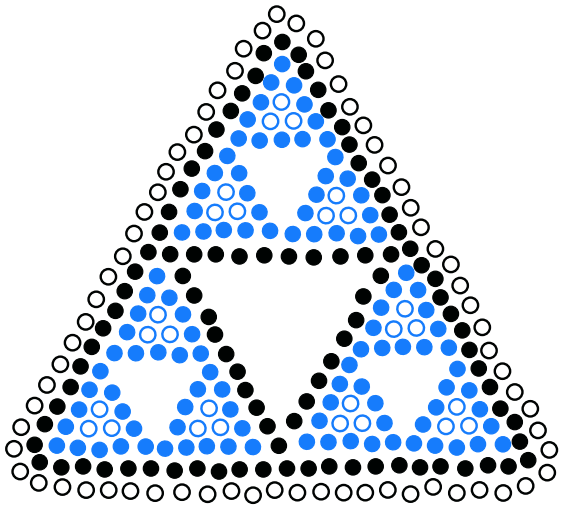
Chapter 5 Recursion



Contents

5.1 Illustrative Examples 191
5.1.1 The Factorial Function 191
5.1.2 Drawing an English Ruler 193
5.1.3 Binary Search 196
5.1.4 File Systems 198
5.2 Analyzing Recursive Algorithms 202
5.3 Further Examples of Recursion 206
5.3.1 Linear Recursion 206
5.3.2 Binary Recursion 211
5.3.3 Multiple Recursion 212
5.4 Designing Recursive Algorithms 214
5.5 Recursion Run Amok 215
5.5.1 Maximum Recursive Depth in Java 218
5.6 Eliminating Tail Recursion 219
5.7 Exercises 221

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

One way to describe repetition within a computer program is the use of loops, such as Java's while-loop and for-loop constructs described in Section 1.5.2. An entirely different way to achieve repetition is through a process known as ***recursion***.

Recursion is a technique by which a method makes one or more calls to itself during execution, or by which a data structure relies upon smaller instances of the very same type of structure in its representation. There are many examples of recursion in art and nature. For example, fractal patterns are naturally recursive. A physical example of recursion used in art is in the Russian Matryoshka dolls. Each doll is either made of solid wood, or is hollow and contains another Matryoshka doll inside it.

In computing, recursion provides an elegant and powerful alternative for performing repetitive tasks. In fact, a few programming languages (e.g., Scheme, Smalltalk) do not explicitly support looping constructs and instead rely directly on recursion to express repetition. Most modern programming languages support functional recursion using the identical mechanism that is used to support traditional forms of method calls. When one invocation of the method makes a recursive call, that invocation is suspended until the recursive call completes.

Recursion is an important technique in the study of data structures and algorithms. We will use it prominently in several later chapters of this book (most notably, Chapters 8 and 12). In this chapter, we begin with the following four illustrative examples of the use of recursion, providing a Java implementation for each.

- The ***factorial function*** (commonly denoted as $n!$) is a classic mathematical function that has a natural recursive definition.
- An ***English ruler*** has a recursive pattern that is a simple example of a fractal structure.
- ***Binary search*** is among the most important computer algorithms. It allows us to efficiently locate a desired value in a data set with upwards of billions of entries.
- The ***file system*** for a computer has a recursive structure in which directories can be nested arbitrarily deeply within other directories. Recursive algorithms are widely used to explore and manage these file systems.

We then describe how to perform a formal analysis of the running time of a recursive algorithm, and we discuss some potential pitfalls when defining recursions. In the balance of the chapter, we provide many more examples of recursive algorithms, organized to highlight some common forms of design.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.1 Illustrative Examples

5.1.1 The Factorial Function

To demonstrate the mechanics of recursion, we begin with a simple mathematical example of computing the value of the ***factorial function***. The factorial of a positive integer n, denoted $n!$, is defined as the product of the integers from 1 to n. If $n = 0$, then $n!$ is defined as 1 by convention. More formally, for any integer $n \geq 0$,

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1 & \text{if } n \geq 1. \end{cases}$$

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$. The factorial function is important because it is known to equal the number of ways in which n distinct items can be arranged into a sequence, that is, the number of ***permutations*** of n items. For example, the three characters `a`, `b`, and `c` can be arranged in $3! = 3 \cdot 2 \cdot 1 = 6$ ways: `abc`, `acb`, `bac`, `bca`, `cab`, and `cba`.

There is a natural recursive definition for the factorial function. To see this, observe that $5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) = 5 \cdot 4!$. More generally, for a positive integer n, we can define $n!$ to be $n \cdot (n-1)!$. This ***recursive definition*** can be formalized as

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot (n-1)! & \text{if } n \geq 1. \end{cases}$$

This definition is typical of many recursive definitions of functions. First, we have one or more ***base cases***, which refer to fixed values of the function. The above definition has one base case stating that $n! = 1$ for $n = 0$. Second, we have one or more ***recursive cases***, which define the function in terms of itself. In the above definition, there is one recursive case, which indicates that $n! = n \cdot (n-1)!$ for $n \geq 1$.

A Recursive Implementation of the Factorial Function

Recursion is not just a mathematical notation; we can use recursion to design a Java implementation of the factorial function, as shown in Code Fragment 5.1.

```
1  public static int factorial(int n) throws IllegalArgumentException {
2    if (n < 0)
3      throw new IllegalArgumentException( );     // argument must be nonnegative
4    else if (n == 0)
5      return 1;                                  // base case
6    else
7      return n * factorial(n-1);                 // recursive case
8  }
```

Code Fragment 5.1: A recursive implementation of the factorial function.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

This method does not use any explicit loops. Repetition is achieved through repeated recursive invocations of the method. The process is finite because each time the method is invoked, its argument is smaller by one, and when a base case is reached, no further recursive calls are made.

We illustrate the execution of a recursive method using a ***recursion trace***. Each entry of the trace corresponds to a recursive call. Each new recursive method call is indicated by a downward arrow to a new invocation. When the method returns, an arrow showing this return is drawn and the return value may be indicated alongside this arrow. An example of such a trace for the factorial function is shown in Figure 5.1.

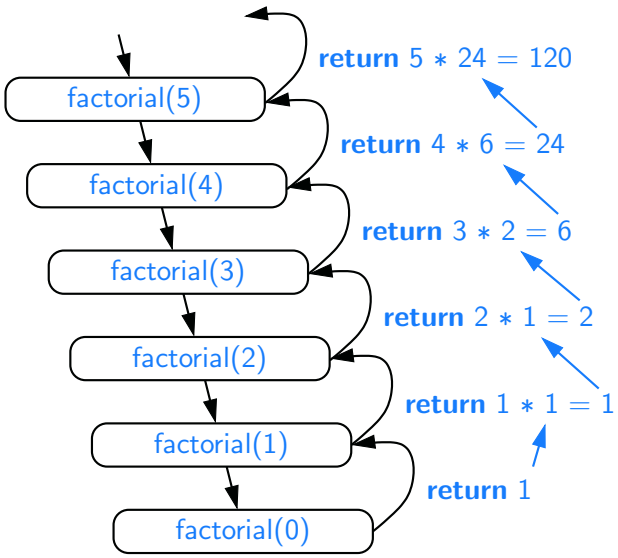


Figure 5.1: A recursion trace for the call factorial(5).

A recursion trace closely mirrors a programming language's execution of the recursion. In Java, each time a method (recursive or otherwise) is called, a structure known as an ***activation record*** or ***activation frame*** is created to store information about the progress of that invocation of the method. This frame stores the parameters and local variables specific to a given call of the method, and information about which command in the body of the method is currently executing.

When the execution of a method leads to a nested method call, the execution of the former call is suspended and its frame stores the place in the source code at which the flow of control should continue upon return of the nested call. A new frame is then created for the nested method call. This process is used both in the standard case of one method calling a different method, or in the recursive case where a method invokes itself. The key point is to have a separate frame for each active call.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.1.2 Drawing an English Ruler

In the case of computing the factorial function, there is no compelling reason for preferring recursion over a direct iteration with a loop. As a more complex example of the use of recursion, consider how to draw the markings of a typical English ruler. For each inch, we place a tick with a numeric label. We denote the length of the tick designating a whole inch as the ***major tick length***. Between the marks for whole inches, the ruler contains a series of ***minor ticks***, placed at intervals of $1/2$ inch, $1/4$ inch, and so on. As the size of the interval decreases by half, the tick length decreases by one. Figure 5.2 demonstrates several such rulers with varying major tick lengths (although not drawn to scale).

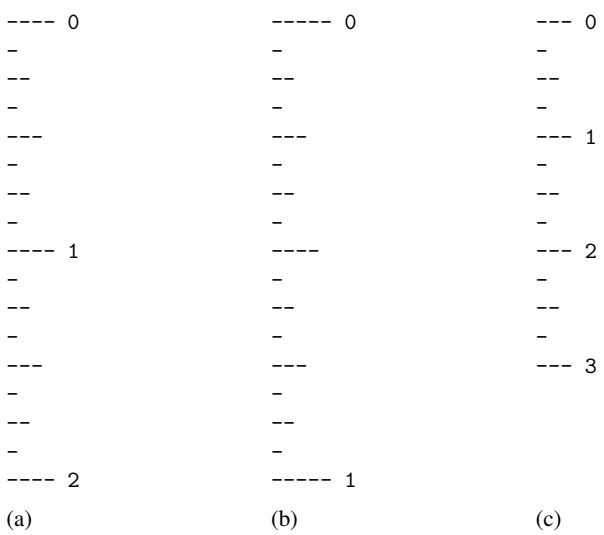


Figure 5.2: Three sample outputs of an English ruler drawing: (a) a 2-inch ruler with major tick length 4; (b) a 1-inch ruler with major tick length 5; (c) a 3-inch ruler with major tick length 3.

