CHAPTER 12

Algorithm Efficiency, Searching, and Sorting

INTRODUCTION

In a user's eyes, many factors contribute to the quality of a program. First and foremost, the program must be correct. In addition, convenience of the user interface, ease of maintenance, and robustness may all be important. This chapter examines another common concern of users: performance.

Many important problems exceed the capabilities of human inventions. Current supercomputers are capable of executing at speeds measured in gigaflops (one billion floating point operations per second). Yet even these machines are too slow for certain applications such as global weather forecasting.

Code performance is an important issue for problems that push the limits of technology—and for smaller problems as well. And even when performance may not be the programmer's top priority, the customer is always happier with a more efficient solution.

The run-time efficiency of a program has two main ingredients: space and time. **Space efficiency** is a measure of the amount of memory or storage a program requires. Computers have a fixed amount of available storage. If two programs perform identical functions, the one consuming less storage is more space-efficient. Sometimes conservation of storage is the dominant factor in software. However, the concern for space efficiency has diminished in recent years because of rapid decreases in the cost of computer storage hardware.

Time efficiency is a measure of how long a program takes to execute. If two programs correctly satisfy a program specification, the one that executes faster is more time-efficient.

The study of efficiency has developed into a separate field of computer science known as **complexity theory.** The main components of complexity theory are **space complexity,** the study of space issues, and **time complexity,** the study of time issues. Space and time efficiency are the most important aspects of code performance. This chapter concentrates primarily on time efficiency. Even though issues of performance must be addressed, keep in mind the larger perspective: a program's *correctness*, not its speed, is the paramount concern. It is always better to have a slower program that executes correctly than a program that executes incorrectly at lightning speed.

12.1 Big-O Notation

A programmer usually knows many alternative algorithms for accomplishing a particular task, so it is important to identify the major differences among the algorithms. A widely used technique for revealing large differences in efficiency comes from the study of **performance analysis.** Performance analysis applies to both space and time efficiency, though this chapter focuses on the latter.

The performance of an algorithm is measured in terms of some value, usually the amount of data processed. For example, the execution time of Algorithm A may depend largely upon the size, N, of some vector. That is, its execution time is proportional to N. An alternative algorithm, Algorithm B, may require time proportional to N^2. Tripling the size of the vector causes A's execution time to approximately triple, but causes B's time to increase 9-fold. Intuitively, Algorithm B has poorer time efficiency than Algorithm A because its execution time increases at a rate proportional to N^2.

Function Dominance

To define performance more formally, we use **cost functions.** A cost function is a numeric function that gives the performance of an algorithm in terms of one or more variables. (Typically, the variables capture the amount of data being processed by the algorithm.) For example, suppose that a particular program processes an array `vSize` cells in length. Suppose also that the cost function for the execution time of this program is

$$\text{ActualTime}(\texttt{vSize}) = \texttt{vSize}^2 + 5\texttt{*}\texttt{vSize} + 100$$

According to this cost function, the computing time for the program when given an array consisting of a single cell is ActualTime(1) = 106 time units. Similarly, the execution time of the program for an array of length 100 is ActualTime(100) = 10,600 time units. (For this discussion, the particular time units used by the cost function are not important.)

Ideally, it would be possible to capture the execution time of any algorithm in terms of simple cost functions like the preceding ActualTime function. However, in reality such cost functions are often impossible to derive and are too complicated to be truly instructional. Because of this difficulty in using actual cost functions, computer scientists generally use functions that *approximate* the actual cost.

To examine ways of approximating a cost function, we begin with the mathematical notion of **dominance.** Given two cost functions f and g, g is said to *dominate* f if there is some positive constant c such that

$$c * g(x) \geq f(x)$$

for all possible values of x.

Performance analysis uses a variation of dominance known as **asymptotic dominance.** A function g asymptotically dominates another function f if there are positive constants c and x_0 such that

$$c * g(x) \geq f(x) \text{ for all } x \geq x_0$$

In other words, g asymptotically dominates f if g dominates f for all "large" values of x. Computer scientists regularly refer to asymptotic dominance simply as "dominance." Informally, an asymptotically dominant function can be thought of as underestimating the actual cost only for small amounts of data.

