# Chapter 2: Algorithms

A computer must be given a series of commands, termed a *program*, in order to make it perform a useful task. Because programs can be large, they are divided into smaller units, termed (depending upon your programming language), functions, procedures, methods, subroutines, subprograms, classes, or packages. In programs that you have written you have undoubtedly used some of these features.

A function, for example, is used to package a task to be performed. Let us use as an example a function to compute the square root of a floating-point number. Among other benefits, the function is providing an *abstraction*. By this we mean that from the outside, from the point of view of the person calling the function, they only need to know the name of the action to be performed. On the other hand, inside the function the programmer must have written the exact series of instructions needed to perform the task. Since to use the function you do not need to know the internal details, we say that the function is a form of *encapsulation*.

```
double sqrt (double val) {
    /* return square root of argument */
    /* works only for positive values */
    assert (val >= 0);
    double guess = val / 2.0;
    …
    return guess;
}
```

The difference between the information necessary to *use* a function and the information necessary to *write* a function, the fact that you can use a function without knowing how it is implemented, is one of the most important tools used to manage complex systems. In your own programming experience you have undoubtedly used libraries of functions, for performing tasks such as input and output, or manipulating graphical interfaces. You can use these functions with only a minimal knowledge of how they go about their task. This makes developing programs infinitely easier than if every programmer had to write all of these actions from scratch.

There is another level of abstraction that is equally important. Let us again illustrate this using the function to compute a square root. In order to make this function available for your use, at some point in the past there must have been a programmer who developed the code that you invoke when you call the square root function. Although this programmer may never have written this particular function, they did not start entirely from scratch. Fortunately for this programmer, there are well known techniques that can be used to solve this problem. In this case, the technique was most likely something termed Newton's Method. This technique, discovered by Issac Newton (1643-1727), finds a square root by first making a guess, then using the first guess to find a better guess, and so on until the correct answer is located.

Newton, of course, lived well before the advent of computers, and so never wrote his technique as a computer function. Instead, he described an *algorithm*. An algorithm is a description of how a specific problem can be solved, written at a level of detail that can

be followed by the reader. Terms that have a related meaning include process, routine, technique, procedure, pattern, and recipe.

It is important that you understand the distinction between an *algorithm* and a *function*. A function is a set of instructions written in a particular programming language. A function will often embody an algorithm, but the essence of the algorithm transcends the function, transcends even the programming language. Newton described, for example, the way to find square roots, and the details of the algorithm will remain the same no matter what programming language is used to write the code.

## Properties of Algorithms

Once you have an algorithm, to solve a problem is simply a matter of executing each instruction of the algorithm in turn. If this process is to be successful there are several characteristics an algorithm must possess. Among these are:

**input preconditions**. The most common form of algorithm is a transformation that takes a set of input values and performs some manipulations to yield a set of output values. (For example, taking a positive number as input, and returning a square root). An algorithm must make clear the number and type of input values, and the essential initial conditions those input values must possess to achieve successful operation. For example, Newtons method works only if the input number is larger than zero.

**precise specification of each instruction**. Each step of an algorithm must be well defined. There should be no ambiguity about the actions to be carried out at any point. Algorithms presented in an informal descriptive form are sometimes ill-defined for this reason, due to the ambiguities in English and other natural languages.

**correctness**. An algorithm is expected to solve a problem. For any putative algorithm, we must demonstrate that, in fact, the algorithm will solve the problem. Often this will take the form of an argument, mathematical or logical in nature, to the effect that if the input conditions are satisfied and the steps of the algorithm executed then the desired outcome will be produced.

**termination, time to execute**. It must be clear that for any input values the algorithm is guaranteed to terminate after a finite number of steps. We postpone until later a more precise definition of the informal term "steps."' It is usually not necessary to know the exact number of steps an algorithm will require, but it will be convenient to provide an upper bound and argue that the algorithm will always terminate in fewer steps than the upper bound. Usually this upper bound will be given as a function of some values in the input. For example, if the input consists of two integer values n and m, we might be able to say that a particular algorithm will always terminate in fewer than n + m steps.

**description of the result or effect**. Finally, it must be clear exactly what the algorithm is intended to accomplish. Most often this can be expressed as the production of a result value having certain properties. Less frequently algorithms are executed for a *side effect*,

such as printing a value on an output device. In either case, the expected outcome must be completely specified.

