CHAPTER 11

RECURSION

CHAPTER GOALS

To learn to "think recursively"

To be able to use recursive helper functions

To understand the relationship between recursion and iteration

To understand when the use of recursion affects the efficiency of an algorithm

To analyze problems that are much easier to solve by recursion than by iteration

To process data with recursive structures using mutual recursion

CHAPTER CONTENTS

The method of recursion is a powerful technique for breaking up complex computational problems into simpler, often smaller, ones. The term "recursion" refers to the fact that the same computation recurs, or occurs repeatedly, as the problem is solved. Recursion is often the most natural way of thinking about a problem, and there are some computations that are very difficult to perform without recursion. This chapter shows you both simple and complex examples of recursion and teaches you how to "think recursively".

11.1 Triangle Numbers

Chapter 5 contains a simple introduction to writing recursive functions—functions that call themselves with simpler inputs. In that chapter, you saw how to print triangle patterns such as this one:

```
[]
[][]
[][][]
[][][][]
```

The key observation is that you can print a triangle pattern of a given side length, provided you know how to print the smaller triangle pattern that is shown in blue.

In this section, we will modify the example slightly and use recursion to compute the area of a triangle shape of side length *n*, assuming that each `[]` square has area 1. This value is sometimes called the *nth triangle number*. For example, as you can tell from looking at the above triangle, the third triangle number is 6 and the fourth triangle number is 10.

If the side length of the triangle is 1, then the triangle consists of a single square, and its area is 1. Let's take care of this case first.

```
int triangle_area(int side_length)
{
   if (side_length == 1) { return 1; }
   ...
}
```

To deal with the general case, suppose you knew the area of the smaller, blue triangle. Then you could easily compute the area of the larger triangle as

```
smaller_area + side_length
```

How can you get the smaller area? Call the `triangle_area` function!

```
int smaller_side_length = side_length - 1;
int smaller_area = triangle_area(smaller_side_length);
```

Now we can complete the `triangle_area` function:

```
int triangle_area(int side_length)
{
   if (side_length == 1) { return 1; }
   int smaller_side_length = side_length - 1;
   int smaller_area = triangle_area(smaller_side_length);
   return smaller_area + side_length;
}
```

If you prefer, you can implement this function more compactly:

```
int triangle_area(int side_length)
{
   if (side_length == 1) { return 1; }
   else
   {
      return triangle_area(side_length - 1) + side_length;
   }
}
```

We will look at the longer, more explicit form in this section.

Here is a trace of the function call `triangle_area(4)`.

- The `triangle_area` function executes with the parameter variable `side_length` set to 4.
- It sets `smaller_side_length` to 3 and calls `triangle_area` with argument `smaller_side_length`.
  - That function call has its own set of parameter and local variables. Its `side_length` parameter variable is 3, and it sets its `smaller_side_length` variable to 2.
  - The `triangle_area` function is called again, now with argument 2.
    - In that function call, `side_length` is 2 and `smaller_side_length` is 1.
    - The `triangle_area` function is called with argument 1.
      - That function call returns 1.
    - The function call sets `smaller_area` to 1 and returns `smaller_area + side_length` = 1 + 2 = 3.
  - The function call sets `smaller_area` to 3 and returns `smaller_area + side_length` = 3 + 3 = 6.
- The function call sets `smaller_area` to 6 and returns `smaller_area + side_length` = 6 + 4 = 10.

A recursive computation solves a problem by using the solution to the same problem with simpler inputs.

As you can see, the function calls itself multiple times, with ever simpler arguments, until a very simple case is reached. Then the recursive function calls return, one by one.

While it is good to understand this pattern of recursive calls, most people don't find it very helpful to think about the call pattern when designing or understanding a recursive solution. Instead, look at the `triangle_area` function one more time. The first part is very easy to understand. If the side length is 1, then of course the area is 1. The next part is just as reasonable. Compute the area of the smaller triangle. Don't worry how that works—treat the function as a black box and simply assume that you will get the correct answer. Then the area of the larger triangle is clearly the sum of the smaller area and the side length.

