Unit 5: Recursive Thinking

Topics:
I. Recursion
II. Computational limits
III. Recursion in graphics

Materials:
I. Hein ch. 3.2
II. Rawlins: Towers of Hanoi
III. Lewis & Loftus: Fractals
IV. Recursion exercises #1
V. Recursion exercises #2
VI. Graphical recursion exercises
VII. Challenge questions
VIII. Tree Fractal specifications
IX. Snowflake Fractal specifications
X. Labyrinth specifications
XI. Review
18. Find an inductive definition for each product set $S$.
   a. $S = \text{lists}(A) \times \text{lists}(A)$ for some set $A$.
   b. $S = A \times \text{lists}(A)$.
   c. $S = N \times \text{lists}(N)$.
   d. $S = N \times N \times N$

Proofs and Challenges

19. Let $A$ be a set. Suppose $O$ is the set of binary trees over $A$ that contain an odd number of nodes. Similarly, let $E$ be the set of binary trees over $A$ that contain an even number of nodes. Find inductive definitions for $O$ and $E$.
   
   Hint: You can use $O$ when defining $E$, and you can use $E$ when defining $O$.

20. Use Example 3.15 as a guide to construct an inductive definition for the set of points in $N \times N$ that describe the area $A$ between two curves $f$ and $g$ defined as follows for two natural numbers $a$ and $b$:

$$A = \{(x, y) \mid x, y \in N, a \leq x \leq b, \text{ and } g(x) \leq y \leq f(x)\}.$$  

21. Prove that a set defined by (3.1) is countable if the basis elements in Step 1 are countable, the outside elements used in Step 2 are countable, and the rules specified in Step 2 are finite.

3.2 Recursive Functions and Procedures

Since we’re going to be constructing functions and procedures in this section, we’d better agree on the idea of a procedure. From a computer science point of view, a procedure is a program that performs one or more actions. So there is no requirement to return a specific value. For example, the execution of a statement like $\text{print}(x, y)$ will cause the values of $x$ and $y$ to be printed. In this case, two actions are performed, and no values are returned. A procedure may also return one or more values through its argument list. For example, a statement like $\text{allocate}(m, a, s)$ might perform the action of allocating a block of $m$ memory cells and return the values $a$ and $s$, where $a$ is the beginning address of the block and $s$ tells whether the allocation was successful.

Definition of Recursively Defined

A function or a procedure is said to be recursively defined if it is defined in terms of itself. In other words, a function $f$ is recursively defined if at least one value $f(x)$ is defined in terms of another value $f(y)$, where $x \neq y$. Similarly, a procedure $P$ is recursively defined if the actions of $P$ for some argument $x$ are defined in terms of the actions of $P$ for another argument $y$, where $x \neq y$.

Many useful recursively defined functions have domains that are inductively defined sets. Similarly, many recursively defined procedures process elements from inductively defined sets. For these cases there are very useful construction techniques. Let’s describe the two techniques.

**Constructing a Recursively Defined Function (3.6)**

If $S$ is an inductively defined set, then we can construct a function $f$ with domain $S$ as follows:

1. For each basis element $x \in S$, specify a value for $f(x)$.
2. Give rules that, for any inductively defined element $x \in S$, will define $f(x)$ in terms of previously defined values of $f$.

Any function constructed by (3.6) is recursively defined because it is defined in terms of itself by the induction part of the definition. In a similar way we can construct a recursively defined procedure to process the elements of an inductively defined set.

**Constructing a Recursively Defined Procedure (3.7)**

If $S$ is an inductively defined set, we can construct a procedure $P$ to process the elements of $S$ as follows:

1. For each basis element $x \in S$, specify a set of actions for $P(x)$.
2. Give rules that, for any inductively defined element $x \in S$, will define the actions of $P(x)$ in terms of previously defined actions of $P$.

In the following paragraphs we’ll see how (3.6) and (3.7) can be used to construct recursively defined functions and procedures over a variety of inductively defined sets. Most of our examples will be functions. But we’ll define a few procedures too.

3.2.1 Numbers

Let’s see how some number functions can be defined recursively. To illustrate the idea, suppose we want to calculate the sum of the first $n$ natural numbers for any $n \in N$. Letting $f(n)$ denote the desired sum, we can write the informal definition

$$f(n) = 0 + 1 + 2 + \cdots + n.$$  

We can observe, for example, that $f(0) = 0$, $f(1) = 1$, $f(2) = 3$, and so on. After a while we might notice that $f(3) = f(2) + 3 = 6$ and $f(4) = f(3) + 4 = 10.$
3.16 Using the Floor Function

Let \( f : \mathbb{N} \rightarrow \mathbb{N} \) be defined in terms of the floor function as follows:

\[
    f(0) = 0, \\
    f(n) = f(\text{floor}(n/2)) + n \quad \text{for} \quad n > 0.
\]

Notice in this case that \( f(n) \) is not defined in terms of \( f(n-1) \) but rather in terms of \( f(\text{floor}(n/2)) \). For example, \( f(16) = f(8) + 16 \). The first few values are \( f(0) = 0, f(1) = 1, f(2) = 3, f(3) = 4, \) and \( f(4) = 7 \). We'll calculate \( f(25) \).

\[
    f(25) = f(12) + 25 \\
    = f(6) + 12 + 25 \\
    = f(3) + 6 + 12 + 25 \\
    = f(1) + 3 + 6 + 12 + 25 \\
    = f(0) + 1 + 3 + 6 + 12 + 25 \\
    = 0 + 1 + 3 + 6 + 12 + 25 \\
    = 47.
\]

3.17 Adding Odd Numbers

Let \( f : \mathbb{N} \rightarrow \mathbb{N} \) denote the function to add up the first \( n \) odd natural numbers. So \( f \) has the following informal definition.

\[
    f(n) = 1 + 3 + \cdots + (2n - 1).
\]

For example, the definition tells us that \( f(0) = 1 \). For \( n > 0 \) we can make the following transformation of \( f(n) \) into an expression in terms of \( f(n-1) \):

\[
    f(n) = 1 + 3 + \cdots + (2n - 1) \\
    = (1 + 3 + \cdots + (2n - 1) - 1) + (2n + 1) \\
    = f(n - 1) + 2(n + 1) + 1.
\]

So we can make the following recursive definition of \( f \):

\[
    f(0) = 1, \\
    f(n) = f(n - 1) + 2n + 1 \quad \text{for} \quad n > 0.
\]

Alternatively, we can write the recursive part of the definition as

\[
    f(n + 1) = f(n) + 2n + 1.
\]

3.2 Recursive Functions and Procedures

We can also write the definition in the following if-then-else form:

\[
    f(n) = \begin{cases} 
        0 & \text{if} \quad n = 0 \\
        f(n-1) + 2n + 1 & \text{else}
    \end{cases} \]

Here is the evaluation of \( f(3) \) using the if-then-else definition:

\[
    f(3) = f(2) + 2(3) + 1 \\
    = f(1) + 2(2) + 1 + 2(3) + 1 \\
    = f(0) + 2(1) + 1 + 2(2) + 1 + 2(3) + 1 \\
    = 1 + 2(1) + 1 + 2(2) + 1 + 2(3) + 1 \\
    = 1 + 3 + 5 + 7 \\
    = 16.
\]

3.18 The Rabbit Problem

The Fibonacci numbers are the numbers in the sequence

\[
    0, 1, 1, 2, 3, 5, 8, 13, \ldots
\]

where each number after the first two is computed by adding the preceding two numbers. These numbers are named after the mathematician Leonardo Fibonacci, who in 1202 introduced them in his book Liber Abaci, in which he proposed and solved the following problem: Starting with a pair of rabbits, how many pairs of rabbits can be produced from that pair in a year if it is assumed that every month each pair produces a new pair that becomes productive after one month?

