMax = 0;
for (int j = 1; j < len; j++) {
    if (a[indexOfMax] < a[j]) {
        indexOfMax = j;
    }
}
```

The remaining code in the function swaps `a[len-1]` with `a[indexOfMax]`, which guarantees that the last element is the largest one.

The main part of the proof is to show that the above loop determines the index of the maximum element. To simplify the math notation, use m to represent `indexOfMax` and l to represent `len`. With these abbreviations, the precondition is $l > 0$. l cannot be equal to zero because there must be at least one element in the array for a maximum element to exist. The postcondition is a statement that $a[m]$ is the largest element with $0 \leq m < l$. Formally, the postcondition is

$$0 \leq m < l \wedge (\forall i \mid 0 \leq i < l : a[m] \geq a[i]) .$$

The next step is to determine the loop invariant. The loop invariant represents progress toward the postcondition. In this problem, the postcondition is the assertion that $a[m]$ is the largest element in the range $0 \leq m < l$. The loop invariant is the same assertion, but not for the entire range. Figure 3.1 shows a dark vertical bar to the left of

j at each point in the trace when the value of j changes. In part (d) of the figure, $j = 2$ and $m = 0$. At this point in the execution of the loop, $a[m]$ is the largest element in the limited range $0 \leq m < 2$.

Figure 3.12 shows the general case. The dark vertical bar to the left of j divides a into two regions. In the region $a[0..j-1]$, $a[m]$ is the maximum element. The region $a[j..l-1]$ has not yet been processed by the loop. Hence, $a[m]$ may or may not be the maximum of the values in this unrestricted region. With each execution of the loop, the unrestricted region gets smaller until, when the loop terminates, it is empty and the postcondition holds. Formally, the loop invariant is

$$(\forall i \mid 0 \leq i < j : a[m] \geq a[i]) .$$

Here are the four steps of the proof of correctness of the loop in `largestLast()`. The invariant is true at the beginning of the loop.

Proof: Starting with the invariant,

$$\begin{array}{ll} & (\forall i \mid 0 \leq i < j : a[m] \geq a[i]) \\ = & \langle \text{Assignment statements } \texttt{indexOfMax = 0} \text{ and } \texttt{j = 1} \text{ in } \texttt{largestLast()} \rangle \\ & (\forall i \mid 0 \leq i < 1 : a[0] \geq a[i]) \\ = & \langle \text{Math, } 0 \leq i < 1 \ \equiv \ i = 0 \rangle \\ & (\forall i \mid i = 0 : a[0] \geq a[i]) \\ = & \langle \text{One-point rule} \rangle \\ & a[0] \geq a[0] \\ = & \langle \text{Math} \rangle \\ & \mathit{true} \ \blacksquare \end{array}$$

The invariant is maintained with each execution of the loop.

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

Case 1: $a[m] \geq a[k]$. The only code that executes is `j++`. Increasing j by one excludes the value of $a[j]$ from the unrestricted region and includes it it the $a[m] =$ maximum region. The invariant is maintained because of the `if` guard, $a[m] < a[j]$.

Case 2: $a[m] < a[j]$. First, the assignment in the `if` statement gives the value of j to m. Then, `j++` executes. Before the `if` statement executes, $a[m]$ has the maximum value in the region $a[0..j-1]$. Because of the `if` guard, $a[j]$ is greater than this maximum. Hence, by transitivity of the $<$ operator, $a[j]$ is the maximum of all the values in the region $a[0..j]$. After j is incremented by 1, $a[j]$ is the maximum of all the values in the region $a[0..j-1]$ which is precisely the loop invariant. $\blacksquare$

The loop terminates.

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

```
for (int j = 1; j < len; j++)
```

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

The postcondition holds at the end of the loop.

Proof: Starting with the invariant,

$$
\begin{aligned}
& (\forall i \mid 0 \le i < j : a[m] \ge a[i]) \\
= \quad & \langle \text{The final value of } j \text{ is } l \rangle \\
& (\forall i \mid 0 \le i < l : a[m] \ge a[i])
\end{aligned}
$$

which is the second conjunct of the postcondition of the loop.

The first conjunct is $0 \le m < l$, which holds because m is initialized to 0, and the only other statement that can change m is the assignment statement