For example, suppose that the following function is being considered as a possible estimate for the earlier ActualTime function:

$$\text{TimeEst}(\texttt{vSize}) = 1.1 * \texttt{vSize}^2$$

Figure 12.1 shows why TimeEst is a reasonable estimate. This figure graphs the ActualTime and TimeEst functions for various values of `vSize`. For small values of `vSize`, both ActualTime and TimeEst increase in value (grow) at about the same rate. For values of `vSize` up to about 60, TimeEst(`vSize`) is less than ActualTime(`vSize`). When `vSize` > 60, TimeEst grows at a faster rate than ActualTime. TimeEst thus dominates ActualTime asymptotically. That is, for large variable values (`vSize` > 60), the value of TimeEst(`vSize`) always exceeds the value of ActualTime(`vSize`).

Many other functions can be shown to dominate ActualTime. A few of them are

$$\text{TimeEst1}(\texttt{vSize}) = 1.01 * \texttt{vSize}^2$$
$$\text{TimeEst2}(\texttt{vSize}) = 1.5 * \texttt{vSize}^2$$
$$\text{TimeEst3}(\texttt{vSize}) = 2 * \texttt{vSize}^2$$
$$\text{TimeEst4}(\texttt{vSize}) = \texttt{vSize}^3$$

Among these functions it can be demonstrated that TimeEst4 dominates TimeEst3, TimeEst3 dominates TimeEst2, TimeEst2 dominates TimeEst, TimeEst dominates TimeEst1, and TimeEst1 dominates ActualTime.

Estimating Functions

The most desirable estimating function for ActualTime is a function that has three characteristics:

1. It asymptotically dominates the ActualTime function.
2. It is simple to express and understand.
3. It is as close an estimate as possible.

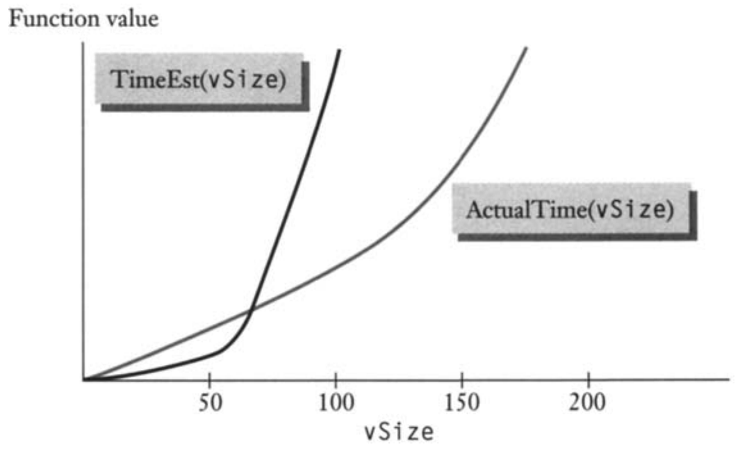


Figure 12.1 Graph of TimeEst(vSize) dominating ActualTime(vSize) asymptotically

Given the function

$$\text{ActualTime}(\texttt{vSize}) = \texttt{vSize}^2 + 5*\texttt{vSize} + 100$$

the following function is a good estimate of ActualTime:

$$\text{EstimateOfActualTime}(\texttt{vSize}) = \texttt{vSize}^2$$

First, EstimateOfActualTime asymptotically dominates ActualTime. Any constant *c* greater than 1 satisfies

$$c * \texttt{vSize}^2 \geq \texttt{vSize}^2 + 5*\texttt{vSize} + 100$$

for large values of vSize. Second, $\texttt{vSize}^2$ is simpler to express than other dominating functions we have mentioned: $1.01*\texttt{vSize}^2$, $1.5*\texttt{vSize}^2$, $2*\texttt{vSize}^2$. Third, the function $\texttt{vSize}^3$ would also satisfy the first two properties but is not as close an estimate as $\texttt{vSize}^2$.