A Recursive Approach to Ruler Drawing

The English ruler pattern is a simple example of a ***fractal***, that is, a shape that has a self-recursive structure at various levels of magnification. Consider the rule with major tick length 5 shown in Figure 5.2(b). Ignoring the lines containing 0 and 1, let us consider how to draw the sequence of ticks lying between these lines. The central tick (at 1/2 inch) has length 4. Observe that the two patterns of ticks above and below this central tick are identical, and each has a central tick of length 3.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

In general, an interval with a central tick length $L \geq 1$ is composed of:

- An interval with a central tick length $L-1$
- A single tick of length L
- An interval with a central tick length $L-1$

Although it is possible to draw such a ruler using an iterative process (see Exercise P-5.29), the task is considerably easier to accomplish with recursion. Our implementation consists of three methods, as shown in Code Fragment 5.2.

The main method, drawRuler, manages the construction of the entire ruler. Its arguments specify the total number of inches in the ruler and the major tick length. The utility method, drawLine, draws a single tick with a specified number of dashes (and an optional integer label that is printed to the right of the tick).

The interesting work is done by the recursive drawInterval method. This method draws the sequence of minor ticks within some interval, based upon the length of the interval's central tick. We rely on the intuition shown at the top of this page, and with a base case when $L = 0$ that draws nothing. For $L \geq 1$, the first and last steps are performed by recursively calling drawInterval$(L-1)$. The middle step is performed by calling method drawLine(L).

```
1  /** Draws an English ruler for the given number of inches and major tick length. */
2  public static void drawRuler(int nInches, int majorLength) {
3    drawLine(majorLength, 0);                 // draw inch 0 line and label
4    for (int j = 1; j <= nInches; j++) {
5      drawInterval(majorLength - 1);          // draw interior ticks for inch
6      drawLine(majorLength, j);               // draw inch j line and label
7    }
8  }
9  private static void drawInterval(int centralLength) {
10   if (centralLength >= 1) {                 // otherwise, do nothing
11     drawInterval(centralLength - 1);        // recursively draw top interval
12     drawLine(centralLength);                // draw center tick line (without label)
13     drawInterval(centralLength - 1);        // recursively draw bottom interval
14   }
15 }
16 private static void drawLine(int tickLength, int tickLabel) {
17   for (int j = 0; j < tickLength; j++)
18     System.out.print("-");
19   if (tickLabel >= 0)
20     System.out.print(" " + tickLabel);
21   System.out.print("\n");
22 }
23 /** Draws a line with the given tick length (but no label). */
24 private static void drawLine(int tickLength) {
25   drawLine(tickLength, -1);
26 }
```

Code Fragment 5.2: A recursive implementation of a method that draws a ruler.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

Illustrating Ruler Drawing Using a Recursion Trace

The execution of the recursive drawInterval method can be visualized using a recursion trace. The trace for drawInterval is more complicated than in the factorial example, however, because each instance makes two recursive calls. To illustrate this, we will show the recursion trace in a form that is reminiscent of an outline for a document. See Figure 5.3.

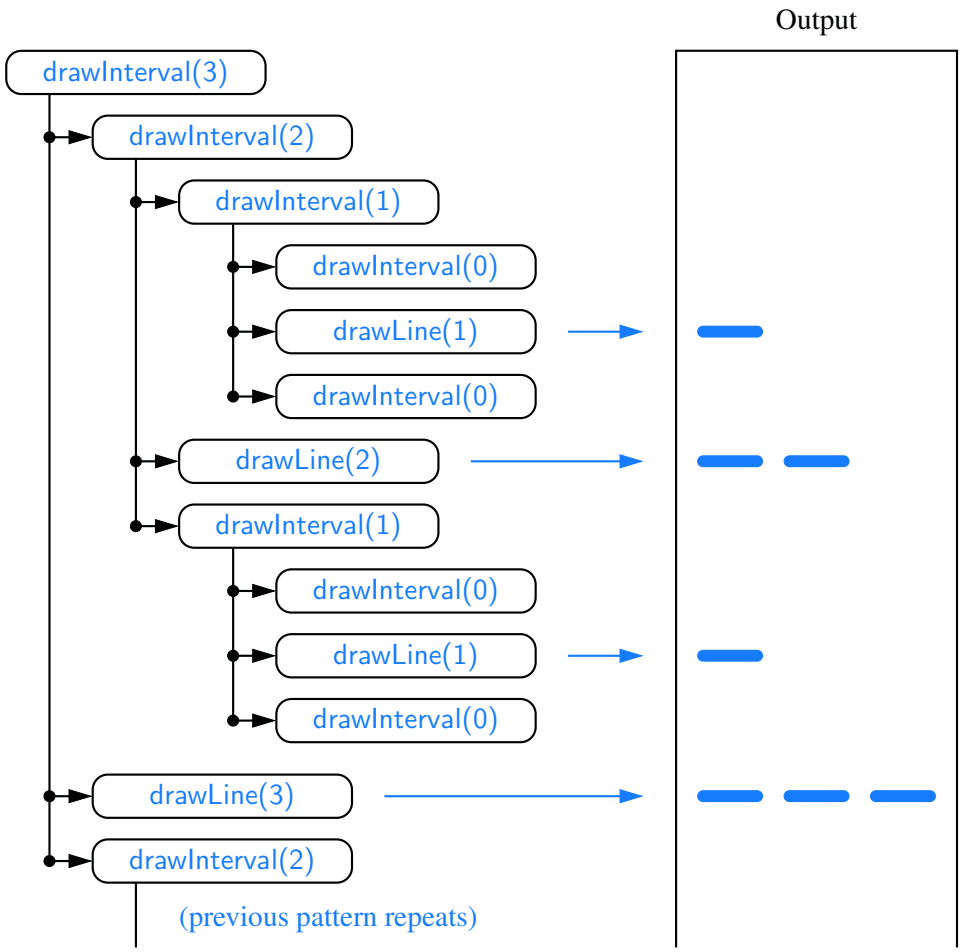


Figure 5.3: A partial recursion trace for the call drawInterval(3). The second pattern of calls for drawInterval(2) is not shown, but it is identical to the first.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.1.3 Binary Search

In this section, we describe a classic recursive algorithm, ***binary search***, used to efficiently locate a target value within a sorted sequence of n elements stored in an array. This is among the most important of computer algorithms, and it is the reason that we so often store data in sorted order (as in Figure 5.4).

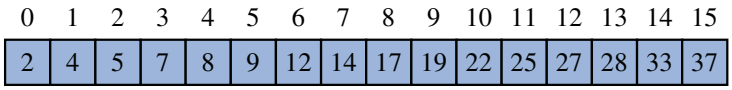


Figure 5.4: Values stored in sorted order within an array. The numbers at top are the indices.

When the sequence is ***unsorted***, the standard approach to search for a target value is to use a loop to examine every element, until either finding the target or exhausting the data set. This algorithm is known as ***linear search***, or ***sequential search***, and it runs in $O(n)$ time (i.e., linear time) since every element is inspected in the worst case.

When the sequence is ***sorted*** and ***indexable***, there is a more efficient algorithm. (For intuition, think about how you would accomplish this task by hand!) If we consider an arbitrary element of the sequence with value v, we can be sure that all elements prior to that in the sequence have values less than or equal to v, and that all elements after that element in the sequence have values greater than or equal to v. This observation allows us to quickly "home in" on a search target using a variant of the children's game "high-low." We call an element of the sequence a ***candidate*** if, at the current stage of the search, we cannot rule out that this item matches the target. The algorithm maintains two parameters, low and high, such that all the candidate elements have index at least low and at most high. Initially, $\mathsf{low} = 0$ and $\mathsf{high} = n-1$. We then compare the target value to the ***median candidate***, that is, the element with index

$$\mathsf{mid} = \lfloor (\mathsf{low} + \mathsf{high})/2 \rfloor .$$

We consider three cases:

- If the target equals the median candidate, then we have found the item we are looking for, and the search terminates successfully.
- If the target is less than the median candidate, then we recur on the first half of the sequence, that is, on the interval of indices from low to $\mathsf{mid} - 1$.
- If the target is greater than the median candidate, then we recur on the second half of the sequence, that is, on the interval of indices from $\mathsf{mid} + 1$ to high.

An unsuccessful search occurs if $\mathsf{low} > \mathsf{high}$, as the interval $[\mathsf{low}, \mathsf{high}]$ is empty.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

This algorithm is known as ***binary search***. We give a Java implementation in Code Fragment 5.3, and an illustration of the execution of the algorithm in Figure 5.5. Whereas sequential search runs in $O(n)$ time, the more efficient binary search runs in $O(\log n)$ time. This is a significant improvement, given that if n is 1 billion, $\log n$ is only 30. (We defer our formal analysis of binary search's running time to Proposition 5.2 in Section 5.2.)

```
1  /**
2   * Returns true if the target value is found in the indicated portion of the data array.
3   * This search only considers the array portion from data[low] to data[high] inclusive.
4   */
5  public static boolean binarySearch(int[ ] data, int target, int low, int high) {
6    if (low > high)
7      return false;                                    // interval empty; no match
8    else {
9      int mid = (low + high) / 2;
10     if (target == data[mid])
11       return true;                                   // found a match
12     else if (target < data[mid])
13       return binarySearch(data, target, low, mid - 1);  // recur left of the middle
14     else
15       return binarySearch(data, target, mid + 1, high);  // recur right of the middle
16   }
17 }
```

Code Fragment 5.3: An implementation of the binary search algorithm on a sorted array.