```
indexOfMax = j;
```

The guard on the `for` statement, `j < len`, guarantees that the assignment happens only when $j < l$. ∎

Proving recursive algorithms

A recursive algorithm achieves repetition by calling itself. To keep the repetition finite the algorithm must have an `if` statement to distinguish between the base case and the inductive case. The recursive calls are only in the inductive case and must be calls for problems smaller than the original call. Proof of correctness for a recursive algorithm is by mathematical induction, as the base case and inductive case for the algorithm correspond directly to the base case and the inductive case for the mathematical induction proof.

Many recursive algorithms are related directly to recursively defined data structures or mathematical functions. The general technique is to use code inspection to determine the processing in the algorithm, and then show that the processing matches the data structure or mathematical function.

For example, consider the recursive function to compute the factorial of integer n.

```
factorial(n)
   if n <= 0 then
      return 1
   else
      return n * factorial(n - 1)
```

The proof that it is correct consists of the observation that the program matches the mathematical definition.

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

Proof: Base case. The base case is $n = 0$. By code inspection, the function returns 1. By the mathematical definition, $0! = 1$.

Inductive case. The inductive case is $n > 0$. By code inspection, the function returns $n \cdot (n-1)!$. By the mathematical definition, $n! = n \cdot (n-1)!$. ∎

For a problem that does not have a natural recursive definition, the proof requires the formulation of an inductive hypothesis in the inductive case. For example, consider the following algorithm to compute the sum of the values in the array segment $a[0..n-1]$ where $n \ge 0$ is the number of elements in the array.


```
sum(a, n)
   if n == 0 then
      return 0
   else
      return sum(a, n - 1) + a[n - 1]
```

The postcondition is

Function $\text{sum}(a,n)$ returns $(\Sigma\, i \mid 0 \le i < n : a[i])$.

Here is the proof by mathematical induction that function `sum()` is correct.

Proof: Base case. The base case is $n = 0$, which represents the empty array. By code inspection, the function returns 0. By the mathematical definition of summation, the sum of the elements in the empty array is 0 as follows. Starting with the expression returned as specified in the postcondition,

$$
\begin{aligned}
& (\Sigma\, i \mid 0 \le i < n : a[i]) \\
= \quad & \langle \text{Base case}, n = 0 \rangle \\
& (\Sigma\, i \mid 0 \le i < 0 : a[i]) \\
= \quad & \langle \text{Math}, \ 0 \le i < 0 \ \equiv \ \mathit{false} \rangle \\
& (\Sigma\, i \mid \mathit{false} : a[i]) \\
= \quad & \langle \text{Empty range rule}, 0 \text{ is the identity for addition} \rangle \\
& 0
\end{aligned}
$$

Inductive case. Prove that

Function $\text{sum}(a,n)$ returns $(\Sigma\, i \mid 0 \le i < n : a[i])$

assuming that

Function $\text{sum}(a,n-1)$ returns $(\Sigma\, i \mid 0 \le i < n-1 : a[i])$

as the inductive hypothesis.

$$
\begin{aligned}
& \text{Value returned by function } \text{sum}(a,n) \\
= \quad & \langle \text{Code inspection} \rangle \\
& \text{sum}(a,n-1) + a[n-1] \\
= \quad & \langle \text{Inductive hypothesis} \rangle \\
& (\Sigma\, i \mid 0 \le i < n-1 : a[i]) + a[n-1] \\
= \quad & \langle \text{Math, split off term} \rangle \\
& (\Sigma\, i \mid 0 \le i < n : a[i]) \quad \blacksquare
\end{aligned}
$$

Exercises

3–1 Figure 3.2 shows the statement execution counts for `largestLast()`. **(a)** What data pattern in the original list causes the best case to occur? **(b)** What data pattern in the original list causes the worst case to occur?

3–2 Figure 3.4 shows the statement execution counts for `selectionSort()`. **(a)** What data pattern in the original list causes the best case to occur? **(b)** What data pattern in the original list causes the worst case to occur?

3–3 Prove the following summations using mathematical induction.

(a) $(\Sigma\, i \mid 1 \le i \le n : i) = n(n+1)/2 \quad \text{for } n \ge 0.$

(b) $(\Sigma\, i \mid 1 \le i \le n : i^2) = n(n+1)(2n+1)/6 \quad \text{for } n \ge 0.$

3–4 Perform a statement execution count for the following algorithm. Show the number of statements executed by each line. For this pseudocode on line 2, the first execution of outer loop is with a value of 1, the last execution is with a value of n, and then there is an additional test to terminate the loop. The `while` statement on line 4 is like the C++ `while` statement with the test at the top of the loop. Evaluate all the summations for both the best case and the worst case, and write each total as a polynomial in n.