Human beings are generally much more forgiving than computers about details such as instruction precision, inputs, or results. Consider the request "Go to the store and buy something to make lunch". How many ways could this statement be interpreted? Is the request to buy ingredients (for example, bread and peanut butter), or for some sort of mechanical "lunch making" device (perhaps a personal robot). Has the input or expected result been clearly identified? Would you be surprised if the person you told this to tried to go to a store and purchase an automated lunch-making robot? When dealing with a computer, every step of the process necessary to achieve a goal must be outlined in detail.

A form of algorithm that most people have seen is a recipe. In worksheet 1 you will examine one such algorithm, and critique it using the categories given above. Creating algorithms that provide the right level of detail, not too abstract, but also not too detailed, can be difficult. In worksheet 2 you are asked to describe an activity that you perform every day, for example getting ready to go to school. You are then asked to exchange your description with another student, who will critique your algorithm using these objectives.

## Specification of Input

An algorithm will, in general, produce a result only when it is used in a proper fashion. Programs use various different techniques to ensure that the input is acceptable for the algorithm. The simplest of these is the idea of *types*, and a function type signature. The function shown at right, for instance, returns the smaller of two integer values. The compiler can check that when you call the function the arguments are integer, and that the result is an integer value.

```
int min (int a, int b) {
    /* return smaller argument */
    if (a < b) return a;
    else return b;
}
```

Some requirements cannot be captured by type signatures. A common example is a range restriction. For instance, the square root program we discussed earlier only works for positive numbers. Typically programs will check the range of their input at run-time, and issue an error or exception if the input is not correct.

## Description of Result

Just as the input conditions for computer programs are specified in a number of ways, the expected results of execution are documented in a variety of fashions. The most obvious is the result type, defined as part of a function signature. But this only specifies the type of the result, and not the relationship to the inputs. So equally important is some sort of documentation, frequently written as a comment. In the min function given earlier it is noted that the result is not only an integer, but it also represents the smaller of the two input values.

## Instruction Precision

Algorithms must be specified in a form that can be followed to produce the desired outcome. In this book we will generally present algorithms in a high level pseudo-code, a form that can easily be translated into a working program in a given programming language. Using a pseudo-code form allows us to emphasize the important properties of the algorithm, and downplay incidental details that, while they may be necessary for a working program, do nothing to assist in the understanding of the procedure.

## Time to execute

Traditionally the discussion of execution time is divided into two separate questions. The first question is showing that the algorithm will terminate for all legal input values. Once this question has been settled, the next question is to provide a more accurate characterization of the amount of time it will take to execute. This second question will be considered in more detail in a later section. Here, we will address the first.

The basic tool used to prove that an algorithm terminates is to find a property or value that satisfies the following three characteristics:

1. The property or value can be placed into a correspondence with integer values.
2. The property is nonnegative.
3. The property or value decreases steadily as the algorithm executes.

The majority of the time this value is obvious, and is left unstated. For instance, if an algorithm operates by examining each element of an array in turn, we can use the remaining size of the array (that is, the unexplored elements) as the property we seek. It is only in rare occasions that an explicit argument must be provided.

An example in which termination may not be immediately obvious occurs in Euclid's Greatest Common Divisor algorithm, first developed in third century B.C. Greece. Donald Knuth, a computer scientist who has done a great deal of research into the history of algorithms, has claimed that this is the earliest-known nontrivial completely specified mathematical algorithm. The algorithm is intended to compute, given two positive integer values n and m, the largest positive integer that evenly divides both values. The algorithm can be written as shown at right.

```
int gcd (int n, int m) {
        /* compute the greatest common divisor
        of two positive integer values */
        while (m != n) {
                if (n > m)
                        n = n - m;
                else
                        m = m - n;
        }
        return n;
}
```

Because there is no definite limit on the number of iterations, termination is not obvious. But we can note that either n or m

becomes smaller on each iteration, and so the *sum* n+m satisfies our three conditions, and hence can be used to demonstrate that the function will halt.

---

**Zeno's Paradox**

The most nonintuitive of the three properties required for proving termination is the first: that the quantity or property being discussed can be placed into a correspondence with a diminishing list of integers. Why integer? Why not just numeric? Why diminishing? To illustrate the necessity of this requirement, consider the following "proof" that it is possible to share a single candy bar with all your friends, no matter how many friends you may have. Take a candy bar and cut it in half, giving one half to your best friend. Take the remaining half and cut it in half once more, giving a portion to your second best friend. Assuming you have a sufficiently sharp knife, you can continue in this manner, at each step producing ever-smaller pieces, for as long as you like. Thus, we have no guarantee of termination for this process. Certainly the sequence here (1/2, ¼, 1/8, 1/16 …) consists of all nonnegative terms, and is decreasing. But a correspondence with the integers would need to increase, not decrease; and any diminishing sequence would be numeric, but not integer.

