Mathematical Preliminaries

One cannot escape the feeling that these mathematical formulae have an independent existence and an intelligence of their own, that they are wiser than we are, wiser even than their discoverers —*Heinrich Hertz*

For many students who have become interested in computer science through their experience with programming, the notion that computer science requires a strong mathematical foundation is met with a certain level of distrust or disapproval. For those students, mathematics and programming often seem to represent antithetical aspects of the science of computing; programming, after all, is usually fun, and mathematics is, to many, quite the opposite.

In many cases, however, understanding the mathematical foundations of computer science can provide practical insights which dramatically affect the programming process. Recursive programming is an important case in point. As outlined in Chapter 1, programmers often find the concept of recursion difficult primarily because they lack faith in its correctness. In attempting to supply this faith, experience is critically important. For this reason, much of this book consists of programming examples and related exercises which reinforce the skills and tactics required for recursive programming. On the other hand, the ability to prove, through a reasonably cogent mathematical argument, that a particular recursive algorithm does what it is supposed to do also serves to increase one's level of confidence in the underlying process.

This chapter addresses two separate mathematical issues that arise in any complete discussion of recursion. The first is *mathematical induction*, which provides a powerful tool for proving the correctness of certain useful formulae. The second issue is that of *computational complexity*, which is considered here in a very elementary form. Throughout the text, it will often be necessary to compare the efficiency of several different algorithms for solving a particular problem. Determining the computational complexity of each algorithm will provide a useful standard for making that comparison.

2-1 Mathematical Induction

Recursive thinking has a parallel in mathematics which is called *mathematical induction*. In both techniques, one must (1) determine a set of *simple cases* for which the proof or calculation is easily handled and (2) find an appropriate *rule* which can be repeatedly applied until the complete solution is obtained. In recursive applications, this process begins with the complex cases, and the rule successively reduces the complexity of the problem until only simple cases are left. When using induction, we tend to think of this process in the opposite direction. We start by proving the simple cases, and then use the inductive rule to derive increasingly complex results.

The nature of an inductive proof is most easily illustrated in the context of an example. In many mathematical and computer science applications, we need to compute the sum of the integers from 1 up to some maximum value N. We could certainly calculate this number by taking each number in order and adding it to a running total. Unfortunately, calculating the sum of the integers from 1 to 1000 by this method would require 1000 additions, and we would quickly tire of performing this calculation in longhand. A much easier approach involves using the mathematical formula

$$1 + 2 + 3 + \cdots + N = \frac{N(N+1)}{2}$$

This is certainly more convenient, but only if we can be assured of its correctness.

For many formulae of this sort, mathematical induction provides an ideal mechanism for proof. In general, induction is applicable whenever we are trying to prove that some formula is true for every positive number N.*

The first step in an inductive proof consists of establishing that the formula holds when N = 1. This constitutes the *base step* of an inductive proof and is quite easy in this example. Substituting 1 for N in the right-hand side of the formula and simplifying the result gives

$$\frac{1(1+1)}{2} = \frac{2}{2} = 1$$

The remaining steps in a proof by induction proceed as follows:

1. Assume that the formula is true for some arbitrary number N. This assumption is called the *inductive hypothesis*.
2. Using that hypothesis, establish that the formula holds for the number N+1.

*Mathematically speaking, induction is conventionally used to prove that some *property* holds for any positive integer N, rather than to establish the correctness of a formula. In this chapter, however, each of the examples is indeed a formula, and it is clearer to define induction in that domain.

Thus, in our current example, the inductive hypothesis consists of making the assumption that

$$1 + 2 + 3 + \cdots + N = \frac{N(N+1)}{2}$$

holds for some unspecified number N. To complete the induction proof, it is necessary to establish that

$$1 + 2 + 3 + \cdots + (N+1) = \frac{(N+1)(N+2)}{2}$$

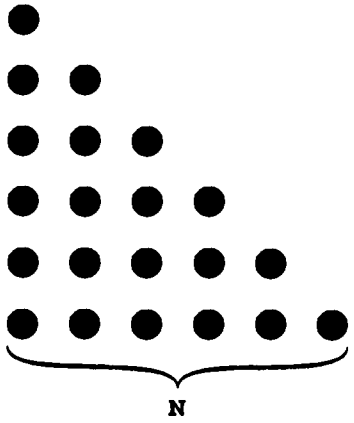
Look at the left-hand side of the expression. If we fill in the last term represented by the ellipsis (i.e., the term immediately prior to $N+1$), we get

