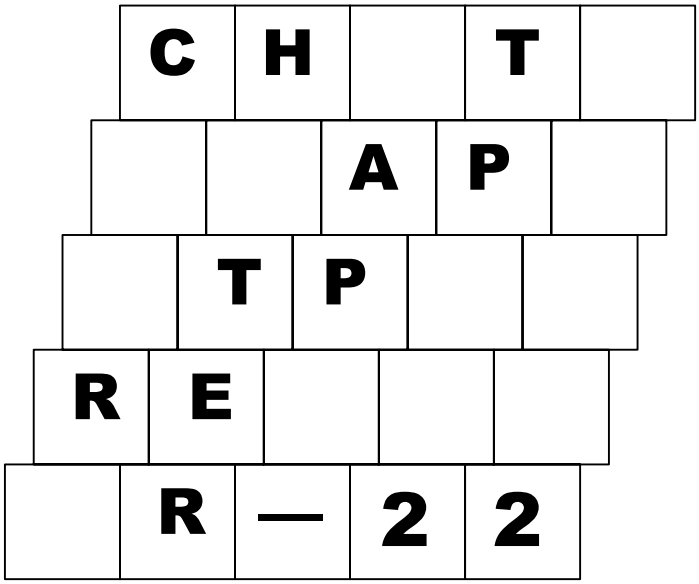

Recursion Revisited

22.1 Prologue

In this chapter we continue the discussion of recursion started in Chapter 4. Recursion is a powerful tool for handling branching processes and nested structures. Take a Java list, for example. It can contain any type of objects as values, including other lists. In an extreme case, a list even can be its own element! Java's documentation says: "While it is permissible for lists to contain themselves as elements, extreme caution is advised..." It is no coincidence that recursive structures and processes are especially common in the computer world. It is easier to implement and use a structure or a process when its substructures and subprocesses have the same form. The same function, for example, can deal with a picture and its graphics elements that are also pictures, or a list and its elements that are also lists. In Chapter 23 we will see how recursion can be used for handling binary trees, where each node is a root of a binary (sub)tree.

Recursion is not specific to Java: it works the same way with any language that allows functions to call themselves. In Java a method can call itself, or there can be a circular sequence of method calls. We saw in Section <...> that the hardware stack mechanism implements a method call in the same way whether the method calls itself or another method. All method parameters, the return address, and the local variables are kept in a separate frame on the system stack, so several methods, including several copies of the same method, can be waiting for control to be returned to them without any conflict. Multiple copies of the same method all share the same code but operate on different data.

In this chapter we will consider a few examples of recursion, discuss when the use of recursion is appropriate and when it is better to stay away from it, and learn how to debug and prove the correctness of recursive methods.

22.2 Three Examples

In Chapter 4, we discussed several recursive methods. Figure <...> on page <...>, for example, shows a recursive implementation of a `gcf` method that finds the greatest common factor of two positive integers. In that example, however, a simple non-recursive implementation exists as well. In our first example here, we take a situation where a recursive implementation is short, while a non-recursive implementation is not at all obvious.

Suppose we want to write a method that evaluates the "degree of separation" between two people in a given set of people. The degree of separation between *A* and *B* is defined as the number of links in the smallest chain of people connecting *A* and *B*, in which neighbors know each other. If *A* knows *B*, the degree of separation between *A* and *B* is 1; if *A* knows *B* and *B* knows *C*, but *A* does not know *C*, the degree of separation between *A* and *C* is 2; and so on. Suppose a person is represented by an object of a class `Person`, which has a `boolean` method `knows`, so `Person p1` "knows" `Person p2` if and only if `p1.knows(p2)` returns `true`. We want to write the following method:

```
// Returns true if and only if the degree of separation between
// p1 and p2 in the set people is less than or equal to n
public boolean degreeOfSeparation(Set<Person> people,
                                  Person p1, Person p2, int n)
```

If $n = 1$, we only need to check whether `p1` and `p2` "know" each other. If $n > 1$ we can try to find a `person` in the set that can serve as an intermediate link in a chain that connects `p1` and `p2`. It would be rather hard to implement this algorithm without recursion. But with recursion it is straightforward:

```
public boolean degreeOfSeparation(Set<Person> people,
                                  Person p1, Person p2, int n)
{
  if (n == 1)             // Base case
  {
    return p1.knows(p2);
  }
  else                    // Recursive case
  {
    for (Person p : people)
    {
      if (p1.knows(p) && degreeOfSeparation(people, p, p2, n-1))
        return true;
    }
    return false;
  }
}
```

As you know, a recursive method must have a base case (or several base cases), when no further recursion is necessary, and a recursive case (or several), where the same method is called recursively. For a recursive method to terminate properly, it must explicitly or implicitly handle the base cases. When a method calls itself recursively, it must be applied to a similar but "smaller" task that eventually converges to one of the base cases; otherwise, the recursive calls will go deeper and deeper and eventually "blow the stack."

In our `degreeOfSeparation` method, the parameter *n* determines the "size" of the task. *n* == 1 is the base case. Since that parameter in the recursive call to `degreeOfSeparation` is reduced to `n-1`, the recursive calls eventually converge to the base case and recursion stops.