In the 5th century B.C. the Greek philosopher Zeno used similar arguments to show that no matter how fast he runs, the famous hero Achilles cannot overtake a Tortoise, if the Tortoise is given a head start. Suppose, for example, that the Tortoise and Achilles start as below at time A. When, at time B, Achilles reaches the point from which the Tortoise started, the Tortoise will have proceeded some distance ahead of this point. When Achilles reaches this further point at time C, the Tortoise will have proceeded yet further ahead. At each point when Achilles reaches a point from which the Tortoise began, the Tortoise will have proceeded at least some distance further. Thus, it was impossible for Achilles to ever overtake and pass the Tortoise.

Achiles          A ->  B ->  C ->  D
Tortoise                A ->  B ->  C ->  D

The paradox arises due to an infinite sequence of ever-smaller quantities, and the nonintuitive possibility that an infinite sum can nevertheless be bounded by a finite value. By restricting our arguments concerning termination to use only a decreasing set of *integer* values, we avoid problems such as those typified by Zeno's paradox.

---

## Recursive Algorithms

Recursion is a very common technique used for both algorithms and the functions that result from implementing algorithms. The basic idea of recursion is to "reduce" a complex problem by "simplifying" it in some way, then invoking the algorithm on the simpler problem. This is performed repeated until one or more special cases, termed *base*

*cases*, are encountered. Arguments that do not correspond to base cases are called *recursive cases*, or sometimes *inductive cases*.

Recursion is a common theme in Art or commercial graphics. A seemingly infinite series of objects can be generated by looking at two opposing mirrors, for example. The image on box of a popular brand of cocoa shows a woman holding a tray of cocoa, including a box of cocoa, on which is found a picture of a woman holding a tray of cocoa, and so on.

You have undoubtedly seen mathematical definitions that are defined in a recursive fashion. The exponential function, for instance, is traditionally defined as follows:

$$N! = N * (N-1)! \quad \text{for all values } N > 0$$
$$0! = 1$$
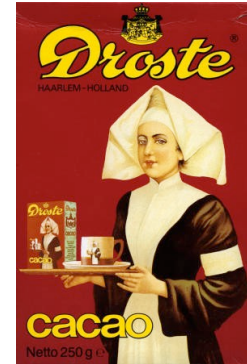
Here zero is being used as the base case, and the simplification consists of subtracting one, then invoking the factorial on the smaller number. The definition suggests an obvious algorithm for computing the factorial.

**How to compute the factorial of N**. If N is less than zero, issue an error. Otherwise, if N is equal to zero, return 1. Otherwise, compute the factorial of (N-1), then multiply this result by N and return the result.

```
int factorial (int N) {
    assert (N >= 0);
    if (N == 0) return 1;
    return N * factorial (N − 1);
}
```

The box at right shows this algorithm expressed as a function. To prove termination we would use the same technique as before; that is, identify a value that is integer, decreasing, and non-negative. In this case the value n itself suffices for this argument.

Many algorithms are most easily expressed in a recursive form. The base case need not be a single value, it can sometimes be a condition that the argument must satisfy. A simple example that illustrates this is printing a decimal number, such as 4973. It is relatively easy to handle the single digits, zero to 9, as special cases. Here there are ten base cases. For larger numbers, recursively print the value you get by dividing the number by ten, then print the single digit you get when you take the remainder divided by ten. In our example printing 4973 would recursively call itself to print 497, which would in turn recursively call itself to print 49, which would yet again recursively call itself to print 4. The final call results in one of our base cases, so 4 would be printed. Once this function returns the value 9 is printed, which is the remainder left after dividing 49 by ten. Once this call returns the value 7 would be printed, which is the remainder left after dividing 497 by ten. And finally the value 3 would be printed, which is the remainder left after diving 4973 by ten.

```
void printInteger (int n) {
    assert (n > 0);
    if (n > 9)
        printInteger (n / 10);
    printDigit (n % 10);
}
```

The function resulting from this algorithm is shown at left. Here the base cases have been hidden in another function,

printDigit, that handles only the values zero to nine. In trying to understand how a recursive function can work, it is important to remember that every time a procedure is invoked, new space is allocated for parameters and local variables. So each time the function printInteger is called, a new value of n is created. Previous values of n are left pending, stacked up so that they can be restored after the function returns. This is called the *activation record stack*. A portion of the activation record stack is shown at right, a snapshot showing the various values of n that are pending when the value 4 is printed at the end of the series of recursive calls. Once the latest version of printInteger finishes the caller, who just happens to be printInteger, will be restarted. But in the caller the value of variable n will have been restored to the value 49.