For example, if we don't count the original pair and assume that the original pair needs one month to mature and that no rabbits die, then the number of new pairs produced each month for 12 consecutive months is given by the sequence

\[
    0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89.
\]

The sum of these numbers, which is 232, is the number of pairs of rabbits produced in one year from the original pair.

Fibonacci numbers seem to occur naturally in many unrelated problems. Of course, they can also be defined recursively. For example, letting \( \text{fib}(n) \) be the \( n \)th Fibonacci number, we can define \( \text{fib} \) recursively as follows:

\[
    \text{fib}(0) = 0, \\
    \text{fib}(1) = 1, \\
    \text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2) \quad \text{for} \quad n \geq 2.
\]

The third line could be written in pattern matching form as

\[
    \text{fib}(n + 2) = \text{fib}(n + 1) + \text{fib}(n).
\]
The definition of \( \text{fib} \) in if-then-else form looks like
\[
\text{fib}(n) = \begin{cases} 
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
\text{fib}(n-1) + \text{fib}(n-2) & \text{else}
\end{cases}.
\]

Sum and Product Notation

Many definitions and properties that we use without thinking are recursively defined. For example, given a sequence of numbers \( (a_1, a_2, \ldots, a_n) \) we can represent the sum of the sequence with \textit{summation notation} using the symbol \( \Sigma \) as follows.
\[
\sum_{i=1}^{n} a_i = a_1 + a_2 + \cdots + a_n.
\]

This notation has the following recursive definition, which makes the practical assumption that an empty sum is 0.
\[
\sum_{i=1}^{n} a_i = \begin{cases} 
0 & \text{if } n = 0 \\
a_n + \sum_{i=1}^{n-1} a_i & \text{else}
\end{cases}.
\]

Similarly, we can represent the product of the sequence with the following \textit{product notation}, where the practical assumption is that an empty product is 1.
\[
\prod_{i=1}^{n} a_i = a_1 \cdot a_2 \cdot \cdots \cdot a_n.
\]

In the special case where \( (a_1, a_2, \ldots, a_n) = (1, 2, \ldots, n) \) the product defines the \textit{factorial function}, which is denoted by \( n! \) and is read "n factorial." In other words, we have
\[
n! = (1)(2) \cdots (n-1)n.
\]

For example, \( 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \), and \( 0! = 1 \). So we can define \( n! \) in the following recursive form.
\[
n! = \begin{cases} 
1 & \text{if } n = 0 \\
n \cdot (n-1)! & \text{else}
\end{cases}.
\]

3.2.2 Strings

Let's see how some string functions can be defined recursively. To illustrate the idea, suppose we want to calculate the complement of any string over the alphabet \{a, b\}. For example, the complement of the string \( bbab \) is \( aaba \).
We’ll demonstrate the definition of \( f \) by calculating \( f(aabbab, aababb) \):
\[
\begin{align*}
f(aabbab, aababb) &= af(abbab, aab) \\
&= aaf(bbab, abb) \\
&= aabf(bab, abb) \\
&= aabA \\
&= aab.
\end{align*}
\]

**Example 3.20 Converting Natural Numbers to Binary**

Recall from Section 2.1 that we can represent a natural number \( x \) as
\[
x = 2(\text{floor}(x/2)) + x \mod 2.
\]

This formula can be used to create a binary representation of \( x \) because \( x \mod 2 \) is the rightmost bit of the representation. The next bit is found by computing \( \text{floor}(x/2) \mod 2 \). The next bit is \( \text{floor}(\text{floor}(x/2)/2) \mod 2 \), and so on. For example, we’ll compute the binary representation of 13.

\[
\begin{align*}
13 &= 2 \left\lfloor 13/2 \right\rfloor + 13 \mod 2 &= 2(6) + 1 \\
6 &= 2 \left\lfloor 6/2 \right\rfloor + 6 \mod 2 &= 2(3) + 0 \\
3 &= 2 \left\lfloor 3/2 \right\rfloor + 3 \mod 2 &= 2(1) + 1 \\
1 &= 2 \left\lfloor 1/2 \right\rfloor + 1 \mod 2 &= 2(0) + 1
\end{align*}
\]

We can read off the remainders in reverse order to obtain 1101, which is the binary representation of 13.

Let’s try to use this idea to write a recursive definition for the function “binary” to compute the binary representation for a natural number. If \( x = 0 \) or \( x = 1 \), then \( x \) is its own binary representation. If \( x > 1 \), then the binary representation of \( x \) is that of \( \text{floor}(x/2) \) with the bit \( x \mod 2 \) attached on the right end. So our recursive definition of binary can be written as follows, where “cat” is the string concatenation function.

\[
\begin{align*}
\text{binary}(0) &= 0, \\
\text{binary}(1) &= 1, \\
\text{binary}(x) &= \text{cat} (\text{binary}(\lfloor x/2 \rfloor), x \mod 2) \quad \text{for } x > 1.
\end{align*}
\]

The definition can be written in if-then-else form as
\[
\text{binary}(x) = \begin{cases} 
0, & \text{if } x = 0 \\
1, & \text{if } x = 1 \\
\text{cat} (\text{binary}(\lfloor x/2 \rfloor), x \mod 2), & \text{for } x > 1.
\end{cases}
\]

For example, we’ll unfold the definition to calculate binary(13):
\[
\begin{align*}
\text{binary}(13) &= \text{cat} (\text{binary}(6), 1) \\
&= \text{cat} (\text{cat} (\text{binary}(3), 0), 1) \\
&= \text{cat} (\text{cat} (\text{cat} (\text{binary}(1), 1), 0), 1) \\
&= \text{cat} (\text{cat} (\text{cat}(1, 1), 0), 1) \\
&= \text{cat} (110, 1) \\
&= 1101.
\end{align*}
\]

**3.2.3 Lists**

Let’s see how some functions that use lists can be defined recursively. To illustrate the idea, suppose we need to define the function \( f : \mathbb{N} \rightarrow \text{lists}(\mathbb{N}) \) that computes the following backwards sequence:
\[
f(n) = (n, n-1, \ldots, 1, 0).
\]

With a little help from the cons function for lists, we can transform the informal definition of \( f(n) \) into a computable expression in terms of \( f(n-1) \):
\[
f(n) = \text{cons}(n, f(n-1))
\]

Therefore, \( f \) can be defined recursively by
\[
\begin{align*}
f(0) &= (0), \\
f(n) &= \text{cons}(n, f(n-1)) \quad \text{for } n > 0.
\end{align*}
\]

This definition can be written in if-then-else form as
\[
f(n) = \begin{cases} 
(0), & \text{if } n = 0 \\
\text{cons}(n, f(n-1)), & \text{if } n > 0.
\end{cases}
\]

To see how the evaluation works, look at the unfolding that results when we evaluate \( f(3) \):
\[
\begin{align*}
f(3) &= \text{cons}(3, f(2)) \\
&= \text{cons}(3, \text{cons}(2, f(1))) \\
&= \text{cons}(3, \text{cons}(2, \text{cons}(1, f(0)))) \\
&= \text{cons}(3, \text{cons}(2, \text{cons}(1, \text{cons}(0, f(-1))))) \\
&= \text{cons}(3, \text{cons}(2, \text{cons}(1, \text{cons}(0, f(-2))))) \\
&= \text{cons}(3, \text{cons}(2, \text{cons}(1, \text{cons}(0, f(-3))) \\
&= \text{cons}(3, \text{cons}(2, \text{cons}(1, \text{cons}(0, (3, 2, 1, 0)))) \\
&= (3, 2, 1, 0).
\end{align*}
\]
3.2.2 The Distribute Function

Suppose we want to write a recursive definition for the distributive function, which we'll denote by "\text{dist}". Recall, for example, that

\text{dist}(a, (b, c, d, e)) = ((a, b), (a, c), (a, d), (a, e)).