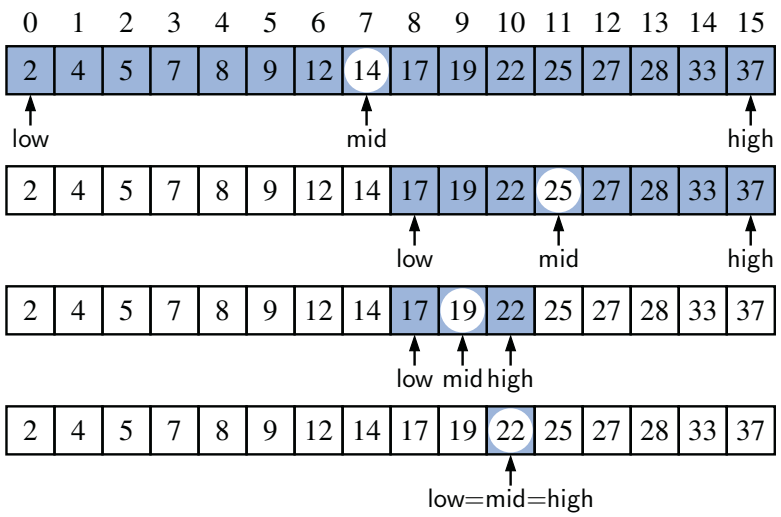


Figure 5.5: Example of a binary search for target value 22 on a sorted array with 16 elements.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.1.4 File Systems

Modern operating systems define file-system directories (also called "folders") in a recursive way. Namely, a file system consists of a top-level directory, and the contents of this directory consists of files and other directories, which in turn can contain files and other directories, and so on. The operating system allows directories to be nested arbitrarily deeply (as long as there is enough memory), although by necessity there must be some base directories that contain only files, not further subdirectories. A representation of a portion of such a file system is given in Figure 5.6.

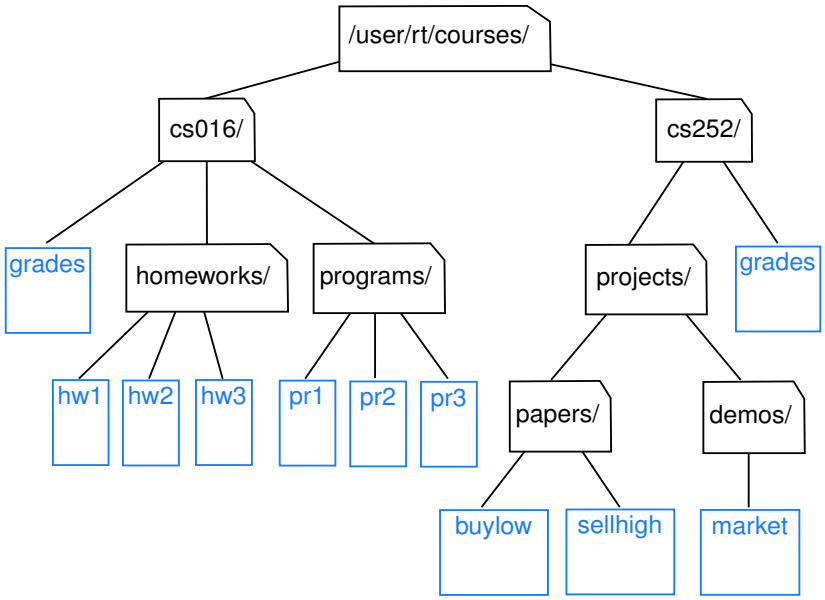


Figure 5.6: A portion of a file system demonstrating a nested organization.

Given the recursive nature of the file-system representation, it should not come as a surprise that many common behaviors of an operating system, such as copying a directory or deleting a directory, are implemented with recursive algorithms. In this section, we consider one such algorithm: computing the total disk usage for all files and directories nested within a particular directory.

For illustration, Figure 5.7 portrays the disk space being used by all entries in our sample file system. We differentiate between the ***immediate*** disk space used by each entry and the ***cumulative*** disk space used by that entry and all nested features. For example, the cs016 directory uses only 2K of immediate space, but a total of 249K of cumulative space.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

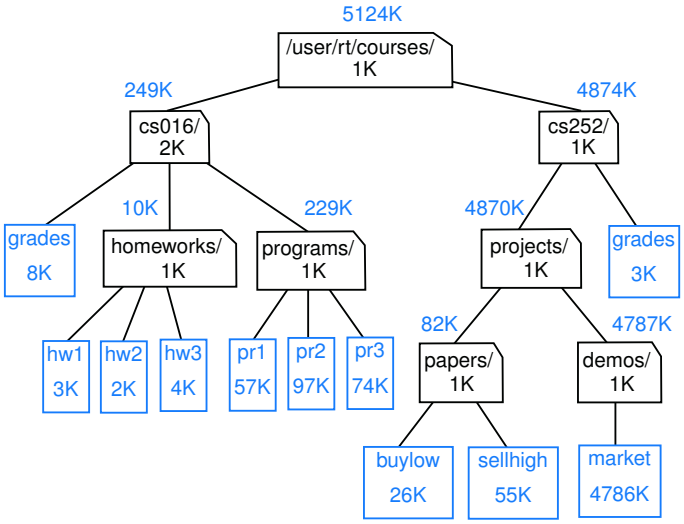


Figure 5.7: The same portion of a file system given in Figure 5.6, but with additional annotations to describe the amount of disk space that is used. Within the icon for each file or directory is the amount of space directly used by that artifact. Above the icon for each directory is an indication of the cumulative disk space used by that directory and all its (recursive) contents.

The cumulative disk space for an entry can be computed with a simple recursive algorithm. It is equal to the immediate disk space used by the entry plus the sum of the cumulative disk space usage of any entries that are stored directly within the entry. For example, the cumulative disk space for cs016 is 249K because it uses 2K itself, 8K cumulatively in grades, 10K cumulatively in homeworks, and 229K cumulatively in programs. Pseudocode for this algorithm is given in Code Fragment 5.4.

Algorithm DiskUsage(*path*):
Input: A string designating a path to a file-system entry
Output: The cumulative disk space used by that entry and any nested entries
total = size(*path*) {immediate disk space used by the entry}
if *path* represents a directory **then**
  for each *child* entry stored within directory *path* **do**
    total = *total* + DiskUsage(*child*) {recursive call}
return *total*

Code Fragment 5.4: An algorithm for computing the cumulative disk space usage nested at a file-system entry. We presume that method size returns the immediate disk space of an entry.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

The java.io.File Class

To implement a recursive algorithm for computing disk usage in Java, we rely on the java.io.File class. An instance of this class represents an abstract pathname in the operating system and allows for properties of that operating system entry to be queried. We will rely on the following methods of the class:

- **new File(pathString)** or **new File(parentFile, childString)**
  A new File instance can be constructed either by providing the full path as a string, or by providing an existing File instance that represents a directory and a string that designates the name of a child entry within that directory.
- **file.length()**
  Returns the immediate disk usage (measured in bytes) for the operating system entry represented by the File instance (e.g., /user/rt/courses).
- **file.isDirectory()**
  Returns true if the File instance represents a directory; false otherwise.
- **file.list()**
  Returns an array of strings designating the names of all entries within the given directory. In our sample file system, if we call this method on the File associated with path /user/rt/courses/cs016, it returns an array with contents: {"grades", "homeworks", "programs"}.

Java Implementation

With use of the File class, we now convert the algorithm from Code Fragment 5.4 into the Java implementation of Code Fragment 5.5.

```
1  /**
2   * Calculates the total disk usage (in bytes) of the portion of the file system rooted
3   * at the given path, while printing a summary akin to the standard 'du' Unix tool.
4   */
5  public static long diskUsage(File root) {
6    long total = root.length();                    // start with direct disk usage
7    if (root.isDirectory()) {                      // and if this is a directory,
8      for (String childname : root.list()) {       // then for each child
9        File child = new File(root, childname);    // compose full path to child
10       total += diskUsage(child);                 // add child's usage to total
11     }
12   }
13   System.out.println(total + "\t" + root);       // descriptive output
14   return total;                                  // return the grand total
15 }
```

Code Fragment 5.5: A recursive method for reporting disk usage of a file system.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

Recursion Trace

To produce a different form of a recursion trace, we have included an extraneous print statement within our Java implementation (line 13 of Code Fragment 5.5). The precise format of that output intentionally mirrors the output that is produced by a classic Unix/Linux utility named du (for "disk usage"). It reports the amount of disk space used by a directory and all contents nested within, and can produce a verbose report, as given in Figure 5.8.

When executed on the sample file system portrayed in Figure 5.7, our implementation of the diskUsage method produces the result given in Figure 5.8. During the execution of the algorithm, exactly one recursive call is made for each entry in the portion of the file system that is considered. Because each line is printed just before returning from a recursive call, the lines of output reflect the order in which the recursive calls are *completed*. Notice that it computes and reports the cumulative disk space for a nested entry before computing and reporting the cumulative disk space for the directory that contains it. For example, the recursive calls regarding entries grades, homeworks, and programs are computed before the cumulative total for the directory /user/rt/courses/cs016 that contains them.

```
8       /user/rt/courses/cs016/grades
3       /user/rt/courses/cs016/homeworks/hw1
2       /user/rt/courses/cs016/homeworks/hw2
4       /user/rt/courses/cs016/homeworks/hw3
10      /user/rt/courses/cs016/homeworks
57      /user/rt/courses/cs016/programs/pr1
97      /user/rt/courses/cs016/programs/pr2
74      /user/rt/courses/cs016/programs/pr3
229     /user/rt/courses/cs016/programs
249     /user/rt/courses/cs016
26      /user/rt/courses/cs252/projects/papers/buylow
55      /user/rt/courses/cs252/projects/papers/sellhigh
82      /user/rt/courses/cs252/projects/papers
4786    /user/rt/courses/cs252/projects/demos/market
4787    /user/rt/courses/cs252/projects/demos
4870    /user/rt/courses/cs252/projects
3       /user/rt/courses/cs252/grades
4874    /user/rt/courses/cs252
5124    /user/rt/courses/
```

Figure 5.8: A report of the disk usage for the file system shown in Figure 5.7, as generated by our diskUsage method from Code Fragment 5.5, or equivalently by the Unix/Linux command du with option -a (which lists both directories and files).