The above code is deceptively short, but it may become prohibitively time-consuming. For example, if the set is split into two groups of equal sizes such that everyone knows everyone else in each group but no one from the other group, and `p1` and `p2` belong to different groups, then the total number of calls to the `knows` method will be $\frac{\left(3N^n - N^{n-1} - 2N\right)}{N-1}$, where *N* is the number of people in each group. This is exponential growth, in terms of *n*. Also, each step deeper into recursion requires a separate frame on the system stack, so this method takes $O(n)$ space on the system stack.

There are actually more efficient algorithms for this task. Even a simple divide-and-conquer trick —

```
public boolean degreeOfSeparation(Set<Person> people,
                           Person p1, Person p2, int n)
{
  if (n == 1)              // Base case
  {
    return p1.knows(p2);
  }
  else if (n == 2)         // Another base case
  {
    for (Person p : people)
    {
      if (p1.knows(p) && p.knows(p2))
        return true;
    }
    return false;
  }
  else                     // Recursive case
  {
    int m = n/2;
    for (Person p : people)
    {
      if (degreeOfSeparation(people, p1, p, m) &&
                  degreeOfSeparation(people, p, p2, n-m))
        return true;
    }
    return false;
  }
}
```

— improves things. In this version `knows` will be called $\frac{n\left(3N^{k+1}-2N^{k}-N\right)}{2\left(N-1\right)}$ times, where $n=2^{k} \Rightarrow k=\log_2 n$, which is better than exponential growth, and the amount of stack space needed is only $O\left(\log n\right)$. There is a much more efficient solution that takes time proprotional to N^2, regardless of n (see Question <...> in the end-of chapter exercises).

In our second example, let us consider a method that implements "deep" traversal of a linked list that may contain strings and similar lists as elements. In effect, it is no longer a simple list, but a whole tree. For now, let's call such a list a "composite list" (Figure 22-1).

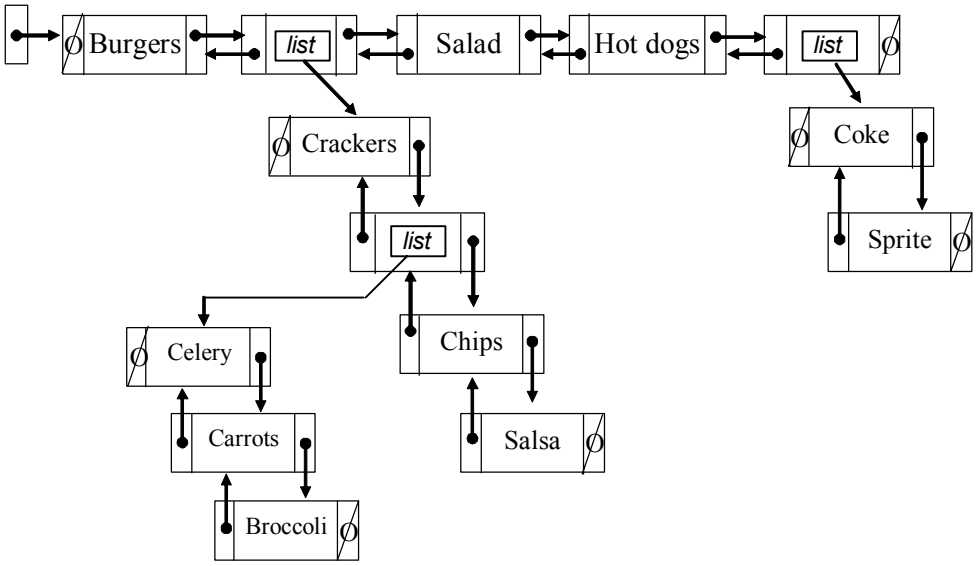

Figure 22-1. A "composite" list: some nodes hold strings, others hold lists

A more formal definition of a "composite list" is recursive: a "composite list" is a list where each element is either a string or a "composite list..." Suppose our task is to print all the strings in the list, including all the strings in all the lists that are its elements, and in all the lists that are elements of elements, and so on. Without recursion, this would be a little tricky. With recursion, it's just a few lines of code:

```
public void printAll(CompositeList list)
{
  String separator = "[";
  for (Object obj : list)
  {
    System.out.print(separator);
    if (obj instanceof String)
      System.out.print(obj);
    else  // if (obj instanceof CompositeList)
      printAll((CompositeList)obj);
    separator = ", ";
  }
  System.out.print("]");
}
```

Here the base case is when `list` contains only `String`s and no lists.

Given

```
CompositeList veggies = new CompositeList();
veggies.add ("Celery");
veggies.add ("Carrots");
veggies.add ("Broccoli");
CompositeList munchies = new CompositeList();
munchies.add ("Crackers");
munchies.add (veggies);
munchies.add ("Chips");
munchies.add ("Salsa");
CompositeList drinks = new CompositeList();
drinks.add ("Coke");
drinks.add ("Sprite");
CompositeList partyFood = new CompositeList();
partySnacks.add("Burgers");
partySnacks.add(munchies);
partySnacks.add("Salad");
partySnacks.add("Hot dogs");
partySnacks.add(drinks);

printAll(partyFood);
```

the output is

```
[Burgers, [Crackers, [Celery, Carrots, Broccoli], Chips, Salsa], Salad, Hot
dogs, [Coke, Sprite]]
```

— a recursive method generates the output with nested lists.