To discover the recursive part of the definition, we'll rewrite the example equation by splitting up the lists into head and tail components as follows:

\text{dist}(a, (b, c, d, e))
= ((a, b), (a, c), (a, d), (a, e))
= (a, b) :: (a, c) :: (a, d) :: (a, e).

That's the key to the recursive part of the definition. Since we are working with lists, the basis case is \text{dist}(a, ( ))$, which we define as ( ). So the recursive definition can be written as follows:

\text{dist}(x, ( )) = ( ),
\text{dist}(x, h :: t) = (x, h) :: \text{dist}(x, t).

For example, we'll evaluate the expression \text{dist}(3, (10, 20)):

\text{dist}(3, (10, 20)) = (3, 10) :: \text{dist}(3, (20))
= (3, 10) :: (3, 20) :: \text{dist}(3, ( ))
= (3, 10) :: (3, 20) :: ( )
= (3, 10) :: ((3, 20))
= (3, 10) :: (3, 20).

An if-then-else definition of \text{dist} takes the following form:

\text{dist}(x, L) = \begin{cases} 
( ) & \text{if } L = ( ) \text{ then } ( ) \\
(x, \text{head}(L)) & \text{else } (x, \text{tail}(L)). 
\end{cases}

3.2.3 The Pairs Function

Recall that the "pairs" function creates a list of pairs of corresponding elements from two lists. For example,

\text{pairs}((a, b, c), (1, 2, 3)) = ((a, 1), (b, 2), (c, 3)).

To discover the recursive part of the definition, we'll rewrite the example equation by splitting up the lists into head and tail components as follows:

\text{pairs}((a, b, c), (1, 2, 3))
= ((a, 1), (b, 2), (c, 3))
= (a, 1) :: ((b, 2), (c, 3))
= (a, 1) :: (b, 2) :: (c, 3).

\text{pairs}((a, b, c), (1, 2, 3))
= (a, 1) :: \text{pairs}((b, c), (2, 3)).
This allows us to write the following definition for the remove function:
remove \( (n, s) \) = if head \( (s) \mod n = 0 \) then remove \( (n, \text{tail} \ (s)) \)
else head \( (s) \) :: remove \( (n, \text{tail} \ (s)) \).

Then our desired sequence of primes is represented by the expression
\[ 
\text{Primes} = \text{sieve} \left( \text{ints}(2) \right) .
\]

In the exercises we'll evaluate some functions dealing with primes.

---

**Exercises**

#### Evaluating Recursively Defined Functions

1. Given the following definition for the \( n \)th Fibonacci number:
\[
\begin{align*}
\text{fib} \ (0) & = 0, \\
\text{fib} \ (1) & = 1, \\
\text{fib} \ (n) & = \text{fib} \ (n - 1) + \text{fib} \ (n - 2) & & \text{if } n > 1.
\end{align*}
\]

Write down each step in the evaluation of \( \text{fib}(4) \).

2. Given the following definition for the length of a list:
\[ 
\text{length} \ (L) = \begin{cases} 
0 & \text{if } L = \emptyset \\
1 + \text{length} \ (\text{tail} \ (L)) & \text{otherwise}
\end{cases}.
\]

Write down each step in the evaluation of \( \text{length} \ ((r, s, t, u)) \).

3. For each of the two definitions of “makeTree” given by (3.9) and (3.10), write down all steps to evaluate \( \text{makeTree} \ ((3, (2, 4))) \).

#### Numbers

4. Construct a recursive definition for each of the following functions, where all variables are natural numbers.
\[
\begin{align*}
a. \ f(n) & = 0 + 2 + 4 + \ldots + 2n. \\
b. \ f(n) & = \text{floor}(0/2) + \text{floor}(1/2) + \ldots + \text{floor}(n/2). \\
c. \ f(n) & = \gcd(1, n) + \gcd(2, n) + \ldots + \gcd(n, n) & & \text{for } n > 0. \\
d. \ f(n) & = (0 \mod 2) + (1 \mod 3) + \ldots + (n \mod (n + 2)). \\
e. \ f(n, k) & = 0 + k + 2k + \ldots + nk. \\
f. \ f(n, k) & = k + (k + 1) + (k + 2) + \ldots + (k + n).
\end{align*}
\]

#### Strings

5. Construct a recursive definition for each of the following string functions for strings over the alphabet \( \{a, b\} \).
\[
\begin{align*}
a. \ f(x) & \text{ returns the reverse of } x. \\
b. \ f(x) & = xy, \text{ where } y \text{ is the reverse of } x. \\
c. \ f(x, y) & \text{ tests whether } x \text{ is a prefix of } y. \\
d. \ f(x, y) & \text{ tests whether } x = y. \\
e. \ f(x) & \text{ tests whether } x \text{ is a palindrome.}
\end{align*}
\]

#### Lists

6. Construct a recursive definition for each of the following functions that involve lists. Use the infix form of cons in the recursive part of each definition. In other words, write \( h : t \) in place of \( \text{cons}(h, t) \).
\[
\begin{align*}
a. \ f(n) & = (2n, 2(n - 1), \ldots, 2, 0). \\
b. \ f(n) & = \text{max}(L) \text{ is the maximum value in nonempty list } L \text{ of numbers.} \\
c. \ f(x, (x_0, \ldots, x_n)) & = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n. \\
d. \ f(x, y) & = \text{cons}(x, y) \text{ is the list of elements } x \text{ in list } L \text{ that have property } P. \\
e. \ f(x, (x_1, \ldots, x_n)) & = (x_1 + a, \ldots, x_n + a). \\
f. \ f(x, (x_1, y_1), \ldots, (x_n, y_n)) & = ((x_1 + a, y_1), \ldots, (x_n + a, y_n)). \\
g. \ f(n) & = (\langle 0, n \rangle, \langle 1, n - 1 \rangle, \ldots, \langle n - 1, 1 \rangle, \langle n, 0 \rangle). \text { Hint: Use part (f).} \\
h. \ f(x, (x_1, x_2), \ldots, x_n) & = (\langle x_1, g(x_1) \rangle, \langle x_2, g(x_2) \rangle, \ldots, \langle x_n, g(x_n) \rangle). \\
i. \ f(y, (h, (x_1, \ldots, x_n))) & = ((g(x_1), h(x_1)), \ldots, (g(x_n), h(x_n))).
\end{align*}
\]

#### Using Cat or ConsR

7. Construct a recursive definition for each of the following functions that involve lists. Use the cat operation or \( \text{consR} \) operation in the recursive part of each definition. (Notice that for any list \( L \) and element \( x \) we have \( \text{cat}(L, x) = \text{consR}(L, x) \).)
\[
\begin{align*}
a. \ f(n) & = \langle 0, 1, \ldots, n \rangle. \\
b. \ f(n) & = \langle 0, 2, 4, \ldots, 2n \rangle. \\
c. \ f(n) & = \langle 1, 3, 5, \ldots, 2n + 1 \rangle. \\
d. \ f(n, k) & = \langle n, n + 1, n + 2, \ldots, n + k \rangle. \\
e. \ f(n, k) & = \langle 0, k, 2k, 3k, \ldots, nk \rangle. \\
f. \ f(y, n) & = \langle (0, g(0)), (1, g(1)), \ldots, (n, g(n)) \rangle. \\
g. \ f(n, m) & = \langle n, n + 1, n + 2, \ldots, m - 1, m \rangle, \text{ where } n \leq m.
\end{align*}
\]

8. Let \( \text{insert} \) be a function that extends any binary function \( f \) so that \( \text{insert}(a, f) \) evaluates a list of two or more arguments. For example,
\[
\text{insert}(+, [1, 4, 2, 9]) = 1 + (4 + (2 + 9)) = 16.
\]
1.1 Problems

And there's a dreadful law here—it was made by mistake, but there it is—that if anyone asks for machinery they have to have it and keep on using it.