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.2 Analyzing Recursive Algorithms

In Chapter 4, we introduced mathematical techniques for analyzing the efficiency of an algorithm, based upon an estimate of the number of primitive operations that are executed by the algorithm. We use notations such as big-Oh to summarize the relationship between the number of operations and the input size for a problem. In this section, we demonstrate how to perform this type of running-time analysis to recursive algorithms.

With a recursive algorithm, we will account for each operation that is performed based upon the particular ***activation*** of the method that manages the flow of control at the time it is executed. Stated another way, for each invocation of the method, we only account for the number of operations that are performed within the body of that activation. We can then account for the overall number of operations that are executed as part of the recursive algorithm by taking the sum, over all activations, of the number of operations that take place during each individual activation. (As an aside, this is also the way we analyze a nonrecursive method that calls other methods from within its body.)

To demonstrate this style of analysis, we revisit the four recursive algorithms presented in Sections 5.1.1 through 5.1.4: factorial computation, drawing an English ruler, binary search, and computation of the cumulative size of a file system. In general, we may rely on the intuition afforded by a ***recursion trace*** in recognizing how many recursive activations occur, and how the parameterization of each activation can be used to estimate the number of primitive operations that occur within the body of that activation. However, each of these recursive algorithms has a unique structure and form.

Computing Factorials

It is relatively easy to analyze the efficiency of our method for computing factorials, as described in Section 5.1.1. A sample recursion trace for our factorial method was given in Figure 5.1. To compute factorial(n), we see that there are a total of $n+1$ activations, as the parameter decreases from n in the first call, to $n-1$ in the second call, and so on, until reaching the base case with parameter 0.

It is also clear, given an examination of the method body in Code Fragment 5.1, that each individual activation of factorial executes a constant number of operations. Therefore, we conclude that the overall number of operations for computing factorial(n) is $O(n)$, as there are $n+1$ activations, each of which accounts for $O(1)$ operations.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

Drawing an English Ruler

In analyzing the English ruler application from Section 5.1.2, we consider the fundamental question of how many total lines of output are generated by an initial call to drawInterval(c), where c denotes the center length. This is a reasonable benchmark for the overall efficiency of the algorithm as each line of output is based upon a call to the drawLine utility, and each recursive call to drawInterval with nonzero parameter makes exactly one direct call to drawLine.

Some intuition may be gained by examining the source code and the recursion trace. We know that a call to drawInterval(c) for $c > 0$ spawns two calls to drawInterval($c-1$) and a single call to drawLine. We will rely on this intuition to prove the following claim.

Proposition 5.1: *For $c \geq 0$, a call to* drawInterval(c) *results in precisely $2^c - 1$ lines of output.*

Justification: We provide a formal proof of this claim by ***induction*** (see Section 4.4.3). In fact, induction is a natural mathematical technique for proving the correctness and efficiency of a recursive process. In the case of the ruler, we note that an application of drawInterval(0) generates no output, and that $2^0 - 1 = 1 - 1 = 0$. This serves as a base case for our claim.

More generally, the number of lines printed by drawInterval(c) is one more than twice the number generated by a call to drawInterval($c-1$), as one center line is printed between two such recursive calls. By induction, we have that the number of lines is thus $1 + 2 \cdot (2^{c-1} - 1) = 1 + 2^c - 2 = 2^c - 1$. ■

This proof is indicative of a more mathematically rigorous tool, known as a ***recurrence equation***, that can be used to analyze the running time of a recursive algorithm. That technique is discussed in Section 12.1.4, in the context of recursive sorting algorithms.

Performing a Binary Search

When considering the running time of the binary search algorithm, as presented in Section 5.1.3, we observe that a constant number of primitive operations are executed during each recursive call of the binary search method. Hence, the running time is proportional to the number of recursive calls performed. We will show that at most $\lfloor \log n \rfloor + 1$ recursive calls are made during a binary search of a sequence having n elements, leading to the following claim.

Proposition 5.2: *The binary search algorithm runs in $O(\log n)$ time for a sorted array with n elements.*

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

Justification: To prove this claim, a crucial fact is that with each recursive call the number of candidate elements still to be searched is given by the value

$$\mathsf{high} - \mathsf{low} + 1.$$

Moreover, the number of remaining candidates is reduced by at least one-half with each recursive call. Specifically, from the definition of mid, the number of remaining candidates is either

$$(\mathsf{mid} - 1) - \mathsf{low} + 1 = \left\lfloor \frac{\mathsf{low} + \mathsf{high}}{2} \right\rfloor - \mathsf{low} \leq \frac{\mathsf{high} - \mathsf{low} + 1}{2}$$

or

$$\mathsf{high} - (\mathsf{mid} + 1) + 1 = \mathsf{high} - \left\lfloor \frac{\mathsf{low} + \mathsf{high}}{2} \right\rfloor \leq \frac{\mathsf{high} - \mathsf{low} + 1}{2}.$$

Initially, the number of candidates is n; after the first call in a binary search, it is at most $n/2$; after the second call, it is at most $n/4$; and so on. In general, after the j^{th} call in a binary search, the number of candidate elements remaining is at most $n/2^j$. In the worst case (an unsuccessful search), the recursive calls stop when there are no more candidate elements. Hence, the maximum number of recursive calls performed, is the smallest integer r such that

$$\frac{n}{2^r} < 1.$$

In other words (recalling that we omit a logarithm's base when it is 2), r is the smallest integer such that $r > \log n$. Thus, we have

$$r = \lfloor \log n \rfloor + 1,$$

which implies that binary search runs in $O(\log n)$ time. ■

Computing Disk Space Usage

Our final recursive algorithm from Section 5.1 was that for computing the overall disk space usage in a specified portion of a file system. To characterize the "problem size" for our analysis, we let n denote the number of file-system entries in the portion of the file system that is considered. (For example, the file system portrayed in Figure 5.6 has $n = 19$ entries.)

To characterize the cumulative time spent for an initial call to diskUsage, we must analyze the total number of recursive invocations that are made, as well as the number of operations that are executed within those invocations.

We begin by showing that there are precisely n recursive invocations of the method, in particular, one for each entry in the relevant portion of the file system. Intuitively, this is because a call to diskUsage for a particular entry e of the file system is only made from within the for loop of Code Fragment 5.5 when processing the entry for the unique directory that contains e, and that entry will only be explored once.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

To formalize this argument, we can define the ***nesting level*** of each entry such that the entry on which we begin has nesting level 0, entries stored directly within it have nesting level 1, entries stored within those entries have nesting level 2, and so on. We can prove by induction that there is exactly one recursive invocation of diskUsage upon each entry at nesting level k. As a base case, when $k = 0$, the only recursive invocation made is the initial one. As the inductive step, once we know there is exactly one recursive invocation for each entry at nesting level k, we can claim that there is exactly one invocation for each entry e at nesting level $k+1$, made within the for loop for the entry at level k that contains e.

Having established that there is one recursive call for each entry of the file system, we return to the question of the overall computation time for the algorithm. It would be great if we could argue that we spend $O(1)$ time in any single invocation of the method, but that is not the case. While there is a constant number of steps reflected in the call to root.length( ) to compute the disk usage directly at that entry, when the entry is a directory, the body of the diskUsage method includes a for loop that iterates over all entries that are contained within that directory. In the worst case, it is possible that one entry includes $n-1$ others.

Based on this reasoning, we could conclude that there are $O(n)$ recursive calls, each of which runs in $O(n)$ time, leading to an overall running time that is $O(n^2)$. While this upper bound is technically true, it is not a tight upper bound. Remarkably, we can prove the stronger bound that the recursive algorithm for diskUsage completes in $O(n)$ time! The weaker bound was pessimistic because it assumed a worst-case number of entries for each directory. While it is possible that some directories contain a number of entries proportional to n, they cannot all contain that many. To prove the stronger claim, we choose to consider the *overall* number of iterations of the for loop across all recursive calls. We claim there are precisely $n-1$ such iterations of that loop overall. We base this claim on the fact that each iteration of that loop makes a recursive call to diskUsage, and yet we have already concluded that there are a total of n calls to diskUsage (including the original call). We therefore conclude that there are $O(n)$ recursive calls, each of which uses $O(1)$ time outside the loop, and that the *overall* number of operations due to the loop is $O(n)$. Summing all of these bounds, the overall number of operations is $O(n)$.

The argument we have made is more advanced than with the earlier examples of recursion. The idea that we can sometimes get a tighter bound on a series of operations by considering the cumulative effect, rather than assuming that each achieves a worst case is a technique called ***amortization***; we will see another example of such analysis in Section 7.2.3. Furthermore, a file system is an implicit example of a data structure known as a ***tree***, and our disk usage algorithm is really a manifestation of a more general algorithm known as a ***tree traversal***. Trees will be the focus of Chapter 8, and our argument about the $O(n)$ running time of the disk usage algorithm will be generalized for tree traversals in Section 8.4.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.3 Further Examples of Recursion

In this section, we provide additional examples of the use of recursion. We organize our presentation by considering the maximum number of recursive calls that may be started from within the body of a single activation.

- If a recursive call starts at most one other, we call this a ***linear recursion***.
- If a recursive call may start two others, we call this a ***binary recursion***.
- If a recursive call may start three or more others, this is ***multiple recursion***.

5.3.1 Linear Recursion

If a recursive method is designed so that each invocation of the body makes at most one new recursive call, this is know as ***linear recursion***. Of the recursions we have seen so far, the implementation of the factorial method (Section 5.1.1) is a clear example of linear recursion. More interestingly, the binary search algorithm (Section 5.1.3) is also an example of *linear recursion*, despite the term "binary" in the name. The code for binary search (Code Fragment 5.3) includes a case analysis, with two branches that lead to a further recursive call, but only one branch is followed during a particular execution of the body.