Actually, it would be more conventional to define a `toString` method in the `CompositeList` class itself, so that we could write

```
System.out.println(list);
```

rather than

```
printAll(list);
```

Here is how such method might look, assuming that `CompositeList` extends `ArrayList<Object>` or `LinkedList<Object>`:

```
public String toString()
{
  String s = "", separator = "[";
  for (Object obj : this)
  {
    s += separator;
    if (obj instanceof String)
      s += obj.toString();
    else  // if (obj instanceof CompositeList)
      s += ((CompositeList)obj).toString();
    separator = ", ";
  }
  s += "]";
  return s;
}
```

Now note that all this "`instanceof`" type checking is totally redundant: polymorphism takes care of it. We can just write:

```
public String toString()
{
  String s = "", separator = "[";
  for (Object obj : this)
  {
    s += separator;
    s += obj;
    separator = ", ";
  }
  s += "]";
  return s;
}
```

And this is roughly how `toString` is defined in `ArrayList` and `LinkedList`.

Now the recursion is hidden, but it kicks in when one of the elements of a list happens to be a list. It turns out an `ArrayList<Object>` or a

`LinkedList<Object>` can handle a composite list without any extra code. Later, in Chapter 26, we will see how we can use the Composite design pattern to properly define a more specific data type for the elements of a composite list.

It is possible, of course, to write the above `toString` method with no recursion, using a stack of iterators:

```
public String toString()
{
  String s = "", separator = "[";

  Stack<Iterator<Object>> stk = new Stack<Iterator<Object>>();

  Iterator<Object> iter = iterator();
  while (iter.hasNext() || !stk.isEmpty())
  {
    if (iter.hasNext())
    {
      s += separator;
      Object obj = iter.next();
      if (obj instanceof String)
      {
        s += obj;
        separator = ", ";
      }
      else
      {
        CompositeList sublist = ((CompositeList)obj);
        stk.push(iter);
        iter = sublist.iterator();
        separator = "[";
      }
    }
    else
    {
      s += "]";
      iter = stk.pop();
    }
  }
  s += "]";
  return s;
}
```

This code will produce the same result. But it is much longer, harder to understand, and more error prone, and does not take advantage of polymorphism.

Do not avoid using recursion where it works well and makes life easier.

❖ ❖ ❖

In our third example, we will generate all possible permutations of a string of characters. Suppose we are building a computer word game that tries to make a valid word out of a given set of letters. The program will require a method that generates all permutations of the letters and matches them against a dictionary of words. Suppose a string of *n* characters is stored in a `StringBuffer`. (We use the `StringBuffer` class rather than `String` to make rearranging characters more efficient — `String`s are immutable objects.) Our strategy for generating all permutations is to place each character in turn in the last place in the string, then generate all permutations of the first (*n*-1) characters. In other words, `permutations` is a recursive method. The method takes two arguments: a `StringBuffer` object and the number *n* of characters in the leading fragment that have to be permutated.

In this example, the process is branching and recursive in nature although there are no nested structures. The base case is when *n* is equal to 1 — there is nothing to do except to report the permutation.

The `permutations` method below is quite short and readable; still, it is hard to grasp why it works! We will return to it in Section 22.4, which explains the best way of understanding and debugging recursive methods.

```
private void swap(StringBuffer str, int i, int j)
{
  char c1 = str.charAt(i);
  char c2 = str.charAt(j);
  str.setCharAt(i, c2);
  str.setCharAt(j, c1);
}

public void permutations(StringBuffer str, int n)
{
  if (n <= 1)                  // Base case
  {
    System.out.println(str);   // The permutation is completed
  }
  else                         // Recursive case
  {
    for (int i = 0; i < n; i++)
    {
      swap(str, i, n-1);
      permutations(str, n-1);
      swap(str, n-1, i);
    }
  }
}
```

22.3 When Not to Use Recursion

Any recursive method can be also implemented through iterations, using a stack if necessary. This raises a question: When is recursion appropriate, and when is it better avoided?

There are some technical considerations that may restrict the use of recursive methods:

1. If a method allocates large local arrays, the program may run out of memory in recursive calls. A programmer has to have a feel for how deeply the recursive calls go; she may choose to implement her own stack and save only the relevant variables there, reusing the same temporary array.

2. When a method manipulates an object's fields, a recursive call may change their values in an unpredictable way unless the manipulation is done on purpose and thoroughly understood.

3. If efficiency is important, a method implemented without recursion may work faster.

But the most important rule is that recursion should be used only when it significantly simplifies the code without excessive performance loss. Recursion is especially useful for dealing with nested structures or branching processes. One typical example is algorithms for traversing tree structures, which are described in Chapter 23. On the other hand, when you are dealing with linear structures and processes, normally you can use simple iterations. The following test will help you decide when to use recursion and when iterations. If the method branches in two or more directions, calling itself recursively in each branch, it is justified to use recursion. If the method calls itself only once, you can probably do the same thing just as easily with iterations.

As an example, let us consider the `factorial(n)` method that calculates the product of all numbers from 1 to *n*. This method has a simple recursive form:

```
public long factorial(int n)
{
  if (n <= 1)     // Base case
    return 1;
  else            // Recursive case
    return n * factorial(n - 1);
}
```

Our test shows that `factorial`'s code has only one recursive call. We are dealing with a linear process. It should be just as easy to accomplish the same thing with iterations, thus avoiding the overhead of recursive method calls:

```
public long factorial(int n)
{
  long product = 1;
  for (int k = 2; k <= n; k++)
  {
     product *= k;
  }
  return product;
}
```

Both versions are acceptable. The recursive version takes $O(n)$ amount of space on the system stack, but the factorial of large n is far too large, anyway.