Edith Nesbit, The Magic City

We want to identify computationally solvable problems, and for each such problem we want bounds on the computation cost. First we only consider problems that are well-defined enough to be solved on computers. Then we restrict ourselves further to problems that we can divide into sizes. For each size there may be many instances of the problem. For example, every time we wash dishes we're solving an instance of the dishwashing problem; each instance is different, if only in time, but many of them may have the same size (say, the number of dishes).

To solve an instance of the problem we first code the instance to form an input to an algorithm. We then feed this input to the algorithm and the algorithm produces some output. Then we decode this output into an answer for the problem instance. To solve a problem we must show that we can go through this process for any instance of the problem.

One nice thing about digital computers is that once we find a solution for all instances of a problem of one size, we can use it for instances of any size. Further, it is usually not interesting to solve a problem for small sizes; those are often solved faster by hand. So we're interested in well-defined problems whose instances we can group by size, and we want efficient solutions when those sizes are large.

Grouping problem instances by size is like grouping books by their page length (or word count, or any other quantitative measure). For example, using length as our size measure, the set of all fifty page books is the set of instances of the book-reading problem of size fifty. Note how arbitrary this notion of size is; in the introduction we considered using page count itself as a measure of the reading difficulty of a book. In effect, we pick one "natural" measure, then see how another measure relates to it.

Selecting a natural measure is not always easy to do. For example, suppose we want to arrange some books by height. One natural size for this problem is the number of books, since the more books there are to arrange the more work there is to do. But this assumes that one book is much like another. If one of the books is as big as the Encyclopedia Britannica then we should take weight into account as well.

Okay, pretend we have a reasonable measure of input size. Now we have to estimate the difficulty of each instance of the problem in terms of its size. We want a function of the problem's size that reflects the effort necessary to solve the problem. So far we have assumed that the first thing to look at is the program's speed. But, our machine may have little memory, so we could look for programs that use the least memory. Similarly, it could be important to reduce the number of disk accesses, the number of comparisons, the number of assignments, or the number of multiplications. In general, there are an infinite number of combinations of program attributes we could try to improve.

We could also choose among programs using more intangible properties. For example: how hard they are to write, how hard they are to modify, and how hard they are to understand. However, lacking a formal definition of these important properties let's leave these measures of program cost to software engineering. This is like trying to measure the difficulty of writing a book, not reading it.

Thus, our aim is to solve problems with as little computational effort per problem instance as possible. Sometimes to reduce computational effort we expend great conceptual effort: we only do the conceptual work once but the resulting program does the computational work every time we run it.

One more thing: to do this analysis we need mathematics. Some people confuse mathematics with mere symbol manipulation, but mathematics is much more about critical thinking than it is about symbols. Thinking mathematically forces us to identify our assumptions and so deal with the unusual. As a corollary, we shouldn't be surprised if we derive counter-intuitive results using mathematics: in a way, that's what it's for. It's especially dangerous to rely only on intuition when designing algorithms since computers typically deal with huge problem sizes, and we don't. Intuition is a product of everyday experience and most of us don't think about things with many parts every day. For example, we have a hard time appreciating the effects predicted by quantum theory—we call quantum effects "counter-intuitive." A ball in a bucket does not spontaneously jump through the wall of the bucket, yet this is precisely what electrons do in potential wells, their analogues of buckets.

Similarly, when designing an algorithm we tend to think of the algorithm working on about ten items when in fact we're going to use it on a million items. As we have seen, any display program producing graphics on a bitmapped screen must look at about a million pixels just to process one screenful. And this is not even counting the work the program must do per pixel. Anything being repeated a million times per screenful must be

1Besides being a great poet Yeats was obviously a great programmer: "A line will take us hours maybe. Yet if it does not seem a moment's thought. / Our stitching and unstitching has been naught." W. B. Yeats, Adam's Curse.
as fast as possible. Further, even if any one problem instance is small, when there are a million instances even small improvements per instance magnify into large savings.

In sum, the Holy Grail of analysis is to put our computational resources to the best possible use. Given the choice between a ten-second solution and a ten-hour solution we would be crazy to choose the ten-hour solution. Of course, the ten-hour solution is perfectly acceptable if we only have to solve the problem once, or if the ten-second solution is hard to program, or is otherwise expensive. But this is just a more general version of the same goal—we are still trying to reduce effort, the only difference is that now “effort” includes more than just the computer resources used in a solution.

Problem Types

We can classify computational problems by problem requirements and problem difficulty.

In terms of problem requirements there are six computational problems:

- **Search** problems: Find an $X$ in the input satisfying property $Y$.
- **Structuring** problems: Transform the input to satisfy property $Y$.
- **Construction** problems: Build an $X$ satisfying property $Y$.
- **Optimization** problems: Find the best $X$ satisfying property $Y$.
- **Decision** problems: Decide whether the input satisfies property $Y$.
- **Adaptive** problems: Maintain property $Y$ over time.

Chapters two and three examine search problems; chapters four and five examine structuring problems; chapters five and six examine construction problems; and chapters six and seven examine optimization and decision problems. We will rarely examine adaptive problems even though they are of great practical importance and they include many real systems (such as operating systems, adaptive control systems, and server systems). Adaptive problems involve practical issues beyond the scope of this book.

Instead of tackling a realistic version of a problem we will look at a simplified version—a toy problem. Toy problems do not include memory management issues and other important details. Although unrealistic, toy problems are useful because we can solve them without the clutter attendant on more realistic versions of the same problem. Thus they better expose the problem’s inherent difficulty. We can then use lessons learned while solving toy problems to solve more realistic problems—in the same way that law students use old cases.

We can also classify problems by difficulty. There are four categories of hard problems:

- A **conceptually hard** problem: We don’t have an algorithm to solve this problem because we don’t understand the problem well enough.
- An **analytically hard** problem: We have an algorithm to solve this problem, but we don’t know how to analyze how long it will take to solve every problem instance.
- A **computationally hard** problem: We have an algorithm and we have analyzed it, but analysis suggests that relatively small problem instances will take millions of years to solve. This category splits into two groups: problems we **know** are computationally hard, and problems we **suspect** are computationally hard.
- A **computationally unsolvable** problem: We don’t have an algorithm to solve this problem because no such algorithm can exist.

We can use these four categories to differentiate among three subfields of computer science: artificial intelligence explores problems in the first and second categories; complexity theory, of which analysis is a part, explores problems in the second and third categories; and computability theory explores problems in the third and fourth categories.

In this book, our central metaproblem revolves around the third category. What does it mean to say that a problem is computationally hard? We will return to this question at the end of this chapter (page 52) and in chapter seven.

Let’s use the following toy problem as a running example of the analysis process. After discussing each step in solving an arbitrary problem we’ll turn to our example problem to see how the step works in practice. As the analysis proceeds we will see the problem first as conceptually hard, then analytically hard, and finally computationally hard.

**The towers of Hanoi problem:** Given three pegs and $n$ disks of different sizes placed in order of size on one peg (see figure 1.3), transfer the disks from the original peg to another peg with the constraints that:

- each disk is on a peg,
- no disk is ever on a smaller disk, and
- only one disk at a time is moved.
1.2 Models

A good model represents a well-balanced abstraction of a real practical situation—not too far from and not too close to the real thing.

