 Answers to review questions from Chapter 7

1. Define the terms *recursive* and *iterative.* Is it possible for a function to employ both strategies?

**A *recursive* solution proceeds by breaking a problem down into simpler instances of the same problem. An *iterative* solution uses repetition to perform a set of operations multiple times. It is perfectly reasonable for a function to employ both strategies.**

2. What is the fundamental difference between recursion and traditional stepwise refinement?

**The fundamental difference is that the subproblems in a recursive solution must have the same form as the original problem.**

3. In the pseudocode for the **collectContributions** function, the **if** statement looks like this:

if (n <= 100)

Why is it important to use the **<=** operator instead of simply checking whether **n** is exactly equal to 100?

**If the original target value for contributions is not a power of 10 or is smaller than 100, the recursive decomposition will not produce target values that are exactly equal to 100. By allowing the simple case to collect contributions that are less than $100, the program becomes general enough to work with any nonnegative target value.**

4. What is the standard recursive paradigm?

if (*test for simple case*) {

*Compute a simple solution without using recursion.*

} else {

*Break the problem down into subproblems of the same form.*

*Solve each of the subproblems by calling this function recursively.*

*Reassemble the subproblem solutions into a solution for the whole.*

}

5. What two properties must a problem have for recursion to make sense as a solution strategy?

**1. You must be able to identify *simple cases* for which the answer is easily determined.**

**2. You must be able to identify a *recursive decomposition* that allows you to break any complex instance of the problem into simpler problems of the same form.**

6. Why is the term *divide and conquer* appropriate to recursive techniques?

**Recursive solutions divide large problems up into smaller instances which are then easier to solve (or conquer).**

7. What is meant by the *recursive leap of faith?* Why is this concept important to you as a programmer?

**The *recursive leap of faith* refers to the idea that, in writing a recursive solution, you can assume that smaller instances of the problem work and then reassemble those solutions to solve the original problem. If you fail to adopt the recursive leap of faith, you are forced to trace the operation of the program all the way down to the simple cases, which often makes the process too complex to follow.**

8. In the section entitled “Tracing the recursive process,” the text goes through a long analysis of what happens internally when **fact(4)** is called. Using this section as a model, trace through the execution of **fib(3)**, sketching out each stack frame created in the process.

|  |  |
| --- | --- |
| **Step 1:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame01.png** |
| **Step 2:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame02.png** |
| **Step 3:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame03.png** |
| **Step 4:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame04.png** |
| **Step 5:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame05.png** |
| **Step 6** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame06.png** |
| **Step 7:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame07.png** |
| **Step 8:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame08.png** |
| **Step 9:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame09.png** |
| **Step 10:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame10.png** |
| **Step 11:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame11.png** |
| **Step 12:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame12.png** |
| **Step 13:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame13.png** |
| **Step 14:** | **/Users/eroberts/Books/ProgrammingAbstractionsInC++/chapters/07-IntroductionToRecursion/pictures/FibFrames/FibFrame14.png** |

9. What is a *recurrence relation?*

**A  *recurrence relation* is an expression defining each new element of a sequence in terms of earlier elements.**

10. Modify Fibonacci’s rabbit problem by introducing the additional rule that rabbit pairs stop reproducing after giving birth to three litters. How does this assumption change the recurrence relation? What changes do you need to make in the simple cases?

**The new recurrence relation must subtract out the rabbits that are too old to reproduce, which are all of those born more than four months ago. The recurrence relation therefore looks like this:**

*tn* = *tn*-1 + *tn*-2 – *tn*-5

**The simple case must now take account of the fact that the value of *n* may now range into negative territory, when there were no rabbits. The code for the revised rabbit problem looks like this:**

int rabbits (int n) {

if (n <= 0) {

return 0;

} else if (n == 1) {

return 1;

} else {

return rabbits (n - 1) + rabbits (n - 2) - rabbits (n - 5);

}

}

11. How many times is **fib(1)** called when calculating **fib(n)** using the recursive implementation given in Figure 7‑1?

**The function fib(1)is called fib(n) times.**

12. What is a wrapper function? Why are they often useful in writing recursive functions?

**A *wrapper function* is a function whose only role is to set things up for a call to a more general recursive function that does all the work. In many situations, the recursive process requires additional state information that the wrapper function can provide.**

13. What would happen if you eliminated the **if** **(n** **==** **1)** check from the function **additiveSequence**, so that the implementation looked like this:

int additiveSequence(int n, int t0, int t1) {

if (n == 0) return t0;

return additiveSequence(n - 1, t1, t0 + t1);

}

Would the function still work? Why or why not?

**The function would still work. In the absence of the check, the function calls itself recursively with 0 as the first argument and the value of t1 as the second. Since n will then be 0 in the recursive call, the function will return the original value of t1, just as before.**

14. Why is it important that the implementation of **isPalindrome** in Figure 7‑3 checks for the empty string as well as the single character string? What would happen if the function didn’t check for the single character case and instead checked only whether the length is 0? Would the function still work correctly?

**The check for the empty string is needed to account for strings of even length. In the absence of this check, the code will try to select the initial character of an empty string. The question of whether the single‑character test is necessary depends on a careful reading of the rules for the substr method. When the string gets down to a single character, the call to substr will have arguments of 1 and −1. The documentation of substr indicates that “if [the first argument] is equal to the string length, the function returns the empty string”; given that definition, the implementation will still work.**

15. Explain the effect of the function call

isPalindrome(str, p1 + 1, p2 - 1)

in the **isPalindrome** implementation given in Figure 7‑4.

**In this implementation, the two arguments indicate the character positions that still have to be checked. The characters at the ends of the substring have already been found to match, so the question of whether str is a palindrome hinges on whether the substring starting just after p1 and ending just before p2 are a palindrome, which is precisely what this recursive call asks.**

16. What is mutual recursion?

***Mutual recursion* occurs when one function calls another that—possibly after additional calls through more levels of functions—calls the original function. It therefore represents a recursive process that takes multiple steps.**

17. What would happen if you defined **isEven** and **isOdd** as follows:

bool isEven(unsigned int n) {

return !isOdd(n);

}

bool isOdd(unsigned int n) {

return !isEven(n);

}

Which of the errors explained in the section “Avoiding the common pitfalls” is illustrated by this example?

**This implementation inevitably leads to an infinite recursion. The common pitfall is that the function doesn’t check for simple cases.**

18. The following definitions of **isEven** and **isOdd** are also incorrect:

bool isEven(unsigned int n) {

if (n == 0) {

return true;

} else {

return isOdd(n - 1);

}

}

bool isOdd(unsigned int n) {

if (n == 1) {

return true;

} else {

return isEven(n - 1);

}

}

Give an example that shows how this implementation can fail. What common pitfall is illustrated here?

**This implementation fails if you call isOdd(0) because the function misses the check in isEven and continues a recursive descent through the negative numbers. (Negative values are ruled out as arguments in the discussion, but 0 is nonnegative and therefore is an acceptable argument to isOdd.) The pitfall here is that the recursive simplification does not always hit the simple cases.**