A more pernicious example is offered by the famous Fibonacci Numbers. These are defined as a sequence where the first two numbers are equal to one, with each consecutive number equal to the sum of the two preceding numbers:

1, 1, 2, 3, 5, 8, 13, ...

Mathematically this is a recursive definition:

$F_1 = 1;\ F_2 = 1;$
$F_n = F_{n-1} + F_{n-2}$ (for $n > 2$).

It can be easily converted into a recursive method:

```
// Precondition: n >= 1
public long fibonacci(int n)
{
  if (n == 1 || n == 2)  // Base case
    return 1;
  else                   // Recursive case
    return fibonacci(n-1) + fibonacci(n-2);
}
```

It may seem, at first, that this method meets our test of having more than one recursive call to `fibonacci`. But in fact, there is no branching here: `fibonacci` simply recalls two previous members in the same linear sequence. Don't be misled by the innocent look of this code. The first term, `fibonacci(n-1)`, will recursively call `fibonacci(n-2)` and `fibonacci(n-3)`. The second term, `fibonacci(n-2)`, will call (again) `fibonacci(n-3)` and `fibonacci(n-4)`. The

`fibonacci` calls will start multiplying like rabbits.* To calculate the *n*-th Fibonacci number, F_n, `fibonacci` will actually make more than F_n recursive calls, which, as we will see in the following section, may be quite a large number.

On the other hand, the same method implemented iteratively will need only *n* iterations:

```
public long fibonacci(int n)
{
  long f1 = 1, f2 = 1, next;
  while (n > 2)
  {
    next = f1 + f2;
    f1 = f2;
    f2 = next;
    n--;
  }
  return f2;
}
```

For our final example of when recursion is inappropriate, let us consider Selection Sort of an array of *n* elements. As you know, the idea is to find the largest element and swap it with the last element, then apply the same method to the array of the first *n*–1 elements. This can be done recursively:

```
public void selectionSort(int[] a, int n)
{
  if (n == 1)     // Base case: array of length 1 -- nothing to do
      return;
  else
  {
    // Find the index of the largest element:
    int iMax = 0;
    for (int i = 1; i < n; i++)
      if (a[iMax] < a[i])
        iMax = i;

    // Swap it with the last element:
    int aTemp = a[n-1]; a[n-1] = a[iMax]; a[iMax] = aTemp;

    // Call selectionSort for the first n-1 elements:
    selectionSort(a, n-1);
  }
}
```

* The numbers are named after Leonardo Pisano (Fibonacci) who invented the sequence in 1202, as part of an effort to develop a model for the growth of a population of rabbits.

This is a case of so-called *tail recursion*, where the recursive call is the last statement in the method: only the return from the method is executed after that call. Therefore, by the time of the recursive call, the local variables (except the parameters passed to the recursive call) are no longer needed. A good optimizing compiler will detect this situation and, instead of calling `selectionSort` recursively, will update the parameter `n` and pass control back to the beginning of the method. Or, we can easily do it ourselves:

```
public void selectionSort(int v[], int n)
{
  while (n > 1)
  {
    // Find the index of the largest element:
    int iMax = 0;
    for (int i = 1; i < n; i++)
      if (v[iMax] < v[i])
        iMax = i;

    // Swap it with the last element:
    int vTemp = v[n-1]; v[n-1] = v[iMax]; v[iMax] = vTemp;

    n--;
  }
}
```

To quote Niklaus Wirth, the inventor of the Pascal programming language,

> In fact, the explanation of the concept of recursive algorithm by such inappropriate examples has been a chief cause of creating widespread apprehension and antipathy toward the use of recursion in programming, and of equating recursion with inefficiency.*

22.4 Understanding and Debugging Recursive Methods

A common way of understanding and debugging non-recursive methods is to trace, either mentally or with a debugger, the sequence of statements and method calls in the code. Programmers may also insert some debugging print statements that will report to them the method's progress and the intermediate values of variables.

These conventional methods are very hard to apply to recursive methods, because it is difficult to keep track of your current location in the hierarchy of recursive calls. Getting to the bottom of the recursive process requires a detailed examination of the system stack — a tedious and useless activity. Instead of such futile attempts,

* Niklaus Wirth, *Algorithms + Data Structures = Programs*. Prentice Hall, 1976.

recursive methods can be more easily understood and analyzed with the help of a method known as *mathematical induction*.

In a nutshell, mathematical induction works as follows. Suppose we have a sequence of statements

$P_1, P_2, \ldots, P_n, \ldots$

Suppose that:

1. We can show that P_1 is true (the base case);
2. We can prove that, for any $n \geq 2$, if $P_1, \ldots, P_{n-1}$ are true (*induction hypothesis*), then P_n is also true.

Then, if both conditions are met, we can conclude that all statements in the sequence are true.

This is so because P_1 implies P_2, P_1 and P_2 imply P_3, and so on. However, we do not have to go through the entire logical sequence for every step. Instead, we can take a shortcut and just say that all the statements are true by mathematical induction.