Arto Salomaa, Computation and Automata

Given a problem, we first select a model so that we can talk about the problem sensibly. To tell whether something is a solution to the problem we pick a set of legal operations that solutions can use. And to distinguish between good and bad solutions we pick an operation, or set of operations, in that set to minimize. The set of legal operations is the environment and the subset of operations we want to minimize is the goal. The environment and the goal together make up the model. Choosing an environment includes choosing the kind of machine that solutions will run on, the type of language that solutions will be written in, and the character of the physical environment that the machine runs in. For example, we usually assume, contrary to reality, that cosmic rays will not disrupt the machine.

Choosing a set of operations and restricting attention to only a few is analogous to scientific modelling, in which we abstract some essential feature and examine it alone. A physicist examining falling bodies first assumes that air resistance is negligible and that the body’s mass is minuscule compared to the mass of the earth. As she better understands these simple situations she allows drag due to air (and the like) back into the equations of motion until she arrives at a good approximation to the fall of real bodies. As she removes more and more simplifying assumptions, the model behaves more and more like the real system.

The analysis goal is to minimize all resources used. This, like the Holy Grail, is difficult to attain; so to begin we will count the number of times only one easily identifiable operation is performed (for example, an addition, a comparison, a disk access). We assume that the chosen operation is proportional to the total computational resources that the problem requires. Our goal then is to attain grace by minimizing the number of times we perform a chosen operation while restricting ourselves to solutions within a well-defined, and small, model. When choosing a model, we will usually assume that the crucial operation is proportional to the run time of the algorithm we choose to solve the problem. Usually the chosen operation is expensive, or frequent, or it otherwise reflects the overall amount of work done.

Here’s how this works for the dishwashing problem. Suppose we choose to measure problem size by the number of dishes to be washed. Suppose we want to predict how much time a new problem instance will take (as opposed, say, to how much water it will take). First, we select the set of operations that we can use to solve the problem; for instance, picking up a dish, immersing it, applying soap, and so on. Different choices of legal operations determine different models. For example, one model may assume that we have a dishwasher!

After choosing the legal operations we choose one as a barometer for all the others. For instance, it’s likely that drying a dish is no harder than washing a dish. And, since we must dry every dish we wash, we may choose to count only the number of times we wash a dish and ignore drying. Note that we have made the further assumption that overall washing time (the property we’re trying to reduce) is proportional to the time taken to wash each dish. Finally, note that we only know that it is proportional, we don’t know the actual time it will take. That’s good enough because we can compare two or more solutions in the same model based only on proportionality information. Further, we can always find the actual time by running experiments to determine the constant of proportionality.

The two assumptions given for the problem of squaring $n$ (page 3), together with some reasonable assumptions about a suitable machine, constitute a model for that problem. In this model, the goal is to minimize the number of assignments since, by assumption, assignments are the only
operations that matter. Further, in this model, the first fragment is the best program for the problem. There is no guarantee, however, that this model captures the real difficulty of the problem. After deciding on a model we have to go back to reality to check its predictive power. Only after checking predictions against reality can we be sure that the model captures the essence of the problem’s difficulty.

For instance, suppose we run each of the three fragments with several inputs and discover that their run times are about the same, no matter how large the input is. That tells us that assignments don’t matter that much (or that the compiler is changing our code). Alternatively, we may find that the run times roughly follow predicted behavior. Now suppose the problem is so frequent, expensive, or important that we need a really good estimate of future run times. Then, and only then, should we add more sophisticated measures to our model in an effort to get more accurate time predictions (for example, we may decide that additions should also be counted).

Our default environment will be that we are running our algorithms on errorless, sequential, digital computers, and that we will translate our algorithms into programs in an imperative language (like Pascal). Our environment for the towers of Hanoi problem will be the default environment plus the three constraints defining legal moves (page 10). Our goal will be to reduce the total number of disk moves.

1.3 Algorithms

It has often been said that a person does not really understand something until he teaches it to someone else. Actually a person does not really understand something until he can teach it to a computer.

Donald E. Knuth
“Computer Science and Its Relation to Mathematics.”
American Mathematical Monthly, 81, 1974

Having chosen a model for the problem, next we devise an algorithm to solve the problem. This algorithm must use only the operations allowed within the model. Up to this point we haven’t seen a formal definition of an algorithm, nor (surprise!) will we see one now. For now, an algorithm is a finite sequence of operations, each chosen from a finite set of well-defined operations, that halts in a finite time.

That looks like a definition, but it really isn’t. It places some restrictions on what an algorithm can possibly be (for example, it cannot take indefinitely long to describe) but it doesn’t say what an algorithm is. A really useful definition would allow us to “mechanically” recognize an algorithm whenever we saw one, in the same way that we can recognize alarm clocks. This is hard because algorithms come bundled with the idea of human purpose; we intend an algorithm to accomplish some goal. Unfortunately, and unlike alarm clocks, no two algorithms necessarily have the same goal (beyond the complex one of “solving a problem”). To get to a formal definition will take us most of this book, and until then, when you see algorithm think recipe, prescription, procedure, method, strategy, technique, or computation.

All right, now let’s design an algorithm solving the towers of Hanoi problem. The first thing to do when confronted with a problem is to solve the problem for its smallest instances. Perhaps there is some insight there that we can generalize to larger instances. For the towers of Hanoi problem the most natural size is the number of disks, $n$; also, for each size there is only one problem instance. When $n = 1$ or $n = 2$ the problem is easy; however $n = 3$ requires a little thought to minimize the number of moves. Label the three pegs in figure 1.3 [p. 11] $A$, $B$, and $C$, and suppose we have to move the disks from $A$ to $C$. If we move the smallest disk to $B$ then we will be in trouble when we move the second smallest to $C$, since $C$ is the eventual destination of the largest disk. Thus, we must move the second smallest, which means that we must move the smallest. This seems to imply that we should first put the smallest on $C$, then the second smallest on $B$, and finally the smallest on top of the second smallest thereby leaving $C$ free for the biggest disk. Now what does this imply when we have $n$ disks? Well, when we’re ready to move the biggest disk (which is still sitting patiently on $A$), there can’t be any disks on top of it since we can only move one disk at a time. So, all the smaller disks must be on $B$. Also, the $n - 1$ disks on $B$ must be stacked in order of size, otherwise some disk must be on top of a smaller one.

Think about this before reading on.

Okay, at some point the biggest disk is alone on $A$ and all $n - 1$ smaller disks are piled neatly on $B$. There is nothing on $C$ in preparation for the big move. Now observe that all the time the biggest disk was patiently sitting on $A$, the other disks were oblivious to it. Why? Well, the rules say that no disk may be put on a smaller disk, but since the biggest disk is bigger than all others, it is always legal to place any other disk on $A$ if there is nothing on $A$ besides the biggest disk!
If we could only solve the same towers of Hanoi problem but with \( n - 1 \) disks and with a different destination peg (\( B \) instead of \( C \)) then we could solve our version of the problem! But how, you quibble, can we solve a problem in terms of itself? Isn’t this circular reasoning? Well, no. In the reduction of the problem with \( n \) disks to one with \( n - 1 \) disks we know that the process will eventually stop since the number of disks is decreasing and we know how to solve one and two disk problems.

Let’s go back to the \( n = 3 \) case. There we realized that we needed to make a tower of the two smallest disks on \( B \) before we could move the biggest disk from \( A \) to \( C \). So, if only we could solve the \( n = 2 \) case (but for a different destination peg) then we could solve the \( n = 3 \) case (try it). But we can easily solve the \( n = 2 \) case. Thus, when \( n = 3 \), we first move the top two disks to \( B \) (three moves), move the biggest disk to \( C \) (one move), then move the two disks on \( B \) to \( C \) (three moves). So three disks take no more than seven moves. See figure 1.4.