$$1 + 2 + 3 + \cdots + N + (N+1)$$

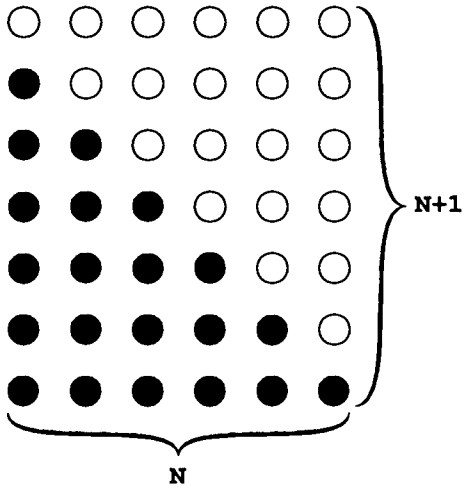
The first N terms in that sum should look somewhat familiar, since they are precisely the left-hand side of the inductive hypothesis. The key to inductive proofs is that we are allowed to use the result for N during the derivation of the $N+1$ case. Thus, we can substitute in the earlier formula and complete the derivation by simple algebra:

$$\begin{aligned}
&\underbrace{1 + 2 + 3 + \cdots + N} + (N+1) \\
&\quad \frac{N(N+1)}{2} + (N+1) \\
&= \frac{N^2 + N}{2} + \frac{2N + 2}{2} \\
&= \frac{N^2 + 3N + 2}{2} \\
&= \frac{(N+1)(N+2)}{2}
\end{aligned}$$

Even though mathematical induction provides a useful mechanism for proving the correctness of this formula, it does not offer much insight into how such a formula might be derived. This intuition often has its roots in the geometry of the structure. In this example, the successive integers can be represented as lines of dots arranged to form a triangle as shown:



Clearly, the sum of the first N integers is simply the number of dots in the triangle. As of yet, we have not simplified the problem but have merely changed its form. To determine the number of dots, we must apply some geometrical insight. If, for example, we take an identical triangle, invert it, and write it above the first triangle, we get a rectangular array of dots which has exactly twice as many dots as in the original triangle:



Fortunately, counting the number of dots in a rectangle is a considerably easier task, since the number of dots is simply the number of columns times the number of rows. Since there are N columns and N + 1 rows in this diagram, there are $N \times (N+1)$ dots in the rectangle. This gives

$$\frac{N(N+1)}{2}$$

as the number of dots in the original triangle.

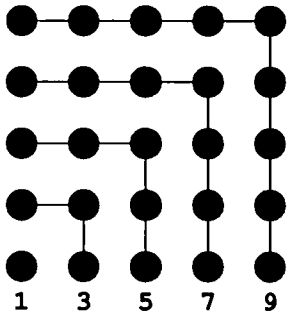
As a second example, suppose we want to find a simple formula for computing the sum of the first N odd integers:

$$1 + 3 + 5 + \cdots + (\text{Nth odd number})$$

The expression "(Nth odd number)" is a little cumbersome for mathematical manipulation and can be represented more concisely as "2N − 1".

$$1 + 3 + 5 + \cdots + (2N - 1)$$

Once again, we can gain some insight by considering a geometric representation. If we start with a single dot and add three dots to it, those three dots can be arranged to form an L shape around the original, creating a 2 × 2 square. Similarly, if we add five more dots to create a new row and column, we get a 3 × 3 square. Continuing with this pattern results in the following figure:



Since we have both an extra row and column, each new L-shaped addition to the figure requires two more dots than the previous one. Given that we started with a single dot, this process therefore corresponds to adding the next odd number each time. Using this insight, the correct formula for the sum of the first N odd numbers is simply

$$1 + 3 + 5 + \cdots + (2N - 1) = N^2$$

Although the geometric argument above can be turned into a mathematically rigorous proof, it is simpler to establish the correctness of this formula by induction. One such proof follows. Both the base step and the inductive derivation are reasonably straightforward. Give it a try yourself before turning the page.