When a function keeps calling itself, you may wonder how you know that the calls will eventually come to an end. Two conditions need to be fulfilled:

- Every recursive call must simplify the computation in some way.
- There must be special cases to handle the simplest computations directly.

For a recursion to terminate, there must be special cases for the simplest inputs.

The `triangle_area` function calls itself with a smaller side length. Eventually the side length must reach 1, and there is a special case for computing the area of a triangle with side length 1. Thus, the `triangle_area` function always succeeds.

Actually, you have to be careful. What happens when you call the area of a triangle with side length –1? It computes the area of a triangle with side length –2, which computes the area of a triangle with side length –3, and so on. To avoid this, you should add a condition to the `triangle_area` function:

```
if (side_length <= 0) { return 0; }
```

Recursion is not really necessary to compute the triangle numbers. The area of a triangle equals the sum

```
1 + 2 + 3 + ... + side_length
```

Of course, we can program a simple loop:

```
double area = 0;
for (int i = 1; i <= side_length; i++)
{
   area = area + i;
}
```

Many simple recursions can be computed as loops. However, loop equivalents for more complex recursions—such as the one in our next example—can be complex.

Actually, in this case, you don't even need a loop to compute the answer. The sum of the first n integers can be computed as

$$1 + 2 + \cdots + n = n \times (n+1)/2$$

Thus, the area equals

```
side_length * (side_length + 1) / 2
```

Therefore, neither recursion nor a loop are required to solve this problem. The recursive solution is intended as a "warm-up" for the sections that follow.

ch11/triangle.cpp

```
#include <iostream>

using namespace std;

/**
   Computes the area of a triangle with a given side length.
   @param side_length the side length of the triangle base
   @return the area
*/
int triangle_area(int side_length)
{
   if (side_length <= 0) { return 0; }
   if (side_length == 1) { return 1; }
   int smaller_side_length = side_length - 1;
   int smaller_area = triangle_area(smaller_side_length);
   return smaller_area + side_length;
}

int main()
{
   cout << "Enter the side length: ";
   int input;
   cin >> input;
   cout << "Area: " << triangle_area(input) << endl;
   return 0;
}
```

Program Run

```
Enter the side length: 4
Area: 10
```

SELF CHECK

1. Why is the statement `if (side_length == 1) { return 1; }` in the `triangle_area` function unnecessary?
2. How would you modify the triangle program to recursively compute the area of a square?
3. In some cultures, numbers containing the digit 8 are lucky numbers. What is wrong with the following function that tries to test whether a number is lucky?

```
bool is_lucky(int number)
{
   int last_digit = number % 10;
   if (last_digit == 8) { return true; }
   else
   {
      return is_lucky(number / 10); // Test the number without the last digit
   }
}
```

4. In order to compute a power of two, you can take the next-lower power and double it. For example, if you want to compute 2^{11} and you know that 2^{10} = 1024, then $2^{11} = 2 \times 1024 = 2048$. Write a recursive function `int pow2(int n)` that is based on this observation.
5. Consider the following recursive function:

```
int mystery(int n)
{
   if (n <= 0) { return 0; }
   int smaller_n = n - 1;
   int result = mystery(smaller_n);
   return result + n * n;
}
```

What is `mystery(4)`?

Practice It Now you can try these exercises at the end of the chapter: R11.2, P11.2, P11.6.

Common Error 11.1

Infinite Recursion

A common programming error is an infinite recursion: a function calling itself over and over with no end in sight. The computer needs some amount of memory for bookkeeping for each call. After some number of calls, all memory that is available for this purpose is exhausted. Your program shuts down and reports a "stack fault".

Infinite recursion happens either because the arguments don't get simpler or because a terminating case is missing. For example, suppose the `triangle_area` function computes the area of a triangle with side length `0`. If you do not include a test for this situation, the function calls itself with side length –1, –2, –3, and so on.