In general, we have \( n \) disks and we want to move them from \( A \) to \( C \). Our algorithm is to first move the top \( n - 1 \) disks from \( A \) to \( B \), move the biggest disk from \( A \) to \( C \), then move the \( n - 1 \) disks on \( B \) to \( C \). An algorithm that uses itself to solve a problem is called a recursive algorithm. See algorithm 1.1; the pegs have been given more meaningful names in the algorithm.

**Algorithm 1.1**

\[
\text{HANOI}\left(\text{Start, Temp, End, } n\right)
\]

\begin{align*}
\{ & \text{ Solve the towers of Hanoi problem for } n \geq 1 \text{ disks. } \\
\text{ if } n &= 1 \\
\text{ then } & \text{ move Start's top disk to End} \\
\text{ else } & \text{ HANOI(Start, End, Temp, } n - 1) \\
& \text{ move Start's top disk to End} \\
& \text{ HANOI(Temp, Start, End, } n - 1) \\
\end{align*}

How did we find this algorithm? The first ideas came from feeling out the problem for small \( n \). Then we looked at large \( n \) and tried to generalize the insights generated from the first phase. Finally we went back to small \( n \) to see if the generalized insights made sense.

**The forward-backward strategy:** Solve simple special cases and generalize their solution, then test the generalization on other special cases.

Depending on how insightful we are in the first phase, and how well we generalize in the second phase, this procedure could repeat many times. *Think small, then think big.* This strategy is very handy, but it won’t do well on problems whose best solution for large \( n \) is not like their best solution for small \( n \).

### 1.4 Analysis

When we mean to build,

We first survey the plot, then draw the model;

And when we see the figure of the house,

Then must we rate the cost of the erection.

William Shakespeare, *Henry IV*, part 2, line 5

Having decided on an algorithm to solve the problem the next step is to analyze both the algorithm and the problem within the model. We want
categories. Perhaps one type of driver is an aggressive one who always tries to bump into the nearest car. With the Driver interface, it is easy to add new kinds of drivers to the simulation. We only need to write a class that implements the Driver interface.

If there are different kinds of drivers, there must be some way of choosing how many of each type will be in the simulation. This could be decided automatically by the program using statistics collected by the bumper car company. For example, perhaps 25 percent of drivers drive aggressively. Or the program might be modified to let the user pick the types of drivers.

Another extension might be to add a mode where the simulation runs in real time, not pausing between steps. The user could stop the simulation at any point. This would be useful with a graphical display showing the cars moving around the arena, instead of a scrolling text display.

exercises
1. Add a toString method to the Direction class that prints a string version of the direction. Write a driver program that tests your new method.
2. List more sample inputs for each of the types of test cases given.
3. What other extensions could we add to the simulation in the future?

8.5 recursion in graphics
Recursion has several uses in images and graphics. The following section explores some image and graphics-based recursion examples.

tiled pictures
Carefully look at the display for the TiledPictures applet shown in Listing 8.14. There are actually three images. The entire area is divided into four equal sections. A picture of the world (with a circle indicating the Himalayan mountain region) is shown in the bottom-right section. The bottom-left section has a picture of Mt. Everest. In the top-right section is a picture of a mountain goat.

The interesting part of the picture is the top-left section. It contains a copy of the entire collage, including itself. In this smaller version you can see the three simple pictures in their three sections. And again, in the top-left corner, the picture is repeated (including itself). This repetition continues for several
Listing 8.14 continued

```
// Performs the initial call to the drawPictures method.
public void paint (Graphics page)
{
  drawPictures (APPLET_WIDTH, page);
}
```

levels. It is like what you see when you look in a mirror in the reflection of another mirror.

This effect is created quite easily using recursion. The applet's init method first loads the three images. The paint method then invokes the drawPictures method, which accepts a parameter that defines the size of the picture area. It draws the three images using the drawImage method, making the picture the correct size. The drawPictures method is then called recursively to draw the upper-left quadrant.

Each time, if the drawing area is large enough, the drawPictures method is invoked again, using a smaller drawing area. Eventually, the drawing area becomes so small that the recursive call is not performed. Note that drawPictures assumes the beginning (0, 0) coordinate as the relative location of the new images, no matter what their size is.

The base case of the recursion in this problem is a minimum size for the drawing area. Because the size is decreased each time, the base case eventually is reached and the recursion stops. This is why the upper-left corner is empty in the smallest version of the collage.

fractals

A fractal is a geometric shape made up of the same pattern repeated at different sizes and positions. Recursion is good at creating fractals. Interest in fractals has grown immensely in recent years, largely due to Benoît Mandelbrot, a Polish mathematician born in 1924. He demonstrated that there are fractals in many places in mathematics and nature. Computers have made fractals much easier to create and study. Over the past quarter century, the bright, interesting images that can be created with fractals have become as much an art form as a mathematical interest.

One example of a fractal is called the Koch snowflake, named after Helge von Koch, a Swedish mathematician. It begins with an equilateral triangle, which we call the Koch fractal of order 1. Koch fractals of higher orders are made by repeatedly changing all of the line segments in the shape.

To create the next higher order Koch fractal, the middle third of each line segment is replaced by two line segments, each having the same length as the replaced part. The line segments always come to an outward point. Figure 8.9 shows several orders of Koch fractals. As the order increases, the shape begins to look like a snowflake.
The applet shown in Listing 8.15 draws a Koch snowflake. The buttons at the top of the applet let the user increase and decrease the order of the fractal. Each time a button is pressed, the fractal image is redrawn. The applet serves as the listener for the buttons.

The fractal image is drawn on a canvas defined by the KochPanel class shown in Listing 8.16. The paint method makes the first calls to the recursive method drawFractal. The three calls to drawFractal in the paint method draw the original three sides of the equilateral triangle that make up a Koch fractal of order 1.

The variable current is the order of the fractal to be drawn. Each recursive call to drawFractal decreases the order by 1. The base case is the fractal with the order 1, which is a simple line.

If the order of the fractal is more than 1, three additional points are computed. Together with the parameters, these points form the four line segments of the new fractal. Figure 8.10 shows the transformation.

Based on the position of the two end points of the original line, a point one-third of the way and a point two-thirds of the way between them are computed. The calculations to determine the three new points actually have nothing to do with the recursive technique used to draw the fractal, so we won't discuss the details of these computations here.

An interesting mathematical feature of a Koch snowflake is that it has an infinite perimeter but a finite area. That is, the outline of the snowflake gets longer and longer because it jogs in and out, but the snowflake stays the same size. The outline can be an infinite length, but a rectangle drawn around the snowflake will never get bigger.
Listing 8.16 continued

```java
// Sets the fractal order to the value specified.
public void setOrder (int order)
{   current = order;
}

// Returns the current order.
public int getOrder ()
{   return current;
}
```

Figure 8.10 The transformation of each line segment of a Koch snowflake
Recursion Exercises #1

1.
Find f(5), given the following:
f(x) = {
    f(x - 2) + 3 if x > 0
    f(x + 1) + 2 if x < 0
    1 if x = 0
}

2. What is returned by the method call int x = method(15);?

    static public int method(int n)
    {
        int value = 0;
        if (n==0)
            value = 0;
        else if (n % 2 > 0)
            value = 2 + method(n-1);
        else
            value = 1 + method(n/2);
        return value;
    }

    a. 0
    b. 1
    c. 4
    d. 7
    e. 11

3. For the same method as the previous questions, what is the result of the method call int x = method(10);?

    a. 0
    b. 1
    c. 4
    d. 7
    e. 11
AP Computer Science

Recursion Exercises#2

Place the following exercises in a file called “RecursionPractice.java” (this file is partially pre-made on the exercises folder).