As an exercise in mathematical induction, let us estimate the running time for the recursive `fibonacci` method discussed in the previous section:

```
public long fibonacci(int n)
{
  if (n == 1 || n == 2)  // Base case
    return 1;
  else                   // Recursive case
    return fibonacci(n-1) + fibonacci(n-2);
}
```

We will prove that `fibonacci(n)` requires not less than $(3/2)^{n-2}$ calls to the method. This is true for $n = 1$ and $n = 2$ (base cases), which both require just one call:

$n = 1$: $1 > (3/2)^{1-2} = (3/2)^{-1} = 2/3$;
$n = 2$: $1 = (3/2)^{2-2} = (3/2)^0$

For any n > 2, in addition to the initial call, the method calls `fibonacci(n-1)` and `fibonacci(n-2)`. From the induction hypothesis, the total number of times `fibonacci` is called in `fibonacci(n-1)` is not less than $(3/2)^{n-3}$, and the total number of times fibonacci is called in `fibonacci(n-2)` is not less than $(3/2)^{n-4}$. So the total number of `fibonacci` calls in `fibonacci(n)` is not less than:

$$1 + (3/2)^{n-3} + (3/2)^{n-4} > (3/2)^{n-3} + (3/2)^{n-4} = (3/2)^{n-4}(3/2 + 1) =$$

$$(3/2)^{n-4} \cdot (5/2) > (3/2)^{n-4} \cdot (3/2)^2 = (3/2)^{n-2}, \text{ q.e.d.}$$

Assuming that a reasonably fast computer can execute a hundred million calls per second (and that we somehow manage to represent very large Fibonacci numbers in memory), `fibonacci(100)` would run for over $(3/2)^{98} / 10^8$ seconds, or more than 57 years! (The iterative implementation, by contrast, would run in less than one microsecond.)

You may notice a close conceptual link between recursion and mathematical induction. The key feature of mathematical induction is that we do not have to trace the sequence of statements to the bottom. We just have to first prove the base case and then, for an arbitrary *n*, show that if the induction hypothesis is true at all previous levels, then it is also true at the *n*-th level.

Let us see how mathematical induction applies to the analysis of code in recursive methods. As an example, let's take the `permutations` method from Section 22.2, which generates all permutations of a string of characters:

```
public void permutations(StringBuffer str, int n)
{
  if (n <= 1)                  // Base case:
  {
    System.out.println(str);   // The permutation is completed
  }
  else                         // Recursive case:
  {
    for (int i = 0; i < n; i++)
    {
      swap(str, i, n-1);
      permutations(str, n-1);
      swap(str, n-1, i);
    }
  }
}
```

We will prove two facts about this code using mathematical induction:

1. `permutations` returns the string to its original order when it is finished.
2. `permutations(str, n)` generates all permutations of the first *n* characters.

In the base case, $n = 1$, the method just reports the string and does nothing else — so both statements are true. Let us assume that both statements are true for any level below n (induction hypothesis). Based on that assumption let us prove that both statements are also true at the level n.

In the recursive case, the method swaps `str[i]` and `str[n-1]`, then calls `permutations(str, n-1)`, then swaps back `str[n-1]` and `str[i]`. By the induction hypothesis, `permutations(str, n-1)` preserves the order of characters in `str`. The two swaps cancel each other. So the order of characters is not changed in `permutations(str, n)`. This proves Statement 1.

In the `for` loop we place every character of the string, in turn, at the end of the string. (This is true because the index `i` runs through all values from `0` to `n-1` and, as we have just shown, the order of characters does not change after each iteration through the loop.) With each character placed at the end of the string we call `permutations(str, n-1)`, which, by the induction hypothesis, generates all permutations of the first n-1 characters. Therefore, we combine each character placed at the end of the string with all permutations of the first n-1 characters, which generates all permutations of n characters. This proves Statement 2.

The above example demonstrates how mathematical induction helps us understand and, with almost mathematical rigor, prove the correctness of recursive methods. By comparison, conventional code tracing and debugging and attempts at unfolding recursive calls to the very bottom are seldom feasible or useful.

22.5 *Lab:* The Tower of Hanoi

The "Tower of Hanoi" puzzle is probably the most famous example of recursion in computer science courses. The puzzle has three pegs, with several disks on the first peg. The disks are arranged in order of decreasing diameter from the largest disk on the bottom to the smallest on top. The rules require that the disks be moved from peg to peg, one at a time, and that a larger disk never be placed on top of a smaller one. The objective is to move the whole tower from the first peg to the second peg.

The puzzle was invented by French mathematician François Edouard Anatole Lucas[lucas] and published in 1883. The "legend" that accompanied the game stated that in Benares, India, there was a temple with a dome that marked the center of the world. The Hindu priests in the temple moved golden disks between three diamond needles. God placed 64 gold disks on one needle at the time of creation and, according to the "legend," the universe will come to an end when the priests have moved all 64 disks to another needle.

There are hundreds of Java applets on the Internet that move disks from peg to peg, either with animation or interactively.[hanoi] In this lab, fancy display is not required.

1. Write a program that solves the puzzle and prints out the required moves for a given number of disks.

2. Examine the number of moves required for 1, 2, 3, etc. disks, find the pattern, and come up with a formula for the minimum number of moves required for *n* disks. Prove the formula using the method of mathematical induction. Estimate how much time it will take your program to move a tower of 64 disks.

22.6 *Case Study and Lab:* The Game of Hex

The game of Hex was first invented in 1942 by Piet Hein, a Danish mathematician. (The same game was apparently reinvented independently a few years later by John Nash, then a graduate student at Princeton, who later won a Nobel prize in economics.) Martin Gardner made the game popular when he described it in his *Scientific American* article in the late 1950s and in a later book.