A consequence of the definition of linear recursion is that any recursion trace will appear as a single sequence of calls, as we originally portrayed for the factorial method in Figure 5.1 of Section 5.1.1. Note that the *linear recursion* terminology reflects the structure of the recursion trace, not the asymptotic analysis of the running time; for example, we have seen that binary search runs in $O(\log n)$ time.

Summing the Elements of an Array Recursively

Linear recursion can be a useful tool for processing a sequence, such as a Java array. Suppose, for example, that we want to compute the sum of an array of n integers. We can solve this summation problem using linear recursion by observing that if $n = 0$ the sum is trivially 0, and otherwise it is the sum of the first $n-1$ integers in the array plus the last value in the array. (See Figure 5.9.)

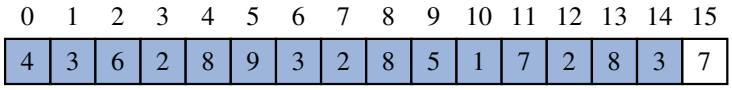


Figure 5.9: Computing the sum of a sequence recursively, by adding the last number to the sum of the first $n-1$.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

A recursive algorithm for computing the sum of an array of integers based on this intuition is implemented in Code Fragment 5.6.

```
1  /** Returns the sum of the first n integers of the given array. */
2  public static int linearSum(int[ ] data, int n) {
3    if (n == 0)
4      return 0;
5    else
6      return linearSum(data, n−1) + data[n−1];
7  }
```

Code Fragment 5.6: Summing an array of integers using linear recursion.

A recursion trace of the linearSum method for a small example is given in Figure 5.10. For an input of size n, the linearSum algorithm makes $n+1$ method calls. Hence, it will take $O(n)$ time, because it spends a constant amount of time performing the nonrecursive part of each call. Moreover, we can also see that the memory space used by the algorithm (in addition to the array) is also $O(n)$, as we use a constant amount of memory space for each of the $n+1$ frames in the trace at the time we make the final recursive call (with $n = 0$).

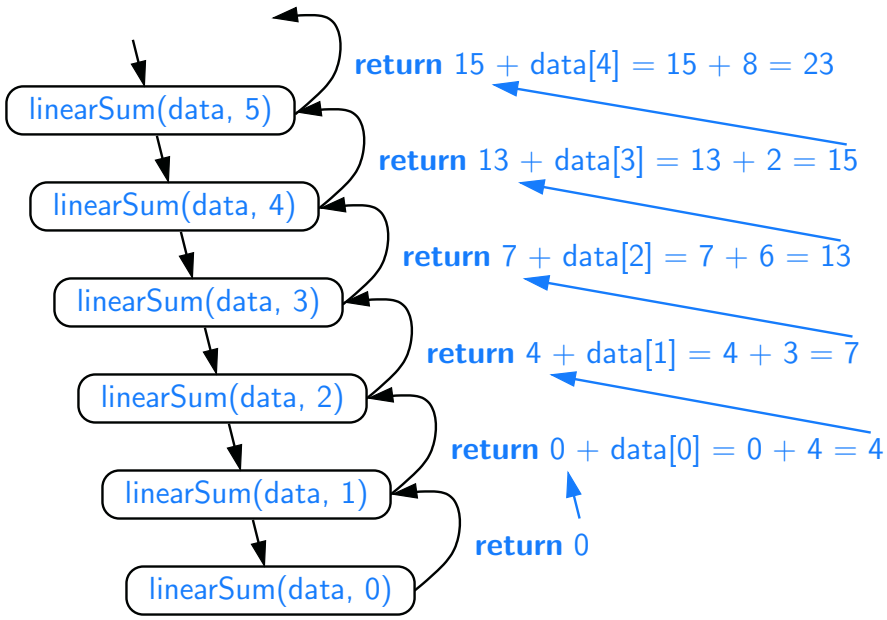


Figure 5.10: Recursion trace for an execution of linearSum(data, 5) with input parameter data = 4, 3, 6, 2, 8.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

Reversing a Sequence with Recursion

Next, let us consider the problem of reversing the n elements of an array, so that the first element becomes the last, the second element becomes second to the last, and so on. We can solve this problem using linear recursion, by observing that the reversal of a sequence can be achieved by swapping the first and last elements and then recursively reversing the remaining elements. We present an implementation of this algorithm in Code Fragment 5.7, using the convention that the first time we call this algorithm we do so as reverseArray(data, 0, n−1).

```
1  /** Reverses the contents of subarray data[low] through data[high] inclusive. */
2  public static void reverseArray(int[ ] data, int low, int high) {
3    if (low < high) {                          // if at least two elements in subarray
4      int temp = data[low];                    // swap data[low] and data[high]
5      data[low] = data[high];
6      data[high] = temp;
7      reverseArray(data, low + 1, high − 1);   // recur on the rest
8    }
9  }
```

Code Fragment 5.7: Reversing the elements of an array using linear recursion.

We note that whenever a recursive call is made, there will be two fewer elements in the relevant portion of the array. (See Figure 5.11.) Eventually a base case is reached when the condition low < high fails, either because low == high in the case that n is odd, or because low == high + 1 in the case that n is even.

The above argument implies that the recursive algorithm of Code Fragment 5.7 is guaranteed to terminate after a total of $1 + \lfloor \frac{n}{2} \rfloor$ recursive calls. Because each call involves a constant amount of work, the entire process runs in $O(n)$ time.

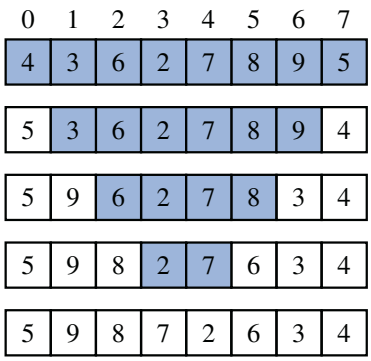


Figure 5.11: A trace of the recursion for reversing a sequence. The highlighted portion has yet to be reversed.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

Recursive Algorithms for Computing Powers

As another interesting example of the use of linear recursion, we consider the problem of raising a number x to an arbitrary nonnegative integer n. That is, we wish to compute the ***power function***, defined as $power(x,n) = x^n$. (We use the name "power" for this discussion, to differentiate from the pow method of the Math class, which provides such functionality.) We will consider two different recursive formulations for the problem that lead to algorithms with very different performance.

A trivial recursive definition follows from the fact that $x^n = x \cdot x^{n-1}$ for $n > 0$.

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot power(x, n-1) & \text{otherwise.} \end{cases}$$

This definition leads to a recursive algorithm shown in Code Fragment 5.8.

```
1  /** Computes the value of x raised to the nth power, for nonnegative integer n. */
2  public static double power(double x, int n) {
3    if (n == 0)
4      return 1;
5    else
6      return x * power(x, n-1);
7  }
```

Code Fragment 5.8: Computing the power function using trivial recursion.

A recursive call to this version of $power(x,n)$ runs in $O(n)$ time. Its recursion trace has structure very similar to that of the factorial function from Figure 5.1, with the parameter decreasing by one with each call, and constant work performed at each of $n+1$ levels.

However, there is a much faster way to compute the power function using an alternative definition that employs a squaring technique. Let $k = \lfloor \frac{n}{2} \rfloor$ denote the floor of the integer division (equivalent to n/2 in Java when n is an int). We consider the expression $(x^k)^2$. When n is even, $\lfloor \frac{n}{2} \rfloor = \frac{n}{2}$ and therefore $(x^k)^2 = \left(x^{\frac{n}{2}}\right)^2 = x^n$. When n is odd, $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$ and $(x^k)^2 = x^{n-1}$, and therefore $x^n = (x^k)^2 \cdot x$, just as $2^{13} = (2^6 \cdot 2^6) \cdot 2$. This analysis leads to the following recursive definition:

$$power(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 \cdot x & \text{if } n > 0 \text{ is odd} \\ \left(power\left(x, \lfloor \frac{n}{2} \rfloor\right)\right)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

If we were to implement this recursion making *two* recursive calls to compute $power(x, \lfloor \frac{n}{2} \rfloor) \cdot power(x, \lfloor \frac{n}{2} \rfloor)$, a trace of the recursion would demonstrate $O(n)$ calls. We can perform significantly fewer operations by computing $power(x, \lfloor \frac{n}{2} \rfloor)$ and storing it in a variable as a partial result, and then multiplying it by itself. An implementation based on this recursive definition is given in Code Fragment 5.9.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

```
1  /** Computes the value of x raised to the nth power, for nonnegative integer n. */
2  public static double power(double x, int n) {
3    if (n == 0)
4      return 1;
5    else {
6      double partial = power(x, n/2);          // rely on truncated division of n
7      double result = partial * partial;
8      if (n % 2 == 1)                          // if n odd, include extra factor of x
9        result *= x;
10     return result;
11   }
12 }
```

Code Fragment 5.9: Computing the power function using repeated squaring.

To illustrate the execution of our improved algorithm, Figure 5.12 provides a recursion trace of the computation power(2, 13).

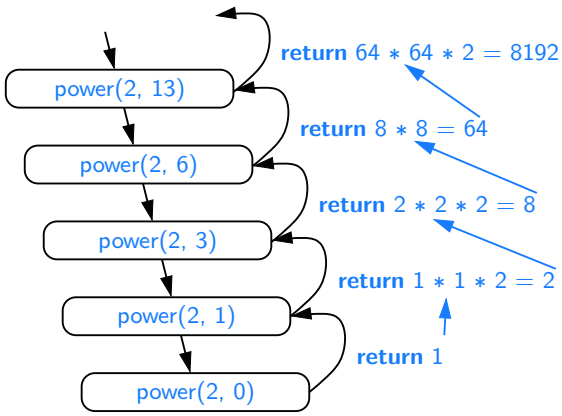


Figure 5.12: Recursion trace for an execution of power(2, 13).