1. Define a “power” function recursively. Identify the base case and how recursive calls can be used to compute the power.
   
   \[ f(n) = \]

   Use recursion to write a function that computes a floating point number raised to any integer power. The method heading is:

   ```java
   static double power(double number, int power)
   ```

2. Define a “summation” function recursively. The summation should compute a total of all integers up to and including a given number. Thus given 5, the function computes 5 + 4 + 3 + 2 + 1, or 15.

   \[ f(n) = \]

   Write a recursive method that takes one positive integer as its parameter. The method will produce the summation of all numbers up to including the parameter. The method heading is:

   ```java
   static int summation(int number)
   ```
Next, modify the previous method so that it displays the current value of number in a decreasing fashion. For example, if the input were 5, the output would be: 54321. Next, modify it again so that the number is displayed at the beginning and end of the method, producing an effect as follows: 5432112345.

3. Define a “factorial” function recursively. This is similar to the previous method, except numbers are multiplied. The factorial of zero is defined to be one.

\[ f(n) = \]

Write a recursive method that takes one positive integer as its parameter. The method will produce the factorial in that series. The method heading is:

\[ \text{static int factorial(int number)} \]
Here’s a start of a sample driver program to test the above methods:

```java
import cs1.*;

class RecursionPractice {
    public static void main(String[] args) {
        int command;
        do {
            System.out.print("Enter a command (1=Power, 2=Summation, 3 = Factorial, 4=Quit): ");
            command = Keyboard.readInt();
            switch (command) {
                case 1: beginPower(); break;
                case 2: beginSummation(); break;
                case 3: beginFactorial(); break;
            }
        } while (command != 4);
    }

    public static void beginPower() {
        double base;
        int exponent;
        System.out.print("Enter the base: ");
        base = Keyboard.readDouble();
        System.out.print("Enter the exponent: ");
        exponent = Keyboard.readInt();
        System.out.println("The result is: “ + power(base, exponent));
    }

    public static double power(double base, int exp) {
        return;  // Implement the recursive function here
    }
}
```
AP Computer Science

Graphical Recursion Exercises

Use recursion to solve each of these problems. Use the Java software available with the SSAWindow class to write the procedures and run them. Remember that our Java package has the predefined procedures drawOval (xLeft, yTop, width, height), drawRect (xLeft, yTop, Width, Height), and drawLine (xStart, yStart, xEnd, yEnd). Practice running through the routines to observe how recursion works.

1. Write a method called CirclesInCircles. First ask the user for the x and y coordinates for the center of a circle and the radius. The computer should then draw co-centric circles starting with the original circle and diminishing the radius by 5 or 10 units each time until the radius is less than or equal to zero.

2. Modify the previous problem so that each circle will be bounded by a square. The circles will be drawn on the “way in” of the recursion; make it so that the squares will be drawn on the “way out” (i.e. after all of the circles have been drawn). Add a “delay(250)” command to slow down the effect.

3. Write a method called Lines. This will input a y-coordinate. The effect of the method is to start by drawing a line from the position (0, 0) to the position (250, y). On the next call to itself, it draws a line from position (0, 0) to (250, y-5). On the third call, it should draw the line from (0, 0) to (250, y-10), etc. decreasing by 5 each time until reaching a stopping line from (0, 0) to (250, 0). On the way “back out”, it should draw lines from (500, 0) to (250, 5), then from (500, 0) to (250, 10), etc.
Here’s a sample driver program that may help you get started:

/* Name:
   In this project, we will practice recursion. User input determines the
x center, y center, and radius. Circles are drawn one inside the other,
decreasing the radius by 5, until the radius is obsolete. */

import cs.ssa.*;

class CirclesInCircles
{
    static public SSAWindow myScreen;

    static public void main(String[] args)
    {  // main routine; execution begins here

        // variable and object declarations
        int userX, userY, userRadius;
        myScreen = new SSAWindow();

        // getting user data
        userX = myScreen.readInt("Please enter center X coordinate:’’);
        userY = myScreen.readInt("Please enter center Y coordinate:’’);
        userRadius = myScreen.readInt("Please enter Radius:’’);

        drawCircles(userX, userY, userRadius);
    }

    static void drawCircles(int x, int y, int rad)
    {  // your code here
    }
}
Recursion Challenge Questions

1. Start with an equilateral triangle. Find the midpoint of each side. Connect those midpoints. Remove the triangle formed. In the second step, the procedure is applied to the 3 remaining triangles. Continuing in this manner, how many new triangles are removed in the fifth step?

2. Find $f(10; 5)$, given the following:
   
   $$f(x; y) = \begin{cases} 
   f(x - 2, y + 1) + 3 & \text{if } x > y \\
   f(x + 1; y - 3) + 2 & \text{if } x < y \\
   5 & \text{otherwise}
   \end{cases}$$

3. Find $f(8; 5)$, given the following:
   
   $$f(x; y) = \begin{cases} 
   f(x - 1, y + 1) - 2 & \text{if } x \text{ and } y \text{ are positive, } x \text{ is even, and } y \text{ is odd} \\
   f(y, x + 1) + 1 & \text{if } x \text{ and } y \text{ are positive, } x \text{ is odd, and } y \text{ is even} \\
   7 & \text{if } x \text{ and } y \text{ are both zero} \\
   xy & \text{otherwise}
   \end{cases}$$
Project 5: 
Tree Fractal

Specifications: Consider the problem of drawing a tree. A tree can be seen as a trunk with miniature trees branching off of it at different angles. Each of these branches can be seen as themselves trees with smaller branches, etc. An image that is generated by such a recursive process is called a fractal.

You are to write a program that will generate a fractal image of a tree. What is more, you must incorporate an aspect of randomness so that each time your fractal is drawn it produces a slightly different effect. Come up with at least two experiments that apply randomness to your drawing. For instance, the number of sub-branches drawn should differ at each instance, not all of your angles should be quite the same, and not all of your branches should be of the exact same length. For a bonus effect, incorporate color into your program so that “leaves” are drawn in a different color from the “branches.”

Design: The design for this project should focus on how randomness will be incorporated into your program. How much randomness is needed for interesting designs? How much similarity is needed to keep it looking like a tree? Consider how many branches you want to have (or a range of branches), what range of angles might be used, and what range of lengths for each branch. Consider generating a short list of Fibonacci numbers to experiment with for some of these purposes (not required). For instance, the number of sub-branches may be chosen at random from a list containing 1, 2, 3, 5, 8.
Implementation: When implementing, be sure to use “world” coordinates to generate your tree. All coordinates should be between -1.0 and +1.0. All points must be stored in a pair of parallel arrays, one for the x coordinates and one for the y coordinates. Store the points in pairs, thus you will have the array slots at 0 start with the initial point, for instance xList[0] = 0.0 and yList[0] = -0.5. The opposite endpoint of that will be determined by the program and will be stored at location 1. Any point that is in the lower, middle part of the screen should work. The height of the initial tree should be the largest that you want to see represented: for instance, 0.7. The angle of the initial tree (the trunk) should start at Math.PI/2 (or 90 degrees). The initial call to your tree-generating algorithm thus could be:

```java
double x0 = 0.0;
double y0 = -0.7;
xList[0] = x0;
yList[0] = y0;
numLines = 0;
makeTree(xList, yList, x0, y0, 0.7, Math.PI/2);
```

A simple recursive method for this may look as follows:

```java
void makeTree(double[] xList, double[] yList, double x0, double y0,
              double height, double angle)
{
    if (height >= 0.1) // sample basis case—adjust for greater detail
    {
        // add x0 to xList at numLines *2
        // add y0 to yList at numLines *2
        // determine x1 to be: x0 + cos(angle)*height;
        // determine y1 to be: y0 + sin(angle)*height;
        // add x1 to xList at numLines *2+1
        // add y1 to yList at numLines *2+1
        // increase numLines by one
        makeTree(xList, yList, x1, y1, height*3/4, angle+Math.PI/4);
        // left branch
        makeTree(xList, yList, x1, y1, height*3/4, angle-Math.PI/4);
        // right branch
    }
}
```

Then you may draw all the lines of the tree in a method called drawTree as follows:

```java
void drawTree(double[] xList, double[] yList)
{
    for (int i=0; i < numLines; i++)
    {
        // find x1 using “worldToScreenX” and xList[i*2]
        // find y1 using “worldToScreenY” and yList[i*2]
        // find x2 using “worldToScreenX” and xList[i*2+1]
        // find y2 using “worldToScreenX” and yList[i*2+1]
        drawLine(x1, y1, x2, y2);
    }
}
```
Note that this example is only a starting point. For instance, you should add more than two recursive calls, you should vary the length of the next subtree with each recursive call, and / or make the angle of the branches differs by something besides 45 degrees.