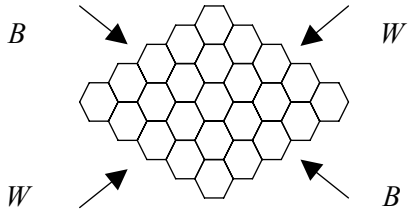

Figure 22-2. A 5 by 5 Hex board

Hex is played on a rhombic board with hexagonal fields, like a honeycomb. A common board size is 11 by 11; Figure 22-2 shows a smaller 5 by 5 board. The game starts with an empty board. Each of the two players, *B* and *W*, is assigned a pair of opposite sides of the board. For example, *B* gets northwest and southeast, and *W* gets northeast and southwest. *B* has a pile of black stones and *W* has a pile of white stones. Players take turns placing a stone of their color on an empty field. A player who first connects his sides of the board with a contiguous chain of his stones wins.

An interesting property of Hex is that a game can never end in a tie: when the board is filled, either *B* has his sides connected or *W* has his sides connected, always one or the other, but never both. Like all finite strategy games with no repeating positions, Hex has a winning strategy for one of the players. In this case it's the first player (it never hurts to have an extra stone of your color on the board). But this winning strategy is hard to find, because the number of all possible positions is large. For a smaller board, a computer can calculate it. It is not too difficult to write such a program, relying on a recursive algorithm. However (unless you have a lot of free time) in this lab our task is more modest: only to decide whether a given Hex position is a win for one of the players.

A human observer can glance at a Hex board and immediately tell whether one of the players has won. Not so in a computer program: it takes some computations to find out whether there is a chain of stones of a particular color connecting two opposite sides of the board. Our task is to develop an algorithm and write a method that does this.

But first we have to somehow represent a Hex board position in the computer. It is not very convenient to deal with rhombuses and hexagons in a program. Fortunately, an equivalent board configuration can be achieved on a regular square board, represented by a 2-D array. Each inner field on a Hex board has six neighbors; border fields have four, and corner fields have two or three. We can emulate the same configuration on a square board using the appropriate designation of "logical" neighbors. Figure 22-3 shows which squares are supposed to be "neighbors": the rows of a 2-D array are slightly shifted so that "neighbors" share a piece of border. Basically, a square at (*row*, *col*) has neighbors at (*row*-1, *col*-1), (*row*-1, *col*), (*row*, *col*-1), (*row*, *col*+*1*), (*row*+1, *col*), and (*row*+1, *col*+1), excluding those positions that fall outside the array.

Write a class `HexBoard` that represents a board position. You can represent stones of different colors using `char`s, `int`s, `String`s, or `Color`s. If you use `char`s (for example, `'b'` for black, `'w'` for white, and space for an empty square), then you can conveniently derive your `HexBoard` class from the `CharMatrix` class that you wrote for the *Chomp* project in Chapter 12. To hide the implementation details, your `HexBoard` class should provide the modifiers `setBlack(row, col)` and `setWhite(row, col)` and the `boolean` accessors `isBlack(row, col)` and `isWhite(row, col)`. Also provide a `toString` method, which will allow us to print the board, each row on a separate line. It is convenient to use a private

`boolean` method `isInBounds(row, col)` to determine whether (*row*, *col*) refers to a legal square on the board.

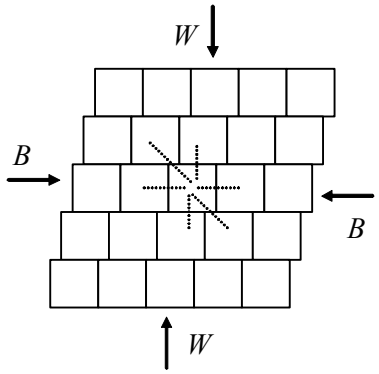

Figure 22-3. A Hex board represented as a 2-D array

It is probably easier to write a separate method that detects a win for each of the players. You need to implement only one of them. For example:

```
/**
 *  Returns true if there is a contiguous chain of black stones
 *  that starts in col 0 and ends in the last column of the board;
 *  otherwise returns false.
 */
public boolean blackHasWon()
```

At first, the task may appear trivial: just try starting at every black stone in the first column and follow a chain of black stones to the last column. But a closer look at the problem reveals that chains can branch in tricky ways (Figure 22-4), and there is no obvious way to trace all of them. What we really need is to find a "blob" of connected black stones that touches both the left and the right sides of the board.

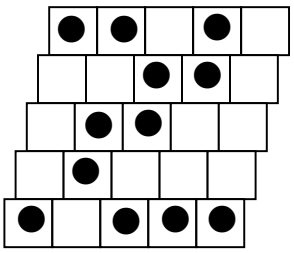

Figure 22-4. Hex: detecting a win for black stones

The task of finding a connected "blob" is known as an "area fill" task. A very similar situation occurs when you need to flood a connected area in a picture, replacing one color with another, or to fill the area inside a given contour (see Question <...> in the exercises for this chapter). The only difference is that usually a pixel or a cell has four neighbors (or eight, if you count the diagonal neighbors), while in *Hex* a square on the board has six neighbors.

Since the black and white colors are taken, we will be filling the area in "gray," so add private `setGray(row, col)` and `isGray(row, col)` methods to your `HexBoard` class.