Inductive proof for the formula

$$1 + 3 + 5 + \cdots + (2N-1) = N^2$$

Base step: $1 = 1^2$

Inductive derivation:

$$\underbrace{1 + 3 + 5 + \cdots + (2N-1)}_{N^2} + 2(N+1)-1$$
$$= N^2 + 2N + 1$$
$$= (N+1)^2$$

There are several ways to visualize the process of induction. One which is particularly compelling is to liken the process of an inductive proof to a chain of dominos which are lined up so that when one is knocked over, each of the others will follow in sequence. In order to establish that the entire chain will fall under a given set of circumstances, two things are necessary. To start with, someone has to physically knock over the first domino. This corresponds to the base step of the inductive argument. In addition, we must also know that, whenever any domino falls over, it will knock over the next domino in the chain. If we number the dominos, this requirement can be expressed by saying that whenever domino N falls, it must successfully upset domino N + 1. This corresponds to using the inductive hypothesis to establish the result for the next value of N.

More formally, we can think of induction not as a single proof, but as an arbitrarily large sequence of proofs of a similar form. For the case N = 1, the proof is given explicitly. For larger numbers, the inductive phase of the proof provides a mechanism to construct a complete proof for any larger value. For example, to prove that a particular formula is true for N = 5, we could, in principal, start with the explicit proof for N = 1 and then proceed as follows:

Since it is true for N = 1, I can prove it for N = 2.
Since I know it is true for N = 2, I can prove it for N = 3.
Since I know it is true for N = 3, I can prove it for N = 4.
Since I know it is true for N = 4, I can prove it for N = 5.

In practice, of course, we are not called upon to demonstrate a complete proof for any value, since the inductive mechanism makes it clear that such a derivation would be possible, no matter how large a value of N is chosen.

Recursive algorithms proceed in a very similar way. Suppose that we have a problem based on a numerical value for which we know the answer when N = 1. From there, all that we need is some mechanism for calculating the result for any value N in terms of the result for N − 1. Thus, to compute the solution when N = 5, we simply invert the process of the inductive derivation:

To compute the value when N = 5, I need the value when N = 4.
To compute the value when N = 4, I need the value when N = 3.
To compute the value when N = 3, I need the value when N = 2.
To compute the value when N = 2, I need the value when N = 1.
I know the value when N = 1 and can use it to solve the rest.

Both recursion and induction require a similar conceptual approach involving a "leap of faith." In writing a recursive program, we assume that the solution procedure will correctly handle any new subproblems that arise, even if we cannot see all of the details in that solution. This corresponds to making the inductive hypothesis—the point at which we assume that the formula is true for some unspecified number N. If the formula we are trying to prove is not correct, this assumption will be contrary to fact in the general case. Nonetheless, the proof technique requires us to make that assumption and hold to it until we either complete the proof or establish a contradiction.

To illustrate this, suppose that we are attempting to prove the rather dubious proposition that "all positive integers are odd." As far as the base step is concerned, everything seems fine; the number 1 is indeed odd. To continue with the proof, we still begin by making the assumption that N is odd, for some unspecified value of N. The proof does not fall apart until we use that assumption in an attempt to prove that N+1 is also odd. By the laws of arithmetic, N+1 is even whenever N is odd, and we have discovered a contradiction to the original assumption.

In the case of both recursion and induction, we have no a priori reason to believe in the truth of our assumption. If it is valid, then the program or formula will operate correctly through all its levels. However, if there is an error in the recursive decomposition or a flaw in the inductive proof, the entire structure breaks down—the domino chain is broken. This faith in the correctness of something in which we as yet have no confidence is reminiscent of Coleridge when he describes "that willing suspension of disbelief for the moment, that constitutes poetic faith." Such faith is as important to the mathematician and the programmer as it is to the poet.

2-2 Computational Complexity

A few years ago, before the advent of the video arcade and the home computer, computer games were a relatively rare treat to be found at your local science or children's museum. One of the most widely circulated was a simple Guess-the-Number game which was played as follows:

Hi! Welcome to Guess-the-Number! I'm thinking of
a number in the range 1 to 100. Try to guess it.
What is your guess? 20
That's too small, try again.