*Testing*: When your program is working, add some counters to determine the following:
- How many lines are ultimately drawn in your program? Run this several times so you get a sense of the average. Record your results.
- Likewise, how many recursive calls are made? Record your results.
- Can you determine how “deep” the recursion goes at worst?
- How big does your array need to be?
Project 5:
Snowflake Fractal

Specifications: Consider the problem of drawing a snowflake. A snowflake can be seen as a line that is tri-sected into three parts. The middle section is removed and displaced. Then the process is repeated with the remaining sections, each of which is like part of the whole. An image that is generated by such a recursive process is called a fractal.

You are to write a program that will generate a fractal image of a snowflake. What is more, you must incorporate an aspect of randomness so that each time your fractal is drawn it produces a slightly different effect. Come up with at least two experiments that apply randomness to your drawing. For instance, displacement of each part of the snowflake can be slightly different. For a bonus effect, incorporate color into your program so that a random shade of blue or gray is used for each line drawn.

Design: The design for this project should focus on how randomness will be incorporated into your program. How much randomness is needed for interesting designs? How much similarity is needed to keep it looking like a snowflake? Consider how many divisions you want to have and what length of displacement would be valid.
Implementation: When implementing, be sure to use “world” coordinates to generate your points. All coordinates should be between -1.0 and +1.0. All points must be stored in a pair of parallel arrays, one for the x coordinates and one for the y coordinates, initially empty. The first two points to be evaluated could be: (-0.5, 0.5) and (0.5, 0.5). The initial call to your tree-generating algorithm thus could be:

```java
double x0 = -0.5;
double y0 = 0.5;
double x1 = 0.5;
double y1 = 0.5;
buildSnowFlake(xList, yList, x0, y0, x1, y1);
```

Store the points in pairs, thus you will have the array slots at 0 start with the initial point and the array slots at 1 have the second point. Skipping by two will provide the list of lines. A simple recursive method for this may look as follows:

```java
void buildSnowFlake(double[] xList, double[] yList, double x0, double y0, double x5, double y5)
{
    // compute distance between x5 and y5
    if (distance < 0.1)  // sample basis case—adjust for greater detail
    {
        // add x1 to xList at numLines *2
        // add y1 to yList at numLines *2
        // add x5 to xList at numLines *2+1
        // add y5 to yList at numLines *2+1
        // increase numLines by one
    }
    else
    {
        // compute deltaX (change in x)
        // compute deltaY (change in y)
        // x2 = x1 + deltaX/3;
        // y2 = y1 + deltaY/3;
        // x3 = (x1+x5)/2 + Math.sqrt(3)/4 * (y1-y5);
        // y3 = (y1+y5)/2 + Math.sqrt(3)/4 * (x5-x1);
        // x4 = x1 + deltaX*2/3;
        // y4 = y1 + deltaY*2/3;
        buildSnowFlake(xList, yList, x1, y1, x2, y2);
        buildSnowFlake(xList, yList, x2, y2, x3, y3);
        buildSnowFlake(xList, yList, x3, y3, x4, y4);
        buildSnowFlake(xList, yList, x4, y4, x5, y5);
    }
}
```

Then you may draw all the lines of the fractal in a method called drawLines as follows:

```java
void drawLines(double[] xList, double[] yList)
{
    for (int i=0; i < numLines; i++)
    {
        // find x1 using “worldToScreenX” and xList[i*2]
        // find y1 using “worldToScreenY” and yList[i*2]
    }
}
```
// find x2 using "worldToScreenX" and xList[i*2+1]
// find y2 using "worldToScreenX" and yList[i*2+1]
drawLine(x1, y1, x2, y2);
}

Note that this example is only a starting point. For instance, you should vary the displacement of the middle section, or even try using a different number of sub-divisions.

**Testing:** When your program is working, add some counters to determine the following:
- How many lines are ultimately drawn in your program? Run this several times so you get a sense of the average. Record your results.
- Likewise, how many recursive calls are made? Record your results.
- Can you determine how “deep” the recursion goes at worst?
- How big does your array need to be?
Specifications:
In this version of the labyrinth, you will send a robot (Karel) into a maze and have it find the solution to the maze recursively. The robot will begin at 1, 1 with infinity beepers. One move will take the robot into the maze. The end of the maze is marked by a place containing two beepers. In between, the robot must try different possibilities, backing up when necessary, to find the exit of the maze. Along the way, the robot may mark places it has been by placing beepers.

Design:
Extend the Robot class so that you may add methods to traverse the maze. Plan out ahead of time the recursive algorithm(s) that are necessary. What is the base case for this problem? What are the recursive method calls that are necessary to try to reduce the scenario to the base case? Include the following methods in your design:

```java
public boolean traverse()
public boolean tryWest()
public boolean tryEast()
public boolean tryNorth()
public boolean trySouth()
```

Implementation: The following pseudocode will help you as you implement the above methods. Note that this implementation uses indirect recursion, meaning the “traverse” method invokes other methods which in turn invoke “traverse.”

```java
public boolean traverse()
{
    // set "done" to false
    // if (atHome()) (returns true if at two beepers )
    // set "done" to true
    // else if the space is not already visited
}
// place a beeper to mark the space
// done = tryNorth();
// if still not done
//   done = tryEast();
// if still not done
//   done = trySouth();
// if still not done
//   done = tryWest();
}
// return done;
}

public boolean tryWest()
{
   // face to the west
   // if (frontIsClear())
   // {
   //   move();
   //   face East
   //   if recursive call to “traverse” succeeds
   //     return true
   //   else
   //     {
   //     move one space to east, returning to previous place
   //     return false
   //   }
   // }
   //else
   // face east
   // return false by default
}

Testing: Test your solution using the worlds “maze1.kwld”, “maze2.kwld”, and “maze3.kwld”. Create your own mazes if you wish. Watch the actual path that your robot takes and document the effect of your solution. Some questions to consider are:
- How many actual spaces does the robot have to land on to get to the end?
- How many actual “move” statements are executed?
- How many recursive calls are necessary?
- Does your robot take the optimal path through the maze?
1. Find $f(5)$, given the following:
   
   \[
   f(x) = \begin{cases} 
   f(x - 1) + 2 & \text{if } x > 2 \\
   x \times x & \text{if } x == 2 \\
   0 & \text{otherwise}
   \end{cases}
   \]

2. Find $f(10, 5)$, given the following:
   
   \[
   f(x, y) = \begin{cases} 
   f(x - 2, y + 1) + x & \text{if } x > y \\
   f(x + 1, y - 3) + y & \text{if } x < y \\
   x + y & \text{otherwise}
   \end{cases}
   \]

3. Write code to define a recursive method that computes the following function:
   
   $f(n) = 0 + 2 + 4 + 6 + \ldots + 2n$

4. Write code to define a recursive method that computes the following function:
   
   $f(n) = n + n/2 + n/4 + n/8 + \ldots + 1$