Common Error 11.2

Tracing Through Recursive Functions

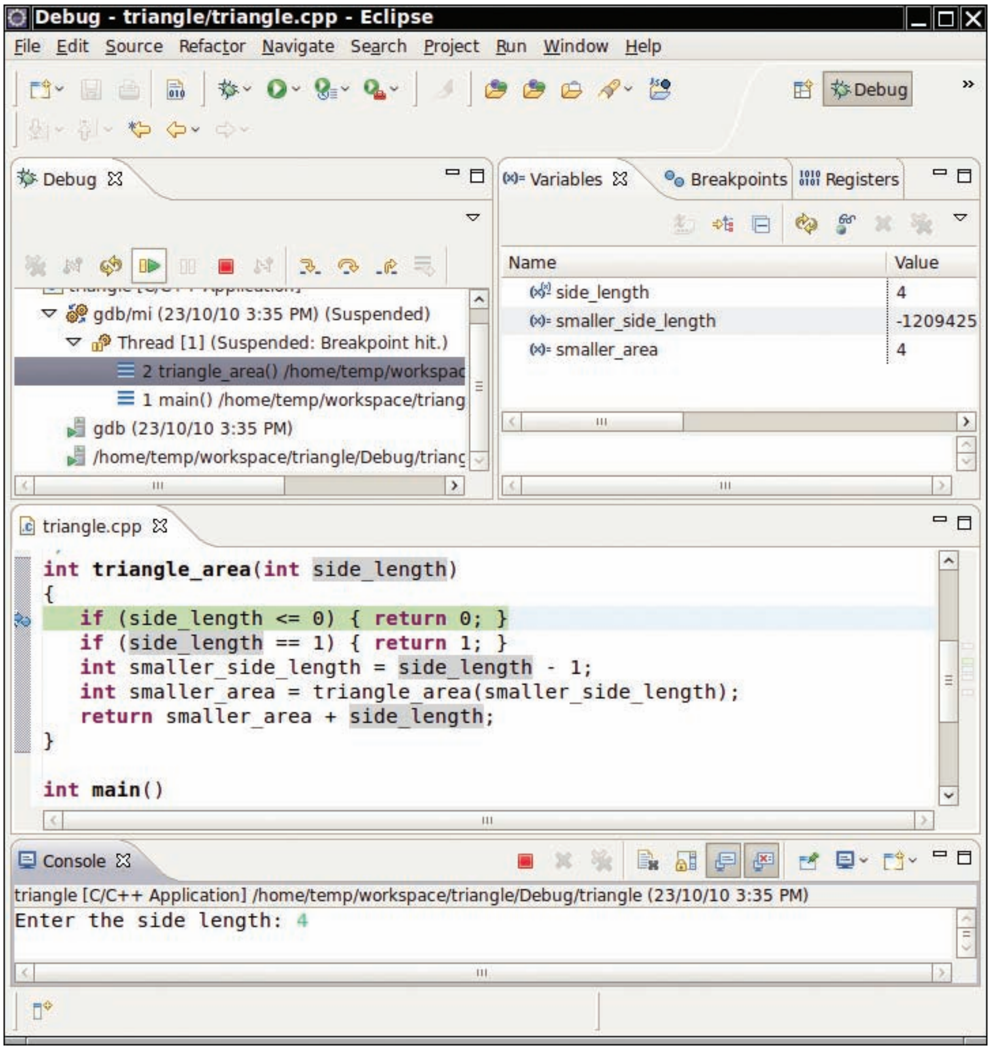
Debugging a recursive function can be somewhat challenging. When you set a breakpoint in a recursive function, the program stops as soon as that program line is encountered in *any call to the recursive function*. Suppose you want to debug the recursive `triangle_area` function. Run until the beginning of the `triangle_area` function (Figure 1). Inspect the `side_length` instance variable. It is 4.

Remove the breakpoint and now run until the statement

```
return smaller_area + side_length;
```

When you inspect `side_length` again, its value is 2! That makes no sense. There was no instruction that changed the value of `side_length`! Is that a bug with the debugger?

No. The program stopped in the first *recursive* call to `triangle_area` that reached the `return` statement. If you are confused, look at the *call stack* (Figure 2). You will see that three calls to `triangle_area` are pending.