1. Write a method `areaFill(int row, int col)`. This method should check whether the (*row*, *col*) square is in bounds and holds a black stone. If both conditions are true, `areaFill` should color in gray the largest contiguous black area that contains the (*row*, *col*) square. To do that, set the color at (*row*, *col*) to gray, then call `areaFill` recursively for each of its six neighbors. Make the `areaFill` method public, so that you can test it from `main`.
2. In the `blackHasWon` method:
    - 2.1 call `areaFill` for each square in the first column;
    - 2.2 count the gray stones in the last column to see if any of the gray blobs touch the last column;
    - 2.3 Restore all gray stones on the board back to black and return `true` or `false`, appropriately.

❖ ❖ ❖

You might wonder whether there is a way to look for a win while doing `areaFill` and quit as soon as you reach the rightmost column. This is possible, of course. However, quitting a recursive method for good is not so easy: your original call may be buried under a whole stack of recursive calls and you need to properly quit all of them. Two approaches are possible.

First approach: make `areaFill` return a `boolean` value: `true` if it `touched` the rightmost column, `false` if it didn't. Then, if one of the recursive calls to `areaFill` returns `true`, skip the rest of the recursive calls and immediately return `true`. Something like this:

```
if (areaFill(row-1, col-1) == true)
  return true;
if (areaFill(row-1, col) == true)
  return true;
...
```

Second approach: define a field in your class (a local variable won't do!) and set it to `true` as soon as `areaFill` touches the rightmost column. Something like:

```
if (col == numCols() - 1)
  won = true;
if (!won)
  areaFill(row-1, col-1);
if (!won)
  areaFill(row-1, col);
...
```

Note how treacherous recursion may be: if you write what seems to be less redundant code —

```
if (col == numCols() - 1)
  won = true;
if (!won)
{
  areaFill(row-1, col-1);
  areaFill(row-1, col);
  ...
}
...
```

— you will lose all the benefits of quitting early and your code will dramatically slow down!

"For extra credit," make `blackHasWon` work using one of the above two variations of a more efficient method that quits as soon as it detects a win. (Note that with this change, the `areaFill` method does not necessarily fill the area completely, so it might be a good idea to rename it into something more appropriate, say `findWin`.)

Test your class using a test program `Hex.java` and a sample data file `hex.dat` provided in `JM\Ch22\Hex`. You will also need the `CharMatrix` class from `JM\Ch12\Chomp`, if you used it.

22.7 Summary

Recursion is a programming technique based on methods calling themselves.

Recursive method calls are supported by the system stack, which keeps the method parameters, return address, and local variables in a separate frame for each call.

Recursion is useful for dealing with nested structures or branching processes where it helps to create short, readable, and elegant code that would otherwise be impossible.

Recursion is not essential in situations that deal with linear structures or processes, which can be as easily and more efficiently implemented with iterations.

The best way to understand and analyze recursive methods is by thinking about them along the lines of *mathematical induction*; attempts at unfolding and tracing recursive code "to the bottom" usually fail, except in simple exercises such as Question <...> below.

Exercises

1. What is the output from the following method when called with $n = 3$? ✓

```
public void printX(int n)
{
  if (n <= 0)
    System.out.print(0);
  else
  {
    printX(n - 1);
    System.out.print(n);
    printX(n - 2);
  }
}
```

2. Consider

```
public void enigma(int n)
{
  for (int i = 0; i < n; i++)
    enigma(i);
  System.out.print(n);
}
```

Does the call `enigma(3)` terminate? If so, what is the output?

3. What is the output from the following method when called with the argument $x = 2009$? ✓

```
public void display(int x)
{
  if (x >= 10)
  {
    display(x/10);
    System.out.print(x % 10);
  }
}
```

4. Fill in the blanks in the following recursive method:

```
// Returns the value of the largest element among the first
// n elements in vector v.
// Precondition: 1 <= n <= v.length
public double max(double[] v, int n)
{
  double m = v[n-1], m2;

  if ( _____________________________ )
  {
    m2 = ____________________________________________________ ;

    if (m2 > m)
      m = m2;
  }
  return m;
}
```

5.■ Rewrite the `degreeOfSeparation` method from Section 22.2 without recursion. ☹ Hint: start with a set that contains only `p1` and keep expanding it, adding on each iteration all the people who know anyone from that set. Iterate `people.size()` times or until you find `p2` in the set of acquaintances. ☺.

6. A positive integer is evenly divisible by 9 if and only if the sum of all its digits is divisible by 9. Suppose you have a method `sumDigits` that returns the sum of the digits of a positive integer *n*. Fill in the blanks in the method `isDivisibleBy9(int n)` that returns `true` if `n` is divisible by 9 and `false` otherwise. Your method may use the assignment and relational operators and `sumDigits`, but no arithmetic operators (`*`, `/`, `%`, `*=`, `/=`, `%=`) are allowed. ✓

```
// Returns n % 9 == 0
// Precondition: n > 0
public boolean isDivisibleBy9(int n)
{
  if (_____________________________________)

    return ____________________________________ ;

  else if (_____________________________________________)

    return ____________________________________ ;

  else
    return ____________________________________ ;
}
```

7. The method below attempts to calculate x^n economically:

```
public double pow(double x, int n)
{
  double y;
  if (n == 1)
    y = x;
  else
    y = pow(x, n/2) * pow(x, n - n/2); // Line 7
  return y;
}
```

(a) How many multiplications will be executed when `pow(1.234, 5)` is called? ✓

(b) How many multiplications will be executed if we replace Line 7 above with the following statements?

```
{ y = pow(x, n/2); y *= y; if (n % 2 != 0) y *= x; }
```

(c) How many multiplications will Version (a) above take to calculate `pow(1.234, 9)`? Version (b)?