To express time estimates more concisely, mathematicians and computer scientists use a concept called the **order of a function.** The order of a function *f* is defined as follows:

Given two nonnegative functions f *and* g, the order of f is g *if and only if* g *asymptotically dominates* f.

There are two other ways of saying "the order of *f* is *g*":

1. "*f* is of order *g*"
2. "*f* = O(*g*)"

The second of these is an example of **big-O notation** (or big-oh notation). The uppercase letter O stands for "Order."

At first, big-O notation may be very confusing. The statement *f* = O(*g*) seems to say that the order of *g* is *f*, when in fact it means that the order of *f* is *g*. Also, with big-O notation you should think of the "=" symbol as meaning "is" rather than "equals." The best way to pronounce $f = O(g)$ is to say "f is of order g."

Applying big-O notation to our ActualTime and EstimateOfActualTime functions, we substitute ActualTime for f and EstimateOfActualTime for g to obtain

$$\text{ActualTime}(\texttt{vSize}) = O(\text{EstimateOfActualTime}(\texttt{vSize}))$$

or

$$\texttt{vSize}^2 + 5*\texttt{vSize}+100 = O(\texttt{vSize}^2)$$

It is best to pronounce the last expression as "$\texttt{vSize}^2 + 5*\texttt{vSize} + 100$ is of order $\texttt{vSize}^2$."

Referring to an algorithm's running time (execution time), if we write

$$\text{ActualTime}(\texttt{vSize}) = O(\texttt{vSize}^2)$$

then it is appropriate to say that the running time "is of order $\texttt{vSize}^2$" or "is proportional to $\texttt{vSize}^2$." Also, we say that the associated algorithm "has running time $O(\texttt{vSize}^2)$" or that the algorithm "is an $O(\texttt{vSize}^2)$ algorithm."

It is important to emphasize that big-O notation expresses the *relative* speed of an algorithm, not its absolute speed. An $O(N)$ algorithm increases its running time roughly in proportion to increasing values of N. Similarly, an $O(N^2)$ algorithm increases its running time roughly in proportion to the square of N. But the precise running time cannot be determined from a big-O measure. Two algorithms that are both $O(N)$ may very well have different absolute execution times for the same value of N.

Big-O Arithmetic

To help in calculating and comparing big-O estimates, there are three important rules. Figure 12.2 summarizes these rules.

According to the first rule in Figure 12.2, constant multipliers do not affect a big-O measure. For example, $O(2*N) = O(N)$, $O(1.5*N) = O(N)$, and $O(2371*N) = O(N)$.

Let f and g be functions and k a constant. Then:

1. $O(k * f) = O(f)$
2. $O(f * g) = O(f) * O(g)$
   and
   $O(f / g) = O(f) / O(g)$
3. $O(f) \geq O(g)$ if and only if f dominates g.
4. $O(f + g) = \text{Max}[O(f), O(g)]$,
   where Max denotes the larger of the two

Figure 12.2 Rules for big-O arithmetic

Let X and Y denote variables and let a, b, n, and m denote constants. Then:

X^X	dominates	$X!$	(X factorial)
$X!$	dominates	a^X	
a^X	dominates	b^X	if $a > b$
a^X	dominates	X^n	if $a > 1$
X^n	dominates	X^m	if $n > m$
X	dominates	$\log_a X$	if $a > 1$
$\log_a X$	dominates	$\log_b X$	if $b > a > 1$
$\log_a X$	dominates	1	if $a > 1$

Any term with a single variable X neither dominates nor is dominated by a term with the single independent variable Y.

Figure 12.3 Examples of function dominance

The second rule states that the order of the product of two functions is equal to the order of the first times the order of the second. For example,

$$O((17*N)*N) = O(17*N)*O(N) = O(N)*O(N) = O(N*N) = O(N^2)$$

By the third and fourth rules in Figure 12.2, the order of the sum of functions is the order of the dominant function. For example,

$$\begin{aligned} O(N^5+N^2+N) &= \text{Max}[O(N^5), O(N^2), O(N)] \\ &= O(N^5) \end{aligned}$$

To apply the fourth rule, it is essential to know which functions dominate others. Figure 12.3 displays several estimating functions that are common in computer science algorithms. The functions are listed from most dominant to least dominant.