| N = 4 |
| N = 49 |
| N = 497 |
| N = 4973 |

A classic example of a recursive algorithm is the puzzle known as the *Towers of Hanoi*. In this puzzle, three poles are labeled A, B and C. A number of disks of decreasing sizes are, initially, all on pole A.
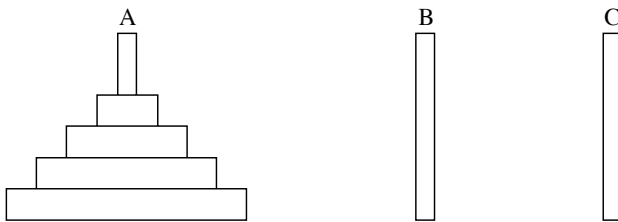


The goal of the puzzle is to move all disks from pole A to pole B, without ever moving a disk onto another disk with a smaller size. The third pole can be used as a temporary during this process. At any point, only the topmost disk from any pole may be moved.
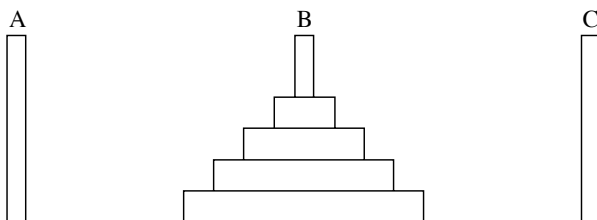


It is obvious that only two first steps are legal. These are moving the smallest disk from pole A to pole B, or moving the smallest disk from pole A to pole C. But which of these is the correct first move?

As is often the case with recursive algorithms, a greater insight is found by considering a point in the middle of solving the problem, rather that thinking about how to begin. For example, consider the point we would like to be one step before we move the largest disk. For this to be a legal move, we must have already moved all the disks except for the largest from pole A to pole C.
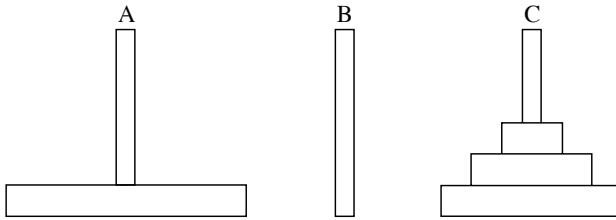
Notice that in this picture we have moved all but the last disk to pole C. Once we move the largest disk, pole A will be empty, and we can use it as a temporary. This observation hints at how to express the solution to the problem as a recursive algorithm.

**How to move N disks from pole A to pole B using pole C as a temporary:** If N is 1, move disk from A to B. Otherwise, move (N – 1) disks from pole A to pole C using pole B as a temporary. Move the largest disk from pole A to pole B. Then move disks from pole C to pole B, using pole A as a temporary.

```
void solveHanoi (int n, char a, char b, char c) {
  if (n == 1) print ("move disk from pole ", a, " to pole ", b);
  else {
    solveHanoi (n – 1, a, c, b);
    print ("move disk from pole ", a, " to pole ", b);
    solveHanoi (n – 1, c, b, a);
  }
}
```

We could express this algorithm as pseudo-code as shown. Here the arguments are used to represent the size of the stack and the names of the poles, and the instructions for solving the puzzle are printed.

We will see many recursive algorithms as we proceed through the text.

## Study Questions

1. What is abstraction? Explain how a function is a form of abstraction.

2. What is an algorithm?

3. What is the relationship between an algorithm and a function?

4. Give an example input condition that cannot be specified using only a type.

5. What are two different techniques used to specify the input conditions for an algorithm?

6. What are some ways used to describe the outcome, or result, of executing an algorithm?

7. In what way does the precision of instructions needed to convey an algorithm to another human being differ from that needed to convey an algorithm to a computer?

8. In considering the execution time of algorithms, what are the two general types of questions one can ask?

9. What are some situations in which termination of an algorithm would not be immediately obvious?

10. What are the three properties a value must possess in order to be used to prove termination of an algorithm?

11. What is a recursive algorithm? What are the two sections of a recursive algorithm?

12. What is the activation record stack? What values are stored on the activation record stack? How does this stack simplify the execution of recursive functions?