```
1  count = 0
2  for i = 1 to n do
3     j = 0
4     while j < i do
5        input num
6        if num < 0 then
7           count = count + 1
8        j = j + 1
9     output j
10 output count
```

3–5 Suppose you measure the execution time of an algorithm to be one second when you run it with 1,000 data values. Estimate how long it would take to execute with 10,000 values assuming the following tight bounds for the algorithm. If the time is greater than a minute give the number of minutes, unless the time is also greater than an hour, in which case give the number of hours, unless the time is also greater than a day, in which case give the number of days, unless the time is also greater than a year, in which case give the number of years in scientific notation with a power of 10. Give each answer in decimal to three significant figures.

(a) $\Theta(1)$ **(b)** $\Theta(\lg n)$ **(c)** $\Theta(n)$ **(d)** $\Theta(n \lg n)$
(e) $\Theta(n^2)$ **(f)** $\Theta(n^2 \lg n)$ **(g)** $\Theta(n^3)$ **(h)** $\Theta(n^{10})$
(i) $\Theta(2^n)$

3–6 Prove the following asymptotic tight bounds from the definitions of O and Ω.

(a) $3n^3 + 2n - 1 = \Theta(n^3)$ **(b)** $2n^2 - 5n + 1 = \Theta(n^2)$
(c) $n^3 - 100n = \Theta(n^3)$ **(d)** $6n^3 - 100n = \Theta(n^3)$
(e) $4n^2 + n \lg n - n = \Theta(n^2)$ **(f)** $2n \lg n - 5n + 6 = \Theta(n \lg n)$
(g) $3n2^n + 5n = \Theta(n2^n)$ **(h)** $3n2^{n+1} + 5n = \Theta(n2^n)$
(i) $n^3 - 2n^2 \lg n + 3n^2 - 4n \lg n + 5n - 6 \lg n + 7 = \Theta(n^3)$

3–7 Write the recurrence for this inefficient algorithm for computing the binomial coefficient.

```
binomCoeff(n, k)
   if k == 0 then
      return 1
   else if k == n then
      return 1
   else
      return binomCoeff(n - 1, k - 1)
```

3–8 Write the recurrence for this inefficient algorithm for computing the nth Fibonacci number

```
fib(n)
   if n == 0 then
      return 0
   else if n == 1 then
      return 1
   else
      return fib(n - 1) + fib(n - 2)
```

3–9 (Gregory Boudreaux) The sum of the first four 1's is

$$(\Sigma\, i \mid 1 \le i \le 4 : 1) = 1+1+1+1 = 4\,.$$

The sum of the first four integers is

$$(\Sigma\, i \mid 1 \le i \le 4 : i) = 1+2+3+4 = 10\,.$$

The sum of the first four squares is

$$(\Sigma\, i \mid 1 \le i \le 4 : i^2) = 1+4+9+16 = 30\,.$$

Write the recurrence for the following algorithm, which computes this sum of the first n integers raised to the power j.