To analyze the running time of the revised algorithm, we observe that the exponent in each recursive call of method power(x,n) is at most half of the preceding exponent. As we saw with the analysis of binary search, the number of times that we can divide n by two before getting to one or less is $O(\log n)$. Therefore, our new formulation of power results in $O(\log n)$ recursive calls. Each individual activation of the method uses $O(1)$ operations (excluding the recursive call), and so the total number of operations for computing power(x,n) is $O(\log n)$. This is a significant improvement over the original $O(n)$-time algorithm.

The improved version also provides significant saving in reducing the memory usage. The first version has a recursive depth of $O(n)$, and therefore, $O(n)$ frames are simultaneously stored in memory. Because the recursive depth of the improved version is $O(\log n)$, its memory usage is $O(\log n)$ as well.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.3.2 Binary Recursion

When a method makes two recursive calls, we say that it uses ***binary recursion***. We have already seen an example of binary recursion when drawing the English ruler (Section 5.1.2). As another application of binary recursion, let us revisit the problem of summing the n integers of an array. Computing the sum of one or zero values is trivial. With two or more values, we can recursively compute the sum of the first half, and the sum of the second half, and add those sums together. Our implementation of such an algorithm, in Code Fragment 5.10, is initially invoked as binarySum(data, 0, n−1).

```
1  /** Returns the sum of subarray data[low] through data[high] inclusive. */
2  public static int binarySum(int[ ] data, int low, int high) {
3    if (low > high)                                    // zero elements in subarray
4      return 0;
5    else if (low == high)                              // one element in subarray
6      return data[low];
7    else {
8      int mid = (low + high) / 2;
9      return binarySum(data, low, mid) + binarySum(data, mid+1, high);
10   }
11 }
```

Code Fragment 5.10: Summing the elements of a sequence using binary recursion.

To analyze algorithm binarySum, we consider, for simplicity, the case where n is a power of two. Figure 5.13 shows the recursion trace of an execution of binarySum(data, 0, 7). We label each box with the values of parameters low and high for that call. The size of the range is divided in half at each recursive call, and so the depth of the recursion is $1+\log_2 n$. Therefore, binarySum uses $O(\log n)$ amount of additional space, which is a big improvement over the $O(n)$ space used by the linearSum method of Code Fragment 5.6. However, the running time of binarySum is $O(n)$, as there are $2n-1$ method calls, each requiring constant time.

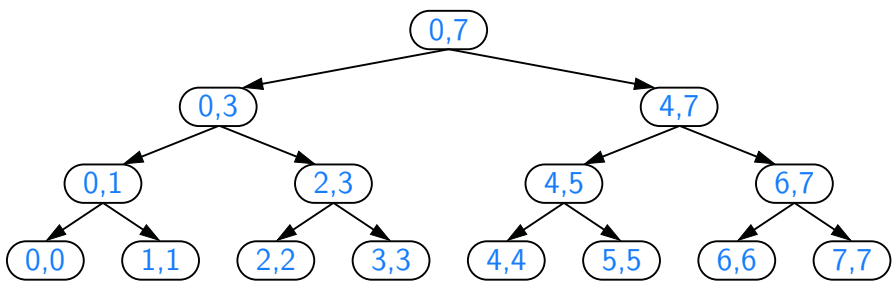


Figure 5.13: Recursion trace for the execution of binarySum(data, 0, 7).

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.3.3 Multiple Recursion

Generalizing from binary recursion, we define ***multiple recursion*** as a process in which a method may make more than two recursive calls. Our recursion for analyzing the disk space usage of a file system (see Section 5.1.4) is an example of multiple recursion, because the number of recursive calls made during one invocation was equal to the number of entries within a given directory of the file system.

Another common application of multiple recursion is when we want to enumerate various configurations in order to solve a combinatorial puzzle. For example, the following are all instances of what are known as ***summation puzzles***:

$$\begin{aligned} pot + pan &= bib \\ dog + cat &= pig \\ boy + girl &= baby \end{aligned}$$

To solve such a puzzle, we need to assign a unique digit (that is, $0, 1, \ldots, 9$) to each letter in the equation, in order to make the equation true. Typically, we solve such a puzzle by using our human observations of the particular puzzle we are trying to solve to eliminate configurations (that is, possible partial assignments of digits to letters) until we can work through the feasible configurations that remain, testing for the correctness of each one.

If the number of possible configurations is not too large, however, we can use a computer to simply enumerate all the possibilities and test each one, without employing any human observations. Such an algorithm can use multiple recursion to work through the configurations in a systematic way. To keep the description general enough to be used with other puzzles, we consider an algorithm that enumerates and tests all k-length sequences, without repetitions, chosen from a given universe U. We show pseudocode for such an algorithm in Code Fragment 5.11, building the sequence of k elements with the following steps:

1. Recursively generating the sequences of $k-1$ elements
2. Appending to each such sequence an element not already contained in it.

Throughout the execution of the algorithm, we use a set U to keep track of the elements not contained in the current sequence, so that an element e has not been used yet if and only if e is in U.

Another way to look at the algorithm of Code Fragment 5.11 is that it enumerates every possible size-k ordered subset of U, and tests each subset for being a possible solution to our puzzle.

For summation puzzles, $U = \{0,1,2,3,4,5,6,7,8,9\}$ and each position in the sequence corresponds to a given letter. For example, the first position could stand for b, the second for o, the third for y, and so on.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

Algorithm PuzzleSolve(k, S, U):
Input: An integer k, sequence S, and set U
Output: An enumeration of all k-length extensions to S using elements in U without repetitions

```
for each e in U do
    Add e to the end of S
    Remove e from U                                  {e is now being used}
    if k == 1 then
        Test whether S is a configuration that solves the puzzle
        if S solves the puzzle then
            add S to output                          {a solution}
    else
        PuzzleSolve(k − 1, S, U)                     {a recursive call}
    Remove e from the end of S
    Add e back to U                                  {e is now considered as unused}
```

Code Fragment 5.11: Solving a combinatorial puzzle by enumerating and testing all possible configurations.

In Figure 5.14, we show a recursion trace of a call to PuzzleSolve(3, S, U), where S is empty and $U = \{a,b,c\}$. During the execution, all the permutations of the three characters are generated and tested. Note that the initial call makes three recursive calls, each of which in turn makes two more. If we had executed PuzzleSolve(3, S, U) on a set U consisting of four elements, the initial call would have made four recursive calls, each of which would have a trace looking like the one in Figure 5.14.

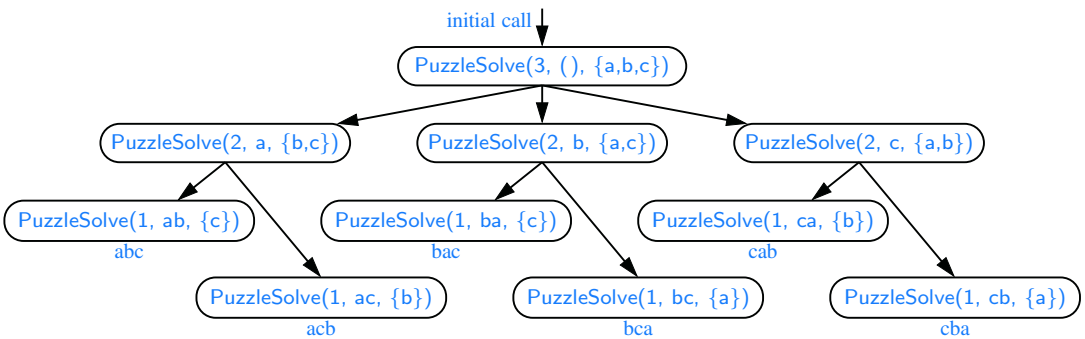


Figure 5.14: Recursion trace for an execution of PuzzleSolve(3, S, U), where S is empty and $U = \{a,b,c\}$. This execution generates and tests all permutations of a, b, and c. We show the permutations generated directly below their respective boxes.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.4 Designing Recursive Algorithms

An algorithm that uses recursion typically has the following form:

- ***Test for base cases.*** We begin by testing for a set of base cases (there should be at least one). These base cases should be defined so that every possible chain of recursive calls will eventually reach a base case, and the handling of each base case should not use recursion.
- ***Recur.*** If not a base case, we perform one or more recursive calls. This recursive step may involve a test that decides which of several possible recursive calls to make. We should define each possible recursive call so that it makes progress towards a base case.

Parameterizing a Recursion

To design a recursive algorithm for a given problem, it is useful to think of the different ways we might define subproblems that have the same general structure as the original problem. If one has difficulty finding the repetitive structure needed to design a recursive algorithm, it is sometimes useful to work out the problem on a few concrete examples to see how the subproblems should be defined.

A successful recursive design sometimes requires that we redefine the original problem to facilitate similar-looking subproblems. Often, this involved reparameterizing the signature of the method. For example, when performing a binary search in an array, a natural method signature for a caller would appear as binarySearch(data, target). However, in Section 5.1.3, we defined our method with calling signature binarySearch(data, target, low, high), using the additional parameters to demarcate subarrays as the recursion proceeds. This change in parameterization is critical for binary search. Several other examples in this chapter (e.g., reverseArray, linearSum, binarySum) also demonstrated the use of additional parameters in defining recursive subproblems.

If we wish to provide a cleaner public interface to an algorithm without exposing the user to the recursive parameterization, a standard technique is to make the recursive version private, and to introduce a cleaner public method (that calls the private one with appropriate parameters). For example, we might offer the following simpler version of binarySearch for public use:

```
/** Returns true if the target value is found in the data array. */
public static boolean binarySearch(int[ ] data, int target) {
  return binarySearch(data, target, 0, data.length − 1); // use parameterized version
}
```

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.5 Recursion Run Amok

Although recursion is a very powerful tool, it can easily be misused in various ways. In this section, we examine several cases in which a poorly implemented recursion causes drastic inefficiency, and we discuss some strategies for recognizing and avoid such pitfalls.