What is your guess? 83
That's too big, try again.

The game continued in this fashion, accepting new guesses from the player, until the computer's secret number was discovered.

What is your guess? 37
Congratulations! You got it in 12 guesses!

For children who were happy to spend a little extra time with one of these games, twelve guesses hardly seemed excessive. Eventually, however, even relatively young players would discover that they could do a little better than this by exploiting a more systematic strategy.

To develop such a strategy, the central idea is that each guess must narrow the range to be searched as quickly as possible. This is accomplished by choosing the value closest to the middle of the available range. For example, the original problem can be expressed in English as

Guess a number in the range 1 to 100.

If we guess 50 and discover that it is too large, we reduce this to the problem

Guess a number in the range 1 to 49.

This has the effect of reducing the original problem to an identical subproblem in which the number is limited to a more restricted range. Eventually, we must guess the correct number, since the range will get smaller and smaller until only a single possibility remains.

In the language of computer science, this algorithm is called *binary search* and is an example of the recursive "divide-and-conquer" strategy presented in Chapter 1.* For this problem, binary search seems to work reasonably well. On the other hand, it is certainly not the only possible approach. For example, when asked to find a number in the range 1 to 100, we could certainly just ask a series of questions of the form

Is it 1?
Is it 2?
Is it 3?

and so forth. We are bound to hit it eventually, after no more than 100 guesses. This algorithm is called *linear search* and is used quite frequently in computer science to find a value in an unordered list.

*As an algorithmic technique, binary search is of enormous practical importance to computer science and can be applied to many problems that are more exciting than the Guess-the-Number game. Nonetheless, Guess-the-Number provides an excellent setting for examining the algorithmic properties of the binary search technique

Intuitively, we have a sense that the binary search mechanism is a better approach to the Guess-the-Number game, but we are not sure how much better it might be. In order to have some standard for comparison, we must find a way to measure the efficiency of each algorithm. In computer science, this is most often accomplished by calculating the *computational complexity* of the algorithm, which expresses the number of operations required to solve a problem as a function of the size of that problem.

The idea that computational complexity includes a consideration of problem size should not come as much of a surprise. In general, we expect that larger problems will require more time to solve than smaller ones. For example, guessing a number in the 1 to 1000 range will presumably take more time than the equivalent problem for 1 to 100. But how much more? By expressing efficiency as a relationship between size (usually represented by N) and the number of operations required, complexity analysis provides an important insight into how a change in N affects the required computational time.

As of yet, however, the definition of complexity is somewhat less precise than we might like, since the definition of an algorithmic operation depends significantly on the presentation of the algorithm involved. For a program that has been prepared for a particular computer system, we might consider each machine-language instruction as a primitive operation. In this context, counting the number of operations corresponds precisely to counting the number of instructions executed. Unfortunately, this would result in a measure of complexity which varies from machine to machine.

Alternatively, we can adopt the more informal definition of an "operation" as a simple conceptual step. In the case of the number-guessing problem, the only operation we perform is that of guessing a number and discovering how that number stands in relation to the value we are trying to discover. Using this approach, our measure of complexity is therefore the number of guesses required.

For many algorithms, the number of operations performed is highly dependent on the data involved and may vary widely from case to case. For example, in the Guess-the-Number game, it is always possible to "get lucky" and select the correct number on the very first guess. On the other hand, this is hardly useful in estimating the overall behavior of the algorithm. Usually, we are more concerned with estimating the behavior in (1) the *average* case, which provides some insight into the typical behavior of the algorithm, and (2) the *worst* case possible, which provides an upper bound on the required time.

In the case of the linear search algorithm, each of these measures is relatively easy to analyze. In the worst possible case, guessing a number in the range 1 to N might require a full N guesses. In the specific example involving the 1 to 100 range, this occurs if the number were exactly 100. To compute the average case, we must add up the number of guesses required for each possibility and divide that total by N. The number 1 is found on the first guess, 2 requires two guesses, and so forth, up to N, which requires N guesses. The sum of these possibilities is then