```
sumP(n, j)
   if j == 0 then
      return n
   else if n == 1 then
      return 1
   else
      temp = 0
      for k = 2 to n do
         temp += sumP(k - 1, j - 1)
      return n * sumP(n, j - 1) - temp
```

Use ellipses … in your expression for the inductive part.

3–10 Use backward substitution to find the closed-form solution to the following recurrences.

(a) $T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(n/2)+1 & \text{if } n = 2^k, \text{ for } k > 1. \end{cases}$

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

3–11 Prove using mathematical induction that the following recurrences have the given closed-form solutions. In the inductive part of the proof, state what you are to prove, and state the inductive hypothesis. In the proof, identify the step where you use the inductive hypothesis.

(a) $T(n) = \begin{cases} 6 & \text{if } n = 0, \\ 2T(n-1) & \text{if } n > 0, \end{cases}$ with solution $T(n) = 3 \cdot 2^{(n+1)}$.

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

(c) $T(n) = \begin{cases} 3 & \text{if } n = 1, \\ 2T(n/2)-3 & \text{if } n > 1, \end{cases}$ with solution $T(n) = \lg n + 3$.

(d) $T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) - \lg n + 2 & \text{if } n > 1, \end{cases}$ with solution $T(n) = n + \lg n$.

(e) $T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 2T(n-1) - n + 2 & \text{if } n > 1, \end{cases}$ with solution $T(n) = 2^n + n$.

(f) $T(n) = \begin{cases} 0 & \text{if } n = 1, \\ 4T(n/2)+n^2 & \text{if } n > 1, \end{cases}$ with solution $T(n) = n^2 \lg n$.

(g) For this problem, assume that n is even. Prove the solution for $n+2$ assuming the solution for n as the inductive hypothesis.

$T(n) = \begin{cases} 6 & \text{if } n = 0, \\ 2T(n-2)-5 & \text{if } n > 1, \end{cases}$ with solution $T(n) = 2^{n/2} + 5$.

3–12 **(a)** The golden ratio is $\phi = (1+\sqrt{5})/2$, which is approximately equal to 1.618. Prove that $\phi^2 = \phi + 1$.

(b) The conjugate of the golden ratio is $\hat{\phi} = (1-\sqrt{5})/2$, which is approximately equal to -0.618. Prove that $\hat{\phi}^2 = \hat{\phi} + 1$.

(c) Prove using mathematical induction that $T(n) = 2(\phi^{n+1} - \hat{\phi}^{n+1})/\sqrt{5} - 1$ is the solution to the recurrence

$T(n) = \begin{cases} 1 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ T(n-1)+T(n-2)+1 & \text{if } n > 1. \end{cases}$

There are two base cases and two inductive hypotheses. Because $\phi > 1$ and $|\hat{\phi}| < 1$ the predominant term in the solution is ϕ^{n+1}. This exercise shows that $T(n) = \Theta(\phi^n)$. So, the asymptotic bound for the inefficient recursive algorithm for computing the nth Fibonacci number is exponential.

3–13 Use the master method to determine the asymptotic execution times for the following recurrences.

(a) $T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 16T(n/2) + n^3 & \text{if } n > 1. \end{cases}$

(b) $T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/4) + n & \text{if } n > 1. \end{cases}$

(c) $T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 4T(n/4) + n & \text{if } n > 1. \end{cases}$

3–14 Draw two Venn diagrams corresponding to the ones in Figure 3.11 but for the sets $O(n \lg n)$ and $O(n^2)$ in the first diagram and sets $\Omega(n \lg n)$ and $\Omega(n^2)$ in the second diagram. Place the following functions in their proper positions in each diagram.

$2n^2 + 3n$	$4n \lg n - 7n$	271	$4n \lg n + 7n^3$
$3n + 1$	$2n! - 5n^2$	$5 \cdot 2^n + \lg n$	$4n \lg n + 7n^2$

3–15 You do not need to prove your answer for the following questions.

(a) Is it true that $2^{n+1} = O(2^n)$?

(b) Is it true that $2^{2n} = O(2^n)$?

(c) Is it true that $10^n = O(2^n)$?

3–16 This exercise is to determine the asymptotic behavior of $f.n = n^{1/2}$ compared to $g.n = \lg n$. You may need the fact $\lg x = \ln x / \ln 2$.

(a) For what value of n does $n^{1/2} = \lg n$?

(b) Considering $f.x = x^{1/2}$ and $g.x = \lg x$ to be a continuous functions of x, find expressions for the derivatives $\frac{d}{dx} x^{1/2}$ and $\frac{d}{dx} \lg x$.

(c) To three places past the decimal point, what are the slopes of the functions $f.x$ and $g.x$ at the crossover point determined in part (a)?

(d) Which is greater at the crossover point, the slope of $f.x$ or the slope of $g.x$?

(e) Will the greater slope always be greater than the other slope at values of x beyond the crossover point? Why?

(f) Which is true, $n^{1/2} = O(\lg n)$ or $\lg n = O(n^{1/2})$?

3–17 Prove the following reflexivity properties.

(a) $f.n = O(f.n)$.

(b) $f.n = \Omega(f.n)$.

(c) $f.n = \Theta(f.n)$.

Prove the following symmetry properties.

(d) $f.n = \Theta(g.n) \equiv g.n = \Theta(f.n)$.

(e) $f.n = O(g.n) \equiv g.n = \Omega(f.n)$.

Prove the following transitivity properties.

(f) $f.n = O(g.n) \wedge g.n = O(h.n) \Rightarrow f.n = O(h.n)$.

(g) $f.n = \Omega(g.n) \wedge g.n = \Omega(h.n) \Rightarrow f.n = \Omega(h.n)$.

(h) $f.n = \Theta(g.n) \wedge g.n = \Theta(h.n) \Rightarrow f.n = \Theta(h.n)$.

3–18 Here is an algorithm to calculate the sum of all the values in the array $a[0..l-1]$.