8. A linked list has four nodes containing the values 10, 20, 30, and 40 (in that order) and is defined by a reference to its first node, `head`. The method `doTheTrick` below returns a reference to the head of the changed list. How many nodes will the changed list have and what values will be stored in them? ✓

```
public ListNode doTheTrick(ListNode head)
{
  if (head == null || head.getNext() == null)
    return head;

  ListNode newHead = head.getNext();
  newHead = doTheTrick(newHead);

  head.setNext(newHead.getNext());
  newHead.setNext(head);
  return newHead;
}
```

9. What is the return value of `mysterySum(10)`, where

```
public int mysterySum(int n)
{
  if (n == 1)
    return 1;
  else
    return mysterySum(n - 1) + 2*n - 1;
}
```

Justify your answer by using mathematical induction. Explain why this is an inappropriate use of recursion. ✓

10.◆ In the `degreeOfSeparation` example in Section 22.2, prove using math induction the formulas given for the numbers of calls to `knows` in both versions of the method.

11. Write a recursive method

```
public String removeDuplicateChars(String str)
```

that takes a string and returns a new string with all duplicate consecutive occurrences of the same character removed. For example, `removeDuplicateChars("OCCURRRENCE")` should return `"OCURENCE"`. Do not use any iterative statements.

12.■ Fill in the blanks in the following method:

```
// Returns the number of all possible paths from the
// point(0,0) to the point(x,y), where x and y are
// any non-negative integers.  From any point the path
// may extend only down or to the right by one unit
// (i.e., one of the current coordinates x or y can be
// incremented by one).
public long countPaths(int x, int y)
{
  if (x == 0 || y == 0)
    return _________________;
  else
    return ____________________________________________ ;
}
```

13.■ Rewrite the Cookie Monster program (Question <...> from Chapter 19) using recursion. For recursive handling, it often helps to restate the question in more general terms. Here we need to refer to the optimal path from (0,0) to any position (*row*, *col*). So our `optimalPath` method should now take two parameters: `row` and `col`. Note that since the monster can only move down and to the right, the maximum number of cookies accumulated at a position (*row*, *col*) is related to the previous positions as follows:

```
optimalPath(row, col) = cookies[row][col] +
    the larger of the two:
    {optimalPath(row-1, col), optimalPath(row, col-1)}
```

The only problem is invalid positions: either out of bounds or "barrels." How can we define `optimalPath` for an invalid position (*row*, *col*), so that the above formula still works? Identify the base case(s) and recursive case(s).

14.■ A "pool" is an irregularly shaped contiguous blob of empty cells (with `null` values) in a two-dimensional array. Two cells in a blob are considered connected if they share a side. The pool is completely surrounded by a "white wall," a connected contour of cells with `Color.WHITE` values. Write a method

```
public void fillPool(Color[][] plane, int row, int col)
```

that takes an empty location `(row, col)` inside a pool in `plane` and fills all the cells in that pool with `Color.BLUE` values.

15.■ On Beavis Island, the alphabet consists of three letters, A, B, and H — but no word may have two A's in a row. Fill in the blanks in the following recursive method `allWords`, which prints out all Beavis Island words of a given length:

```
/**
 *  Prints all Beavis Island words that have
 *  length letters.  word contains the initial
 *  sequence of letters in a word that is being built
 *  (use an empty string buffer of capacity length
 *  when calling allWords from main).
 */
public void allWords(StringBuffer word, int length)
{
  if (length == word.length())  // base case
  {
    // Display the string:
    ____________________________________;
  }
  else  // recursive case
  {
    int k = word.length();
    word.append('*'); // reserve room for one char

    // Append 'A' only if last letter is not an 'A':
    if (k == 0 || word.charAt(k-1) != 'A')
    {
      word.setCharAt(k, 'A');
      ____________________________________ ;
    }
    ____________________________________; // append 'B'
    ____________________________________;

    ____________________________________; // append 'H'
    ____________________________________;

    ____________________________________;
  }
}
```

≤ Hint: you might need `StringBuffer`'s `setLength` method, which truncates the string to a specified length. ≥

16.◆ Suppose we have a list of positive integers. We want to choose several of them so that their sum is as large as possible but does not exceed a given limit. This type of problem is called a "Knapsack Problem." For example, we may want to choose several watermelons at the market so that their total weight is as large as possible but does not exceed the airline limit for one bag.

(a) Write a recursive method that solves a simplified Knapsack Problem: it only calculates the optimal sum but does not report the selected items:

```
/**
 *  w contains n positive integers (n <= w.length).
 *  Returns the sum of some of these integers such that
 *  it has the largest possible value
 *  without exceeding limit.
 */
public int knapsackSum(int[] w, int n, int limit)
```

Use mathematical induction to prove that your code is correct.

(b) Write a more complete version that builds a stack of the values selected for the optimal sum. Pass a `Stack` of `Integer` objects to the method as the fourth argument. The stack should be initially empty when the method is called from `main` (or from another method).

(c)◆ Can you think of an alternative algorithm that uses neither recursion nor a stack?