$$1 + 2 + 3 + \cdots + N = \frac{N(N+1)}{2}$$

Dividing this by N gives the average number of guesses, which is

$$\frac{N+1}{2}$$

The binary search case takes a little more thought but is still reasonably straightforward. In general, each guess we make allows us to reduce the size of the problem by a factor of two. For example, if we guess 50 in the 1 to 100 example and discover that our guess is low, we can immediately eliminate the values in the 1 to 50 range from consideration. Thus, the first guess reduces the number of possibilities to N/2, the second to N/4, and so on. Although, in some instances, we might get lucky and guess the exact value at some point in the process, the worst case will require continuing this process until there is only one possibility left. The number of steps required to accomplish this is illustrated by the diagram

$$\underbrace{N / 2 / 2 / 2 \cdots / 2}_{k \text{ times}} = 1$$

where k indicates the number of guesses required. Simplifying this gives

$$\frac{N}{2^k} = 1$$

or

$$N = 2^k$$

Since we want an expression for k in terms of the value of N, we must use the definition of logarithms to turn this around.

$$k = \log_2 N$$

Thus, in the worst case, the number of steps required to guess a number through binary search is the base-2 logarithm of the number of values.* In the average case, we can expect to find the correct value using one less guess (see Exercise 2-5).

*In analyzing the complexity of an algorithm, we frequently will make use of logarithms which, in almost all instances, are calculated using 2 as the logarithmic base. For the rest of this text, we will follow the standard convention in computer science and use "log N" to indicate base 2 logarithms without explicitly writing down the base.

Estimates of computational complexity are most often used to provide insight into the behavior of an algorithm as the size of the problem grows large. Here, for example, we can use the worst-case formula to create a table showing the number of guesses required for the linear and binary search algorithms, respectively:

N	Linear search	Binary search
10	10	4
100	100	7
1,000	1,000	10
10,000	10,000	14
100,000	100,000	17
1,000,000	1,000,000	20

This table demonstrates conclusively the value of binary search. The difference between the algorithms becomes increasingly pronounced as N takes on larger values. For ten values, binary search will yield the result in no more than four guesses. Since linear search requires ten, the binary search method represents a factor of 2.5 increase in efficiency. For 1,000,000 values, on the other hand, this factor has increased to one of 50,000. Surely this represents an enormous improvement.

In the case of the search algorithms presented above, the mechanics of each operation are sufficiently simple that we can carry out the necessary computations in a reasonably exact form. Often, particularly when analyzing specific computer programs, we are forced to work with approximations rather than exact values. Fortunately, those approximations turn out to be equally useful in terms of predicting relative performance.

For example, consider the following nested loop structure in Pascal:

```
FOR I := 1 TO N DO
  FOR J := 1 TO I DO
    A[I,J] := 0
```

The effect of this statement, given a two-dimensional array A, is to set each element on or below the main diagonal to zero. At this point, however, our concern is not with the purpose of the program but with its computational efficiency. In particular, how long would this program take to run, given a specific matrix of size N?

As a first approximation to the running time, we should count the number of times the statement

```
A[I,J] := 0
```

is executed. On the first cycle of the outer loop, when I is equal to 1, the inner

loop will be executed only once for **J** = 1. On the second cycle, **J** will run through both the values 1 and 2, contributing two more assignments. On the last cycle, **J** will range through all N values in the 1 to N range. Thus, the total number of assignments to some element **A[I,J]** is given by the formula

$$1 + 2 + 3 + \cdots + N$$

Fortunately, we have seen this before in the discussion of mathematical induction and know that this may be simplified to

$$\frac{N(N+1)}{2}$$

or, expressing the result in polynomial form,

$$\tfrac{1}{2}N^2 + \tfrac{1}{2}N$$

Although this count is accurate in terms of the number of assignments, it can be used only as a rough approximation to the total execution time since it ignores the other operations that are necessary, for example, to control the loop processes. Nonetheless, it can be quite useful as a tool for predicting the efficiency of the loop operation as N grows large. Once again, it helps to make a table showing the number of assignments performed for various values of N.

N	Assignments to A[I,J]
10	55
100	5,050
1,000	500,500
10,000	50,005,000

From this table, we recognize that the number of assignments grows much more quickly than N. Whenever we multiply the size of the problem by ten, the number of assignments jumps by a factor of nearly 100.