We begin by revisiting the ***element uniqueness problem***, defined on page 174 of Section 4.3.3. We can use the following recursive formulation to determine if all n elements of a sequence are unique. As a base case, when $n = 1$, the elements are trivially unique. For $n \geq 2$, the elements are unique if and only if the first $n-1$ elements are unique, the last $n-1$ items are unique, and the first and last elements are different (as that is the only pair that was not already checked as a subcase). A recursive implementation based on this idea is given in Code Fragment 5.12, named unique3 (to differentiate it from unique1 and unique2 from Chapter 4).

```
1  /** Returns true if there are no duplicate values from data[low] through data[high].*/
2  public static boolean unique3(int[ ] data, int low, int high) {
3    if (low >= high) return true;                          // at most one item
4    else if (!unique3(data, low, high−1)) return false;    // duplicate in first n−1
5    else if (!unique3(data, low+1, high)) return false;    // duplicate in last n−1
6    else return (data[low] != data[high]);                 // do first and last differ?
7  }
```

Code Fragment 5.12: Recursive unique3 for testing element uniqueness.

Unfortunately, this is a terribly inefficient use of recursion. The nonrecursive part of each call uses $O(1)$ time, so the overall running time will be proportional to the total number of recursive invocations. To analyze the problem, we let n denote the number of entries under consideration, that is, let $n = 1 + \text{high} - \text{low}$.

If $n = 1$, then the running time of unique3 is $O(1)$, since there are no recursive calls for this case. In the general case, the important observation is that a single call to unique3 for a problem of size n may result in two recursive calls on problems of size $n-1$. Those two calls with size $n-1$ could in turn result in four calls (two each) with a range of size $n-2$, and thus eight calls with size $n-3$ and so on. Thus, in the worst case, the total number of method calls is given by the geometric summation

$$1 + 2 + 4 + \cdots + 2^{n-1},$$

which is equal to $2^n - 1$ by Proposition 4.5. Thus, the running time of method unique3 is $O(2^n)$. This is an incredibly inefficient method for solving the element uniqueness problem. Its inefficiency comes not from the fact that it uses recursion—it comes from the fact that it uses recursion poorly, which is something we address in Exercise C-5.12.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

An Inefficient Recursion for Computing Fibonacci Numbers

In Section 2.2.3, we introduced a process for generating the progression of Fibonacci numbers, which can be defined recursively as follows:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-2} + F_{n-1} \quad \text{for } n > 1. \end{aligned}$$

Ironically, a recursive implementation based directly on this definition results in the method fibonacciBad shown in Code Fragment 5.13, which computes a Fibonacci number by making two recursive calls in each non-base case.

```
1  /** Returns the nth Fibonacci number (inefficiently). */
2  public static long fibonacciBad(int n) {
3    if (n <= 1)
4      return n;
5    else
6      return fibonacciBad(n-2) + fibonacciBad(n-1);
7  }
```

Code Fragment 5.13: Computing the n^{th} Fibonacci number using binary recursion.

Unfortunately, such a direct implementation of the Fibonacci formula results in a terribly inefficient method. Computing the n^{th} Fibonacci number in this way requires an exponential number of calls to the method. Specifically, let c_n denote the number of calls performed in the execution of fibonacciBad(n). Then, we have the following values for the c_n's:

$$\begin{aligned} c_0 &= 1 \\ c_1 &= 1 \\ c_2 &= 1 + c_0 + c_1 = 1 + 1 + 1 = 3 \\ c_3 &= 1 + c_1 + c_2 = 1 + 1 + 3 = 5 \\ c_4 &= 1 + c_2 + c_3 = 1 + 3 + 5 = 9 \\ c_5 &= 1 + c_3 + c_4 = 1 + 5 + 9 = 15 \\ c_6 &= 1 + c_4 + c_5 = 1 + 9 + 15 = 25 \\ c_7 &= 1 + c_5 + c_6 = 1 + 15 + 25 = 41 \\ c_8 &= 1 + c_6 + c_7 = 1 + 25 + 41 = 67 \end{aligned}$$

If we follow the pattern forward, we see that the number of calls more than doubles for each two consecutive indices. That is, c_4 is more than twice c_2, c_5 is more than twice c_3, c_6 is more than twice c_4, and so on. Thus, $c_n > 2^{n/2}$, which means that fibonacciBad(n) makes a number of calls that is exponential in n.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

An Efficient Recursion for Computing Fibonacci Numbers

We were tempted into using the bad recursive formulation because of the way the n^{th} Fibonacci number, F_n, depends on the two previous values, F_{n-2} and F_{n-1}. But notice that after computing F_{n-2}, the call to compute F_{n-1} requires its own recursive call to compute F_{n-2}, as it does not have knowledge of the value of F_{n-2} that was computed at the earlier level of recursion. That is duplicative work. Worse yet, both of those calls will need to (re)compute the value of F_{n-3}, as will the computation of F_{n-1}. This snowballing effect is what leads to the exponential running time of fibonacciBad.

We can compute F_n much more efficiently using a recursion in which each invocation makes only one recursive call. To do so, we need to redefine the expectations of the method. Rather than having the method return a single value, which is the n^{th} Fibonacci number, we define a recursive method that returns an array with two consecutive Fibonacci numbers $\{F_n, F_{n-1}\}$, using the convention $F_{-1} = 0$. Although it seems to be a greater burden to report two consecutive Fibonacci numbers instead of one, passing this extra information from one level of the recursion to the next makes it much easier to continue the process. (It allows us to avoid having to recompute the second value that was already known within the recursion.) An implementation based on this strategy is given in Code Fragment 5.14.

```
1  /** Returns array containing the pair of Fibonacci numbers, F(n) and F(n−1). */
2  public static long[ ] fibonacciGood(int n) {
3    if (n <= 1) {
4      long[ ] answer = {n, 0};
5      return answer;
6    } else {
7      long[ ] temp = fibonacciGood(n − 1);          // returns {F_{n−1}, F_{n−2}}
8      long[ ] answer = {temp[0] + temp[1], temp[0]}; // we want {F_n, F_{n−1}}
9      return answer;
10   }
11 }
```

Code Fragment 5.14: Computing the n^{th} Fibonacci number using linear recursion.

In terms of efficiency, the difference between the bad and good recursions for this problem is like night and day. The fibonacciBad method uses exponential time. We claim that the execution of method fibonacciGood(n) runs in $O(n)$ time. Each recursive call to fibonacciGood decreases the argument n by 1; therefore, a recursion trace includes a series of n method calls. Because the nonrecursive work for each call uses constant time, the overall computation executes in $O(n)$ time.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.5.1 Maximum Recursive Depth in Java

Another danger in the misuse of recursion is known as ***infinite recursion***. If each recursive call makes another recursive call, without ever reaching a base case, then we have an infinite series of such calls. This is a fatal error. An infinite recursion can quickly swamp computing resources, not only due to rapid use of the CPU, but because each successive call creates a frame requiring additional memory. A blatant example of an ill-formed recursion is the following:

```
1  /** Don't call this (infinite) version. */
2  public static int fibonacci(int n) {
3    return fibonacci(n);                    // After all F_n does equal F_n
4  }
```

However, there are far more subtle errors that can lead to an infinite recursion. Revisiting our implementation of binary search (Code Fragment 5.3), when we make a recursive call on the right portion of the sequence (line 15), we specify the subarray from index mid+1 to high. Had that line instead been written as

```
return binarySearch(data, target, mid, high);   // sending mid, not mid+1
```

this could result in an infinite recursion. In particular, when searching a range of two elements, it becomes possible to make a recursive call on the identical range.

A programmer should ensure that each recursive call is in some way progressing toward a base case (for example, by having a parameter value that decreases with each call). To combat against infinite recursions, the designers of Java made an intentional decision to limit the overall space used to store activation frames for simultaneously active method calls. If this limit is reached, the Java Virtual Machine throws a StackOverflowError. (We will further discuss the "stack" data structure in Section 6.1.) The precise value of this limit depends upon the Java installation, but a typical value might allow upward of 1000 simultaneous calls.

For many applications of recursion, allowing up to 1000 nested calls suffices. For example, our binarySearch method (Section 5.1.3) has $O(\log n)$ recursive depth, and so for the default recursive limit to be reached, there would need to be 2^{1000} elements (far, far more than the estimated number of atoms in the universe). However, we have seen several linear recursions that have recursive depth proportional to n. Java's limit on the recursive depth might disrupt such computations.

It is possible to reconfigure the Java Virtual Machine so that it allows for greater space to be devoted to nested method calls. This is done by setting the -Xss runtime option when starting Java, either as a command-line option or through the settings of an IDE. But it often possible to rely upon the intuition of a recursive algorithm, yet to reimplement it more directly using traditional loops rather than method calls to express the necessary repetition. We discuss just such an approach to conclude the chapter.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.6 Eliminating Tail Recursion

The main benefit of a recursive approach to algorithm design is that it allows us to succinctly take advantage of a repetitive structure present in many problems. By making our algorithm description exploit the repetitive structure in a recursive way, we can often avoid complex case analyses and nested loops. This approach can lead to more readable algorithm descriptions, while still being quite efficient.

However, the usefulness of recursion comes at a modest cost. In particular, the Java Virtual Machine must maintain frames that keep track of the state of each nested call. When computer memory is at a premium, it can be beneficial to derive nonrecursive implementations of recursive algorithms.

In general, we can use the stack data structure, which we will introduce in Section 6.1, to convert a recursive algorithm into a nonrecursive algorithm by managing the nesting of the recursive structure ourselves, rather than relying on the interpreter to do so. Although this only shifts the memory usage from the interpreter to our stack, we may be able to further reduce the memory usage by storing the minimal information necessary.