We can use the rules from Figure 12.2 and the information from Figure 12.3 to determine estimating functions. For example, if

$$f(N) = 3*N^4 + 17*N^3 + 13*N + 175$$

then the order of f is

$$\begin{aligned} \text{Max}[O(3*N^4), O(17*N^3), O(13*N), O(175)] &= \text{Max}[O(N^4), O(N^3), O(N), O(1)] \\ &= O(N^4) \end{aligned}$$

because N^4 is the dominant function. Therefore, we can say that

$$f(N) = O(N^4)$$

Below are several additional examples. For each function, its big-O order appears on the right. All identifiers are variables.

Function	**Order**
$53*\text{employeeCount}^2$	$O(\text{employeeCount}^2)$
$65*\text{ordersProcessed}^3 + 26*\text{ordersProcessed}$	$O(\text{ordersProcessed}^3)$
$\text{headCount}^6 + 3*\text{headCount}^5 + 5*\text{headCount}^2 + 7$	$O(\text{headCount}^6)$
$75 + 993*\text{numberToSearch}$	$O(\text{numberToSearch})$

4*size*($\log_2$size) + 15*size	O(size*$\log_2$size)
5*count^4 + 2*3$^{\text{count}}$ + 85	O(3$^{\text{count}}$)
7642	O(1)
2*employees^2 + 3*employees + 4*managers + 6	O(employees^2 + managers)

Categories of Running Time

Algorithms with running time O(1) are said to take **constant time** or sometimes are referred to as **constant algorithms.** Such algorithms are very efficient and are almost always the preferred alternative. Any algorithm whose execution time never varies with the amount of the data is a constant algorithm.

Most algorithms in computer programs execute in **polynomial time,** expressed as $O(N^a)$ for some variable N (usually the number of data items processed) and some constant $a > 0$. Speeds of polynomial algorithms vary greatly. Running times $O(N)$, $O(N^2)$, and $O(N^3)$ are referred to as **linear time, quadratic time,** and **cubic time,** respectively.

Some algorithms execute in **logarithmic time.** An algorithm with running time $O(\log_a N)$ for some variable N and constant $a > 1$ will be faster than an $O(N)$ algorithm if N is large enough. Often the base of the logarithm is omitted when specifying a big-O measure. That is, $O(\log_a N)$ is often written simply as $O(\log N)$.

Algorithms of higher order than polynomial are said to be **exponential algorithms.** Exponential algorithms require time proportional to some function a^N, where N is a variable and a is a constant. Exponential algorithms are considered impractical for large values of N. Table 12.1 displays growth rates for several different estimating functions.

Table 12.2 further emphasizes the importance of big-O measures. The first column of the table displays estimating functions of various algorithms exe-

Table 12.1 Growth rates for selected functions

N	$\log_2 N$	N*$\log_2 N$	N^2	N^3	2^N	3^N
1	0	0	1	1	2	3
2	1	2	4	8	4	9
4	2	8	16	64	16	81
8	3	24	64	512	256	6,561
16	4	64	256	4,096	65,536	43,046,721
32	5	160	1,024	32,768	4,294,967,296	.
64	6	384	4,096	262,144	(Note 1)	.
128	7	896	16,384	2,097,152	(Note 2)	.
256	8	2,048	65,536	16,777,216	?????????????	

Note 1: The value here is approximately the number of machine instructions executed by a 1 gigaflop supercomputer in 5000 years.

Note 2: The value here is about 500 billion times the age of the universe in nanoseconds (billionths of a second), assuming a universe age of 20 billion years.

Table 12.2 Effect of increased computer speed

Algorithm time estimating function	Number of data collections processed on a computer 1000 times faster
N	1000.00
$N^*\log_2 N$	140.22
N^2	31.62
N^3	10.00
2^N	9.97
3^N	6.29
4^N	4.98

cuted on a particular computer. Each algorithm processes one complete collection of data in a certain amount of time. The second column lists the number of complete collections that could be processed in the same length of time by a computer 1000 times faster than the existing one.