The table also illustrates another important property of the formula

$$\tfrac{1}{2}N^2 + \tfrac{1}{2}N$$

As N increases, the contribution of the second term in the formula decreases in importance. Since this formula serves only as an approximation, it was probably rather silly to write 50,005,000 as the last entry in the table. Certainly, 50,000,000 is close enough for all practical purposes. As long as N is relatively

large, the first term will always be much larger than the second. Mathematically, this is often indicated by writing

$$N^2 >> N$$

The symbol ">>" is read as "dominates" and indicates that the term on the right is insignificant compared with the term on the left, whenever the value of N is sufficiently large. In more formal terms, this relationship implies that

$$\lim_{N \to \infty} \frac{N}{N^2} = 0$$

Since our principal interest is usually in the behavior of the algorithm for large values of N, we can simplify the formula and say that the nested loop structure runs in a number of steps roughly equal to

$$\tfrac{1}{2}N^2$$

Conventionally, however, computer scientists will apply yet another simplification here. Although it is occasionally useful to have the additional precision provided by the above formula, we gain a great deal of insight into the behavior of this algorithm by knowing that it requires a number of steps *proportional* to

$$N^2$$

For example, this tells us that if we double N, we should expect a fourfold increase in running time. Similarly, a factor of ten increase in N should increase the overall time by a factor of 100. This is indeed the behavior we observe in the table, and it depends only on the

$$N^2$$

component and not any other aspect of the complexity formula.

In computer science, this "proportional" form of complexity measure is used so often that it has acquired its own notational form. All the simplifications that were introduced above can be summarized by writing that the Pascal statement

```
FOR I := 1 TO N DO
  FOR J := 1 TO I DO
    A[I,J] := 0
```

has a computational complexity of

$$O(N^2)$$

This notation is read either as "big-O of N squared" or, somewhat more simply, "order N squared."

Formally, saying that a particular algorithm runs in time

$$O(f(N))$$

for some function f(N) means that, as long as N is sufficiently large, the time required to perform that algorithm is never larger than

$$C \times f(N)$$

for some unspecified constant C.

In practice, certain orders of complexity tend to arise quite frequently. Several common algorithmic complexity measures are given in Table 2-1. In the table, the last column indicates the name which is conventionally used to refer to algorithms in that class. For example, the linear search algorithm runs, not surprisingly, in linear time. Similarly, the nested Pascal loop is an example of a quadratic algorithm.

Table 2-1. Common Complexity Characteristics

Given an algorithm of this complexity	When N doubles, the running time	Conventional name
$O(1)$	Does not change	Constant
$O(\log N)$	Increases by a small constant	Logarithmic
$O(N)$	Doubles	Linear
$O(N \log N)$	Slightly more than doubles	N log N
$O(N^2)$	Increases by a factor of 4	Quadratic
$O(N^k)$	Increases by a factor of 2^k	Polynomial
$O(\alpha^N)$ $\alpha > 1$	Depends on α, but grows very fast	Exponential

The most important characteristic of complexity calculation is given by the center column in the table. This column indicates how the performance of the algorithm is affected by a doubling of the problem size. For example, if we

double the value of N for a quadratic algorithm, the run time would increase by a factor of four. If we were somehow able to redesign that algorithm to run in time N log N, doubling the size of the input data would have a less drastic effect. The new algorithm would still require more time for the larger problem, but the increase would be only a factor of two over the smaller problem (plus some constant amount of time contributed by the logarithmic term). If N grows even larger, this constitutes an enormous savings in time.

In attempting to improve the performance of almost any program, the greatest potential savings come from improving the algorithm so that its complexity bound is reduced. "Tweaking" the code so that a few instructions are eliminated along the way can only provide a small percentage increase in performance. On the other hand, changing the algorithm offers an unlimited reduction in the running time. For small problems, the time saved may be relatively minor, but the percentage savings grows much larger as the size of the problem increases. In many practical settings, algorithmic improvement can reduce the time requirements of a program by factors of hundreds or thousands—clearly an impressive efficiency gain.

Bibliographic Notes

One of the best discussions of mathematical induction and how to use it is contained in Solow's *How to Read and Do Proofs* [1982]. The classic text in this area is Polya [1957]. A more formal discussion of asymptotic complexity and the use of the big-O notation may be found in Knuth [1973] or Sedgewick [1983].