## Analysis Exercises

1. Examine a recipe from your favorite cookbook. Evaluate the recipe with respect to each of the characteristics described at the beginning of this chapter.

2. This chapter described how termination of an algorithm can be demonstrated by finding a value or property that satisfies three specific conditions. Show that all three conditions are necessary. Do this by describing situations that satisfy two of the three, but not the third, and that do not terminate. For example, the text explained how you can share a single candy bar with an infinite number of friends, by repeatedly dividing the candy bar in half. What property does this algorithm violate?

3. The version of the towers of Hanoi used a stack of size 1 as the base case. Another possibility is to use a stack of size zero. What actions need to be performed to move a stack of size zero from one pole to the next? Rewrite the algorithm to use this formulation.

4. What integer value suffices to demonstrate that the towers of Hanoi algorithm must eventually terminate for any size tower?

## Exercises

1. Assuming you have the ability to concatenate a character and a string, describe in pseudo-code a recursive algorithm to reverse a string. For example, the input "function" should produce the output "noitcnuf".

2. Express the algorithm to compute the value a raised to the $n^{th}$ power as a recursive algorithm. What are your input conditions? What is your base case?

3. Binomial coefficients (the number of ways that n elements can be selected out of a collection of m values) can be defined recursively by the following formula:

picture

Write a recursive procedure comb (n, m) to compute this value.

4.  Rewrite the GCD algorithm as a recursive procedure.

5.  List the sequence of moved the Towers of Hanoi problem would make in moving a tower of four disks from the source pole to the destination pole.

6.  The *Fibonacci sequence* is another famous series that is defined recursively. The definition is as follows:

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Write in pseudo-code a function to compute the $n^{th}$ Fibonacci number.

7.  A *palindrome* is a word that reads the same both forward and backwards, such as rotor. An algorithm to determine whether or not a word is a palindrome can be expressed recursively. Simply strip off the first and last letters; if they are different, the word is not a palindrome. If they are, test the remaining string (after the first and last letters have been removed) to see if it is a palindrome. What is your base case for this procedure?

8.  Extend the procedure you wrote in the previous question so that it can handle (by ignoring them) spaces and punctuation. An example of this sort is "A man, a plan, a canal, panama!"

9.  Describe in pseudo-code, but do not implement, a recursive algorithm for finding the smallest number in an Array between two integer bounds (that is, a typical call such as smallest(a, 3, 7) would find the smallest value in the array a among those elements with indices between 3 and 7. What is the base case for your algorithm? How does the inductive case simplify the problem?

10. Using the algorithm you developed in the preceding question describe in pseudo-code a recursive sorting algorithm that works by finding the smallest element, swapping it into the first position, then finding the smallest element in the remainder, swapping it into the next position, and so on. What is the base case for your algorithm?


## Programming Projects

1.  Write a recursive procedure to print the octal (base-8) representation of an integer value.

2. Write a program that takes an integer argument and prints the value spelled out as English words. For example, the input - 3472 would produce the output "negative three thousand four hundred seventy-two". After removing any initial negative signs, this program is most easily handled in a recursive fashion. Numbers greater than millions can be printed by printing the number of millions (via a recursive call), then printing the work "million", then printing the remainder. Similarly with thousands, hundreds, and most values larger than twenty. Base cases are zero, numbers between 1 and 20.

3. Write a program to compute the fibonocci sequence recursively (see earlier exercise). Include a global variable fibCount that is incremented every time the function calls itself recursively. Using this, determine the number of calls for various values of N, such as n from 1 to 20. Can you determine a relationship between n and the resulting number of calls?

4. Do the same sort of analysis as described in the preceding question, but this time for the towers of Hanoi.

## On the Web

Wikipedia (http://en.wikipedia.org) has a detailed entry for Algorithm that provides a history of the word, and several examples illustrating the way algorithms can be presented. The GCD algorithm is described in an entry "Euclidian Algorithm". Another interesting entry can be found on "recursive algorithms". The entry on "Fibonacci Numbers" provides an interesting history of the sequence, as well as many of the mathematical relationships of these numbers. The entry of "Towers of Hanoi" includes an animation that demonstrates the solution for N equal to 4. The more mathematically inclined might want to explore the entries on "Fractals" and "Mathematical Induction", and the relationship of these to recursion. Newtons method of computing square roots is described in an entry on "Methods of computing square roots". Another example of recursive art in commercial advertising is the "Morton Salt girl", who is spilling a container of salt, on which is displayed a picture of the Morton salt girl.