Table 12.2 shows that a linear algorithm, when executed on a machine 1000 times faster, can process 1000 times as much data. For an algorithm of order $N^*\log N$, the increase in processing capability is only about 140 times. The quadratic algorithm improves by a factor of approximately 32; the cubic algorithm, by a factor of 10.

The last three rows of Table 12.2 point out the impracticality of exponential algorithms. Even when the speed of a computer improves 1000 times, the best exponential algorithm from this table processes less than 10 times as much data. Increasing the hardware speed has little effect upon the speed of exponential algorithms.

12.2 Control Structures and Run-Time Performance

A programmer needs to be able to inspect an algorithm and estimate its running time. Information about run-time performance can be drawn from control structures. Table 12.3 lists specific control structures and their corresponding big-O measures.

Table 12.3 Big-O measures of various control structures

Control structure	Running time
Single assignment statement	O(1)
Simple expression	O(1)
The sequence <statement1> <statement2>	The maximum of O(*S1*) and O(*S2*)
`if` (<condition>)	The maximum of O(*S1*), O(*S2*), and O(*Cond*)

Table 12.3 Big-O measures of various control structures (continued)

Control structure	**Running time**

<statement1>
`else`
    <statement2>
	(This is a worst case)

`for (i=1; i<=N; i++)`
    <statement1>
	O(*N* * *S1*)

Note: In this table, *S1* denotes the running time of <statement1>, *S2* denotes the running time of <statement2>, and *Cond* denotes the running time for evaluating <condition>.

An algorithm without loops or recursion is not very "interesting." According to Table 12.3, such an algorithm requires constant time, O(1). Without recursion or repetition, all control structures reduce to selections among O(1) instructions. The result is therefore a constant time algorithm.

Determining the efficiency of an algorithm depends primarily on identifying loops and recursions that repeat a varying number of times. For example, the following algorithm processes every vector element exactly once:

```
for (i = 0; i < vSize; i++) { // INV (prior to test):
                              //    All vec[0..i-1] have been
                              //    processed
    // Perform some simple task on element vec[i]
    // Assume this task requires constant time
}
```

This loop requires time O(`vSize`), because the loop body executes `vSize` times and the body's computing time is O(1).

Any other loop where the number of repetitions depends solely on `vSize` in this way would also have running time O(`vSize`). The initial value of the loop control variable—here, zero—could be any other constant value without altering the big-O measure of this loop. Likewise, the kind of loop (`for`, `while`, `do-while`, or recursive repetition) is unimportant in determining big-O performance.

Quadratic running time occurs when one loop is nested within another and both loops depend upon the same variable. Processing each element of a square (*n*-row by *n*-column) matrix is a simple example of a quadratic algorithm:

```
for (row = 0; row < n; row++)  // INV (prior to test):
                               //   matrix[0..row-1][0..n-1]
                               //   have been processed
    for (col = 0; col < n; col++) { // INV (prior to test):
                                    //   matrix[row][0..col-1]
                                    //   have been processed
        // Perform some simple task on element matrix[row][col]
        // Assume this task requires constant time
    }
```

The running time of this algorithm is O(n^2).

Table 12.4 Forms of repetition and corresponding big-O measures

Algorithm form	Running time
Algorithm without loop or recursion	O(1)—Constant time

`for (i=a; i<=b; i++) {`
`      // Loop body requiring`
`      // constant time`
`}`
	O(1)

`for (i=a; i<=N; i++) {`
`            // Loop body requiring`
`            // constant time`
`}`
	O(N)—Linear time

`for (i=a; i<=N; i++)`
`      for (j=b; j<=N; j++) {`
`            // Loop body requiring`
`            // constant time`
`}`
	O(N^2)—Quadratic time

`for (i=a; i<=N; i++) {`
`     // Loop body requiring`
`     // time O(M)`
`}`
	O(N*M)

Note: The lowercase letters a and b denote constants, and N and M denote variables.