The inductive argument used in Exercise 2-6 to "prove" that all horses are the same color is adapted from Joel Cohen's essay "On the Nature of Mathematical Proofs" [1961].

Exercises

2-1. Prove, using mathematical induction, that the sum of the first N even integers is given by the formula

$$N^2 + N$$

How could you predict this expression using the other formulae developed in this chapter?

2-2. Establish that the following formulae are correct using mathematical induction:

(a) $1 + 2 + 4 + 8 + \cdots + 2^{N-1} = 2^N - 1$

(b) $1 + 3 + 9 + 27 + \cdots + 3^N = \frac{3^{N+1} - 1}{2}$

(c) $1\times1 + 2\times2 + 3\times4 + 4\times8 + \cdots + N\times2^{N-1} = (N-1)\,2^N + 1$

2-3. The year: 1777. The setting: General Washington's camp somewhere in the colonies. The British have been shelling the Revolutionary forces with a large cannon within their camp. You have been assigned a dangerous reconnaissance mission—to infiltrate the enemy camp and determine the amount of ammunition available for that cannon.

Fortunately for you, the British (being relatively neat and orderly) have stacked the cannonballs into a single pyramid-shaped stack. At the top is a single cannonball resting on a square of four cannonballs, which is itself resting on a square of nine cannonballs, and so forth. Given the danger in your situation, you only have a chance to count the number of layers before you escape back to your own encampment.

Using mathematical induction, prove that, if N is the number of layers, the total number of cannonballs is given by

$$\frac{N(N+1)(2N+1)}{6}$$

2-4. Assuming that N is an integer which defines the problem size, what is the order of each of the program fragments shown below:

(a)
```
K := 0;
FOR I := 1 TO 1000 DO
  K := K + I * N;
```

(b)
```
K := 0;
FOR I := 1 TO N DO
  FOR J := I TO N DO
    K := K + I * J;
```

(c)
```
K := 0;
WHILE N > 0 DO
  BEGIN
    N := N DIV 2;
    K := K + 1
  END;
```

Assuming that the statements within the loop body take the same amount of time, for what values of N will program (a) run more quickly than program (c)?

2-5. [For the mathematically inclined] In trying to locate a number by binary

search, it is always possible that it will require many fewer guesses than would be required in the worst possible case. For example, if 50 were in fact the correct number in the 1 to 100 range, binary search would find it on the very first guess. To determine the average-case behavior of binary search, we need to determine the expected value of the number of guesses over the entire possible range.

Assuming that there are N numbers in the complete range, we know that only one of them (specifically, the number at the center of the range) will be guessed on the very first try. Two numbers will be guessed in two tries, four numbers in three tries, and so forth. Thus, the average number of guesses required is given by the formula

$$\frac{1\times 1 + 2\times 2 + 3\times 4 + 4\times 8 + \cdots + G\times 2^{G-1}}{N}$$

where G is the maximum number of guesses that might be required, which in this case is simply log N.

Using the expression from Exercise 2-2(c), simplify this formula. As N grows large, what does this approach?

2-6. Mathematical induction can have its pitfalls, particularly if one is careless. For example, the following argument appears to be a proof that all horses are the same color. What is wrong here? Is there really no horse of a different color?

Preliminary Define a set of horses to be *monochromatic* if all the horses in the set have the same coloration.

Conjecture Any set of horses is monochromatic.

Technique Proof by induction on the number of horses.

Base step Any set of one horse is monochromatic, by definition.

Induction Assume that any set of N horses is monochromatic. Consider a set of N + 1 horses. That can be divided into smaller subsets in several ways. For example, consider the division indicated in the following diagram:

$$\underbrace{H_1H_2\cdots H_N}_{A}\ \underbrace{H_{N+1}}_{A'}$$

The subset labeled A in this diagram is a set of N horses and is therefore monochromatic by the inductive hypothesis.

Similarly, if we divide the complete set as follows:

$$\underbrace{H_1}_{B'}\ \underbrace{H_2 \cdots H_N H_{N+1}}_{B}$$

we get a subset B which is also monochromatic.

Thus, all the horses in subset A are the same color as are all the horses in subset B. But H_2 is in both subsets A and B, which implies that both subsets must contain horses of the *same* color.