Debug - triangle/triangle.cpp - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Debug
Variables
Breakpoints
Registers
triangle.cpp
Console

```
gdb/mi (23/10/10 3:35 PM) (Suspended)
  Thread [1] (Suspended: Breakpoint hit.)
    2 triangle_area() /home/temp/workspac
    1 main() /home/temp/workspace/triang
gdb (23/10/10 3:35 PM)
/home/temp/workspace/triangle/Debug/trianc
```

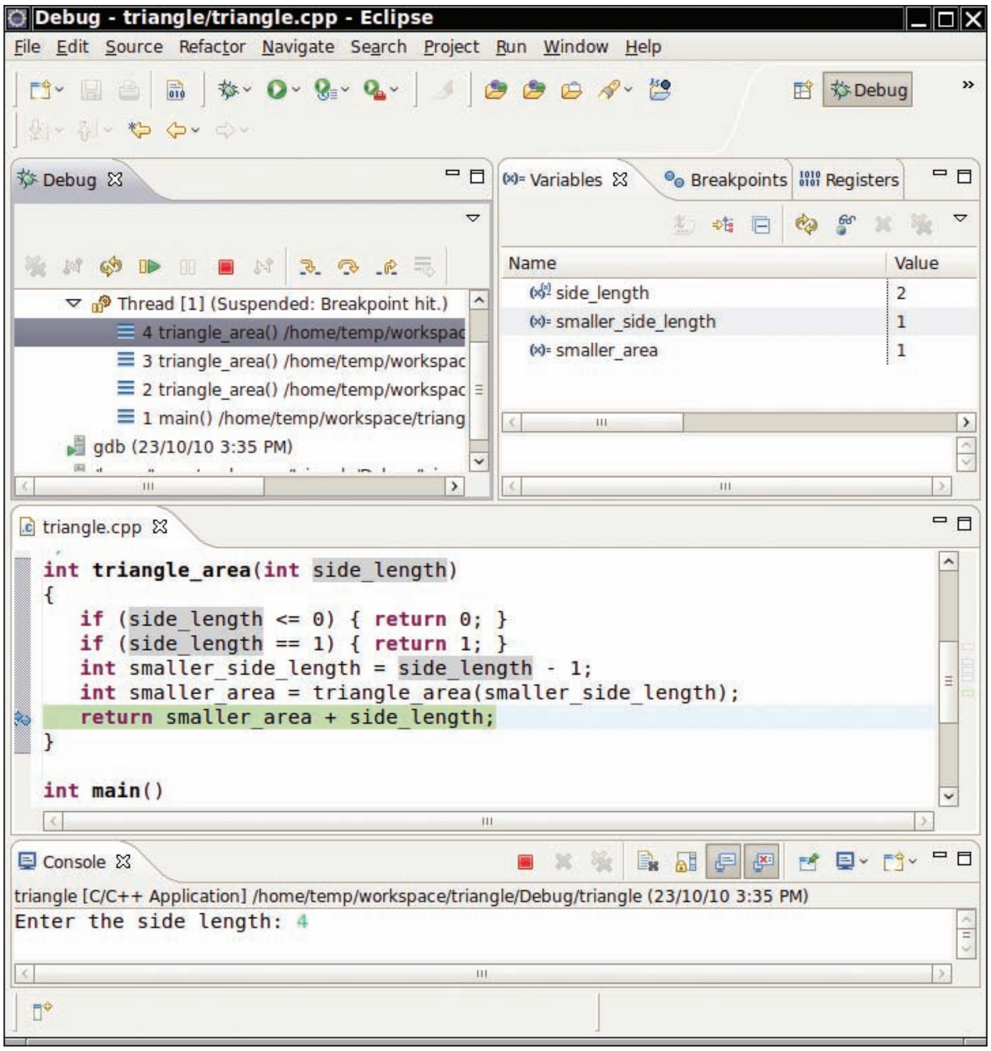
Name	Value
side_length	4
smaller_side_length	-1209425
smaller_area	4

```
int triangle_area(int side_length)
{
   if (side_length <= 0) { return 0; }
   if (side_length == 1) { return 1; }
   int smaller_side_length = side_length - 1;
   int smaller_area = triangle_area(smaller_side_length);
   return smaller_area + side_length;
}

int main()
```

```
triangle [C/C++ Application] /home/temp/workspace/triangle/Debug/triangle (23/10/10 3:35 PM)
Enter the side length: 4
```