Table 12.4 summarizes various forms of repetition and their corresponding big-O measures. For uniformity, we express all repetition with `for` loops.

Table 12.4 illustrates how nested repetition is the primary determinant of efficiency. Most of the execution time in an algorithm is spent within deeply nested loops. We will return to this issue in the next section.

12.3 Analyzing Run-Time Performance of an Algorithm

We now use the techniques from the last section to analyze a program. The `PythagoreanTriples` program, shown in Figure 12.4, outputs Pythagorean triples—three integers a, b, and c such that $a^2 + b^2 = c^2$. These represent the lengths of the sides of a right triangle. Examples of Pythagorean triples are (3, 4, 5) and (5, 12, 13). The program prompts the user to input the maximum length of a side.

Figure 12.4 `PythagoreanTriples` program

```
//----------------------------------------------------------------
//  pythag.cpp
//  This program prints all Pythagorean triples up through some user-
//  specified value.  All output triples are in increasing order.
//----------------------------------------------------------------
#include <iostream.h>
```

```
int main()
{
    int maxLen;
    int small;
    int next;
    int last;

    cout << "Max. side length: ";
    cin >> maxLen;
    small = 1;
    while (small <= maxLen) {
                    // INV (prior to test):
                    //    All triples in the range
                    //      (1..small-1, 1..maxLen, 1..maxLen)
                    //    have been output  &&  small<=maxLen+1
        next = small;
        while (next <= maxLen) {
                        // INV (prior to test):
                        //    All triples in the range
                        //      (small, 1..next-1, 1..maxLen)
                        //    have been output  &&  next<=maxLen+1
            last = next;
            while (last <= maxLen) {
                            // INV (prior to test):
                            //    All triples in the range
                            //      (small, next, 1..last-1)
                            //    have been output
                            // && last<=maxLen+1
                if (last*last == small*small + next*next)
                    cout << small << ", " << next << ", "
                         << last << '\n';
                last++;
            }
            next++;
        }
        small++;
    }
    return 0;
}
```

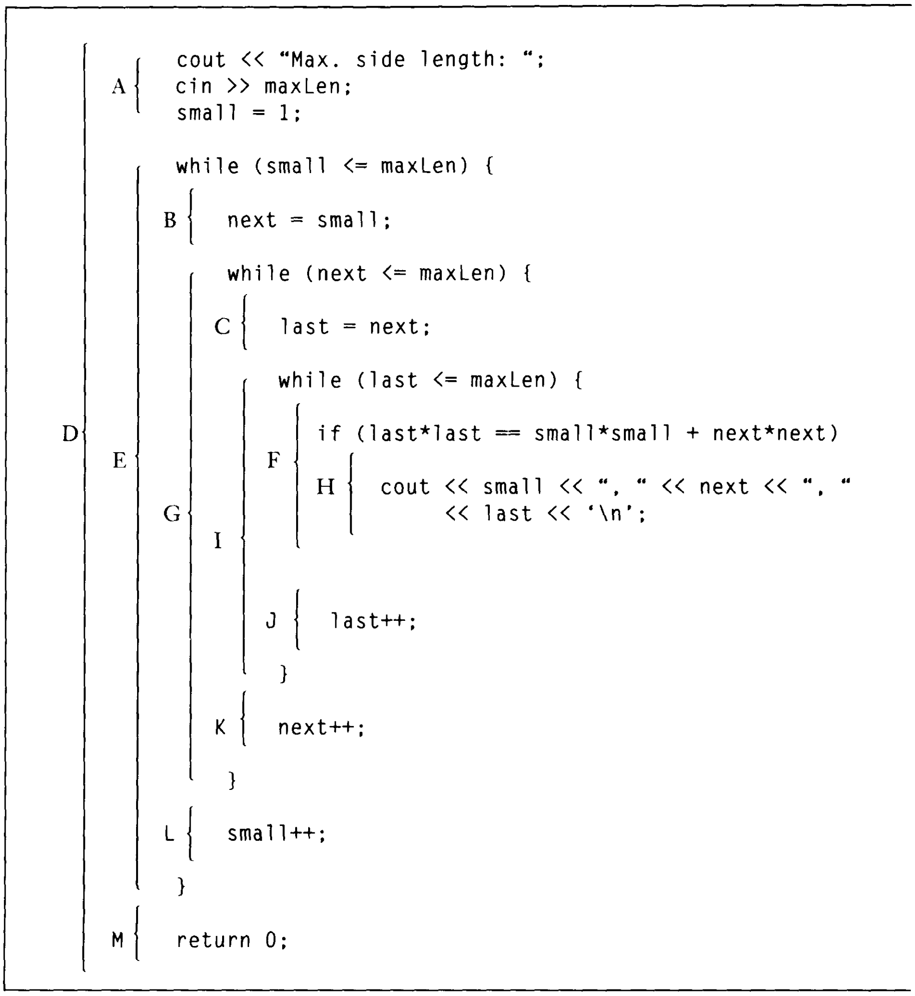
For this problem there are other, more efficient algorithms, but we present this one for its simplicity. We also use `while` instead of `for` loops to give a more exhaustive demonstration of algorithm analysis.