Even better, some forms of recursion can be eliminated without any use of auxiliary memory. One such form is known as ***tail recursion***. A recursion is a tail recursion if any recursive call that is made from one context is the very last operation in that context, with the return value of the recursive call (if any) immediately returned by the enclosing recursion. By necessity, a tail recursion must be a linear recursion (since there is no way to make a second recursive call if you must immediately return the result of the first).

Of the recursive methods demonstrated in this chapter, the binarySearch method of Code Fragment 5.3 and the reverseArray method of Code Fragment 5.7 are examples of tail recursion. Several others of our linear recursions are almost like tail recursion, but not technically so. For example, our factorial method of Code Fragment 5.1 is *not* a tail recursion. It concludes with the command:

```
return n * factorial(n-1);
```

This is not a tail recursion because an additional multiplication is performed after the recursive call is completed, and the result returned is not the same. For similar reasons, the linearSum method of Code Fragment 5.6, both power methods from Code Fragments 5.8 and 5.9, and the fibonacciGood method of Code Fragment 5.13 fail to be tail recursions.

Tail recursions are special, as they can be automatically reimplemented nonrecursively by enclosing the body in a loop for repetition, and replacing a recursive call with new parameters by a reassignment of the existing parameters to those values. In fact, many programming language implementations may convert tail recursions in this way as an optimization.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

```
1  /** Returns true if the target value is found in the data array. */
2  public static boolean binarySearchIterative(int[ ] data, int target) {
3    int low = 0;
4    int high = data.length − 1;
5    while (low <= high) {
6      int mid = (low + high) / 2;
7      if (target == data[mid])                    // found a match
8        return true;
9      else if (target < data[mid])
10       high = mid − 1;                           // only consider values left of mid
11     else
12       low = mid + 1;                            // only consider values right of mid
13   }
14   return false;                                 // loop ended without success
15 }
```

Code Fragment 5.15: A nonrecursive implementation of binary search.

As a tangible example, our binarySearch method can be reimplemented as shown in Code Fragment 5.15. We initialize variables low and high to represent the full extent of the array just prior to our while loop. Then, during each pass of the loop, we either find the target, or we narrow the range of the candidate subarray. Where we made the recursive call binarySearch(data, target, low, mid -1) in the original version, we simply replace high $=$ mid $-$ 1 in our new version and then continue to the next iteration of the loop. Our original base case condition of low $>$ high has simply been replaced by the opposite loop condition, **while** low $<=$ high. In our new implementation, we return false to designate a failed search if the while loop ends without having ever returned true from within.

Most other linear recursions can be expressed quite efficiently with iteration, even if they were not formally tail recursions. For example, there are trivial nonrecursive implementations for computing factorials, computing Fibonacci numbers, summing elements of an array, or reversing the contents of an array. For example, Code Fragment 5.16 provides a nonrecursive method to reverse the contents of an array (as compared to the earlier recursive method from Code Fragment 5.7).

```
1  /** Reverses the contents of the given array. */
2  public static void reverseIterative(int[ ] data) {
3    int low = 0, high = data.length − 1;
4    while (low < high) {                          // swap data[low] and data[high]
5      int temp = data[low];
6      data[low++] = data[high];                   // post-increment of low
7      data[high−−] = temp;                        // post-decrement of high
8    }
9  }
```

Code Fragment 5.16: Reversing the elements of a sequence using iteration.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

5.7 Exercises

Reinforcement

R-5.1 Describe a recursive algorithm for finding the maximum element in an array, A, of n elements. What is your running time and space usage?

R-5.2 Explain how to modify the recursive binary search algorithm so that it returns the index of the target in the sequence or -1 (if the target is not found).

R-5.3 Draw the recursion trace for the computation of $power(2,5)$, using the traditional algorithm implemented in Code Fragment 5.8.

R-5.4 Draw the recursion trace for the computation of $power(2,18)$, using the repeated squaring algorithm, as implemented in Code Fragment 5.9.

R-5.5 Draw the recursion trace for the execution of reverseArray(data, 0, 4), from Code Fragment 5.7, on array data $=$ 4, 3, 6, 2, 6.

R-5.6 Draw the recursion trace for the execution of method PuzzleSolve$(3,S,U)$, from Code Fragment 5.11, where S is empty and $U = \{a,b,c,d\}$.

R-5.7 Describe a recursive algorithm for computing the n^{th} ***Harmonic number***, defined as $H_n = \sum_{k=1}^{n} 1/k$.

R-5.8 Describe a recursive algorithm for converting a string of digits into the integer it represents. For example, `'13531'` represents the integer $13,531$.

R-5.9 Develop a nonrecursive implementation of the version of the power method from Code Fragment 5.9 that uses repeated squaring.

R-5.10 Describe a way to use recursion to compute the sum of all the elements in an $n \times n$ (two-dimensional) array of integers.

Creativity

C-5.11 Describe a recursive algorithm to compute the integer part of the base-two logarithm of n using only addition and integer division.

C-5.12 Describe an efficient recursive algorithm for solving the element uniqueness problem, which runs in time that is at most $O(n^2)$ in the worst case without using sorting.

C-5.13 Give a recursive algorithm to compute the product of two positive integers, m and n, using only addition and subtraction.

C-5.14 In Section 5.2 we prove by induction that the number of *lines* printed by a call to drawInterval(c) is $2^c - 1$. Another interesting question is how many *dashes* are printed during that process. Prove by induction that the number of dashes printed by drawInterval(c) is $2^{c+1} - c - 2$.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

C-5.15 Write a recursive method that will output all the subsets of a set of n elements (without repeating any subsets).

C-5.16 In the ***Towers of Hanoi*** puzzle, we are given a platform with three pegs, a, b, and c, sticking out of it. On peg a is a stack of n disks, each larger than the next, so that the smallest is on the top and the largest is on the bottom. The puzzle is to move all the disks from peg a to peg c, moving one disk at a time, so that we never place a larger disk on top of a smaller one. See Figure 5.15 for an example of the case $n = 4$. Describe a recursive algorithm for solving the Towers of Hanoi puzzle for arbitrary n. (Hint: Consider first the subproblem of moving all but the n^{th} disk from peg a to another peg using the third as "temporary storage.")

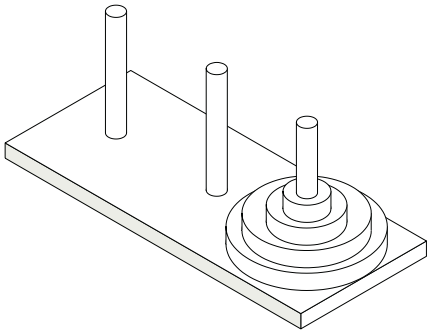


Figure 5.15: An illustration of the Towers of Hanoi puzzle.

C-5.17 Write a short recursive Java method that takes a character string s and outputs its reverse. For example, the reverse of `'pots&pans'` would be `'snap&stop'`.

C-5.18 Write a short recursive Java method that determines if a string s is a palindrome, that is, it is equal to its reverse. Examples of palindromes include `'racecar'` and `'gohangasalamiimalasagnahog'`.

C-5.19 Use recursion to write a Java method for determining if a string s has more vowels than consonants.

C-5.20 Write a short recursive Java method that rearranges an array of integer values so that all the even values appear before all the odd values.

C-5.21 Given an unsorted array, A, of integers and an integer k, describe a recursive algorithm for rearranging the elements in A so that all elements less than or equal to k come before any elements larger than k. What is the running time of your algorithm on an array of n values?

C-5.22 Suppose you are given an array, A, containing n distinct integers that are listed in increasing order. Given a number k, describe a recursive algorithm to find two integers in A that sum to k, if such a pair exists. What is the running time of your algorithm?

C-5.23 Describe a recursive algorithm that will check if an array A of integers contains an integer $A[i]$ that is the sum of two integers that appear earlier in A, that is, such that $A[i] = A[j] + A[k]$ for $j, k < i$.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

C-5.24 Isabel has an interesting way of summing up the values in an array A of n integers, where n is a power of two. She creates an array B of half the size of A and sets $B[i] = A[2i] + A[2i+1]$, for $i = 0, 1, \ldots, (n/2) - 1$. If B has size 1, then she outputs $B[0]$. Otherwise, she replaces A with B, and repeats the process. What is the running time of her algorithm?

C-5.25 Describe a fast recursive algorithm for reversing a singly linked list L, so that the ordering of the nodes becomes opposite of what it was before.

C-5.26 Give a recursive definition of a singly linked list class that does not use any Node class.

Projects

P-5.27 Implement a recursive method with calling signature `find(path, filename)` that reports all entries of the file system rooted at the given path having the given file name.

P-5.28 Write a program for solving summation puzzles by enumerating and testing all possible configurations. Using your program, solve the three puzzles given in Section 5.3.3.

P-5.29 Provide a nonrecursive implementation of the `drawInterval` method for the English ruler project of Section 5.1.2. There should be precisely $2^c - 1$ lines of output if c represents the length of the center tick. If incrementing a counter from 0 to $2^c - 2$, the number of dashes for each tick line should be exactly one more than the number of consecutive 1's at the end of the binary representation of the counter.

P-5.30 Write a program that can solve instances of the Tower of Hanoi problem (from Exercise C-5.16).

Chapter Notes

The use of recursion in programs belongs to the folklore of computer science (for example, see the article of Dijkstra [31]). It is also at the heart of functional programming languages (for example, see the book by Abelson, Sussman, and Sussman [1]). Interestingly, binary search was first published in 1946, but was not published in a fully correct form until 1962. For further discussions on lessons learned, see papers by Bentley [13] and Lesuisse [64].

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.

Copyright © 2014. Wiley. All rights reserved.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data structures and algorithms in java, 6th edition. Retrieved from http://ebookcentral.proquest.com
Created from deakin on 2019-07-04 01:20:00.