```
int sum = 0;
for (int j = 0; j < len; j++) {
    sum += a[j];
}
```

For this exercise, use the abbreviations l for `len` and s for `sum`.

(a) Write a formal expression for the postcondition.

(b) Write a formal expression for the loop invariant.

(c) Prove all four steps in the proof of correctness for the algorithm.

3–19 The proof of correctness example for the recursive computation of $n!$ assumed a mathematical definition that was also recursive. Using the following iterative definition

$$n! = (\Pi\, i \mid 1 \le i \le n : i) \qquad \text{for } n > 0$$

instead of the recursive one, prove that the recursive computation of $n!$ is correct as follows.

(a) State the postcondition.

(b) Prove the base case.

(c) For the inductive case, state what is to be proved and what the inductive hypothesis is.

(d) Prove the inductive case.

3–20 Here is a recursive algorithm for computing n^2.

```
square(n)
   if n == 0 then
      return 0
   else
      return square(n - 1) + 2 * n - 1
```

Prove that the recursive computation of n^2 is correct as follows.

(a) State the postcondition.

(b) Prove the base case.

(c) For the inductive case, state what is to be proved and what the inductive hypothesis is.

(d) Prove the inductive case.

3–21 Exercise 9 shows a recursive algorithm for computing the sum of the first n integers raised to the power j. The postcondition for this algorithm is

Function sumP(n, j) returns $(\Sigma\, i \mid 0 \le i \le n : i^j)$.

Prove that the computation is correct using mathematical induction on j for constant n as follows.

(a) Prove the base case using $j = 0$.

(b) For the inductive case, state what is to be proved (using j) and what the inductive hypothesis is (using $j-1$).

(c) Prove the inductive case. You may use the following theorem in your proof.

$$(\Sigma\, k \mid 2 \le k \le n : (\Sigma\, i \mid 1 \le i \le k-1 : f.i)) = (\Sigma\, i \mid 1 \le i \le n : (n-1) \cdot f.i)$$