Figure 12.5 displays the `PythagoreanTriples` algorithm with all comments removed and with nested code sections identified by the labels A through M.

Performance analysis can proceed either bottom-up (from inner control structures to outer) or top-down (from outer control structures to inner). We begin a bottom-up analysis of `PythagoreanTriples` by examining Section H:

H {`cout << small << ", " << next << ", "  << last << '\n';`

Time for H = time to perform one output instruction
= O(1)



```
cout << "Max. side length: ";
cin >> maxLen;
small = 1;

while (small <= maxLen) {

    next = small;

    while (next <= maxLen) {

        last = next;

        while (last <= maxLen) {

            if (last*last == small*small + next*next)

                cout << small << ", " << next << ", "
                     << last << '\n';

            last++;
        }

        next++;
    }

    small++;
}

return 0;
```

Figure 12.5
PythagoreanTriples algorithm

Next, we determine the running time of F as shown below. The notation Max[*a*, *b*] denotes the maximum of *a* and *b*.

```
F {if (last*last == small*small + next*next)
       // Section H: O(1)
```

Time for F = Max[time to compute if condition, time for H]
= Max[O(1), O(1)]
= O(1)

The running time of the innermost loop, I, is determined as follows:

```
  while (last <= maxLen) {
      // Section F: O(1)
I     // Section J: O(1)
  }
```

$$\begin{aligned}\text{Time for I} &= O(\texttt{maxLen}) * \text{Max[time for F, time for J]}\\ &= O(\texttt{maxLen}) * O(1)\\ &= O(\texttt{maxLen})\end{aligned}$$

We examine next the time for the middle loop:

$$G\left\{\begin{array}{l}\texttt{while (next <= maxLen) \{}\\ \quad\text{// Section C: } O(1)\\ \quad\text{// Section I: } O(\texttt{maxLen})\\ \quad\text{// Section K: } O(1)\\ \texttt{\}}\end{array}\right.$$

$$\begin{aligned}\text{Time for G} &= O(\texttt{maxLen}) * \text{Max[time for C, time for I, time for K]}\\ &= O(\texttt{maxLen}) * O(\texttt{maxLen})\\ &= O(\texttt{maxLen}^2)\end{aligned}$$

The outer loop, E, is next:

$$E\left\{\begin{array}{l}\texttt{while (small <= maxLen) \{}\\ \quad\text{// Section B: } O(1)\\ \quad\text{// Section G: } O(\texttt{maxLen}^2)\\ \quad\text{// Section L: } O(1)\\ \texttt{\}}\end{array}\right.$$

$$\begin{aligned}\text{Time for E} &= O(\texttt{maxLen}) * \text{Max[time for B, time for G, time for L]}\\ &= O(\texttt{maxLen}) * O(\texttt{maxLen}^2)\\ &= O(\texttt{maxLen}^3)\end{aligned}$$

Finally, the running time of the entire algorithm unfolds as follows:

$$D\left\{\begin{array}{l}\text{// Section A: } O(1)\\ \text{// Section E: } O(\texttt{maxLen}^3)\\ \text{// Section M: } O(1)\end{array}\right.$$

$$\begin{aligned}\text{Time for D} &= \text{Max[time for A, time for E, time for M]}\\ &= O(\texttt{maxLen}^3)\end{aligned}$$

The `PythagoreanTriples` algorithm therefore has cubic running time.

We could also produce the same result by analyzing the code from the top down. Figure 12.6 summarizes a top-down analysis.

Figure 12.6 Top-down determination of `PythagoreanTriples` running time

Running time of D
  = Max[time for A,
        time for E,
        time for M]

```
= Max[O(1),
      O(maxLen)*Max[time for B, time for G, time for L],
      O(1)]

= Max[O(1),
      O(maxLen)*Max[O(1), time for G, O(1)],
      O(1)]

= Max[O(1),
      O(maxLen)*Max[O(1),
                    O(maxLen)*Max[time for C, time for I, time for K],
                    O(1)],
      O(1)]

= Max[O(1),
      O(maxLen)*Max[O(1),
                    O(maxLen)*Max[O(1),
                                  O(maxLen)*Max[time for F, time for J],
                                  O(1)],
                    O(1)],
      O(1)]

= Max[O(1),
      O(maxLen)*Max[O(1),
                    O(maxLen)*Max[O(1),
                                  O(maxLen)*Max[O(1), O(1)],
                                  O(1)],
                    O(1)],
      O(1)]

= Max[O(1),
      O(maxLen)*Max[O(1),
                    O(maxLen)*Max[O(1),
                                  O(maxLen)*O(1),
                                  O(1)],
                    O(1)],
      O(1)]

= Max[O(1),
      O(maxLen)*Max[O(1),
                    O(maxLen^2),
                    O(1)],
      O(1)]

= Max[O(1),
      O(maxLen^3),
      O(1)]

= O(maxLen^3)
```

Note: Max[*a*, *b*] denotes the maximum of *a* and *b*.

Designing with Wisdom

Code Efficiency Depends on Innermost Loops

Empirical testing has shown that a typical program spends 90 percent of its entire execution in repetition of only 10 percent of the code. Performance analysis points out precisely where this 10 percent often lies—the innermost loops. This repetition may take the form of nested loops or nested recursion.

A programmer can use this information to improve code efficiency in two ways. First, try to reduce the levels of nested repetition if at all possible. Second, concentrate on reducing the number of instructions executed in the innermost loops. If 90 percent of all execution time occurs in the innermost loops, then a 30 percent gain in these small sections of code yields a 27 percent (.90 × .30) overall improvement.

12.4 Searching Algorithms

Searching a vector for some specific value is one of the most frequently used computer algorithms. In this section we discuss two searching algorithms and compare them in terms of efficiency.

Each algorithm implements the abstract specification shown in Figure 12.7. The Lookup function accepts an integer vector, the size of that vector, and a particular value to search for (the **search key**). If Lookup finds the key in the vector, it returns the location of the key (as an array index) and reports success through the Boolean flag named found. If Lookup cannot find the key, it returns with found equal to FALSE.

(In the postcondition, recall that the right arrow "-->" means "implies.")

Linear Search

The most straightforward implementation of Lookup is a standard **linear search** algorithm, such as the one in Figure 12.8. A linear search probes the elements of vec in order, one after the other, until it finds the key or reaches the end of the vector. The running time of a linear search is linear—in this case, O(vSize).

```
void Lookup( /* in */  const int vec[],
             /* in */  int       vSize,
             /* in */  int       key,
             /* out */ Boolean&  found,
             /* out */ int&      loc   );

   // PRE:  vSize >= 0  &&  Assigned(vec[0..vSize-1])
   //    && Assigned(key)
   // POST: (found) --> vec[loc] == key
   //    && (NOT found) --> No vec[0..vSize-1] == key
   //                       && loc is undefined
```

Figure 12.7 Specification of Lookup function