Figure 1 Debugging a Recursive Function


Debug - triangle/triangle.cpp - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Debug
Variables
Breakpoints
Registers
triangle.cpp
Console

```
Thread [1] (Suspended: Breakpoint hit.)
    4 triangle_area() /home/temp/workspac
    3 triangle_area() /home/temp/workspac
    2 triangle_area() /home/temp/workspac
    1 main() /home/temp/workspace/triang
gdb (23/10/10 3:35 PM)
```

Name	Value
side_length	2
smaller_side_length	1
smaller_area	1

```
int triangle_area(int side_length)
{
   if (side_length <= 0) { return 0; }
   if (side_length == 1) { return 1; }
   int smaller_side_length = side_length - 1;
   int smaller_area = triangle_area(smaller_side_length);
   return smaller_area + side_length;
}

int main()
```

```
triangle [C/C++ Application] /home/temp/workspace/triangle/Debug/triangle (23/10/10 3:35 PM)
Enter the side length: 4
```

Figure 2 Three Calls to `triangle_area` Are Pending

You can debug recursive functions with the debugger. You just need to be particularly careful, and watch the call stack to understand which nested call you currently are in.

11.2 Thinking Recursively

How To 5.2 in Chapter 5 tells you how to solve a problem recursively by pretending that “someone else” will solve the problem for simpler inputs and by focusing on how to turn the simpler solutions into a solution for the whole problem.

In this section, we walk through these steps with a more complex problem: testing whether a sentence is a *palindrome*—a string that is equal to itself when you reverse all characters. Typical examples of palindromes are:

- rotor
- A man, a plan, a canal—Panama!
- Go hang a salami, I'm a lasagna hog

and, of course, the oldest palindrome of all:

- Madam, I'm Adam

Our goal is to implement the function

```
bool is_palindrome(string s)
```

For simplicity, assume for now that the string has only lowercase letters and no punctuation marks or spaces. Exercise P11.14 asks you to generalize the function to arbitrary strings.

Step 1 Break the input into parts that can themselves be inputs to the problem.

The key step in finding a recursive solution is reducing the input to a simpler input for the same problem.

When you consider simpler inputs, you may want to remove just a little bit from the original input—maybe remove one or two characters from a string, or remove a small portion of a geometric shape. But sometimes it is more useful to cut the input in half and then see what it means to solve the problem for both halves.

In the palindrome test problem, the input is the string that we need to test. How can you simplify the input? Here are several possibilities:

- Remove the first character.
- Remove the last character.
- Remove both the first and the last character.
- Remove a character from the middle.
- Cut the string into two halves.

These simpler inputs are all potential inputs for the palindrome test.

Step 2 Combine solutions with simpler inputs to a solution of the original problem.

When designing a recursive solution, do not worry about multiple nested calls. Simply focus on reducing a problem to a slightly simpler one.

You should simply assume that the solutions for the simpler problem are available, without worrying how they were obtained. In this step, ask yourself how you can turn the solution for the simpler inputs into a solution for the input that you are currently thinking about. Maybe you need to add a small quantity, related to the quantity that you lopped off to arrive at the simpler input. Maybe you cut the original input in two halves and have solutions for both halves. Then you may need to add both solutions to arrive at a solution for the whole.

Consider the methods for simplifying the inputs for the palindrome test. Cutting the string in half doesn't seem a good idea. If you cut

```
"rotor"
```

in half, you get two strings:

```
"rot"
```

and

```
"or"
```

Neither of them is a palindrome. Cutting the input in half and testing whether the halves are palindromes seems a dead end.

The most promising simplification is to remove the first *and* last characters, provided they match. Removing the `r` at the front and back of `"rotor"` yields

```
"oto"
```

Assume you can verify that the shorter string is a palindrome. Then *of course* the original string is a palindrome—we put the same letter in the front and the back. That's extremely promising. A word is a palindrome if

- The first and last letters match, and
- The word obtained by removing the first and last letters is a palindrome.

Again, don't worry how the test works for the shorter string. It just works.

Step 3 Find solutions to the simplest inputs.

A recursive function keeps calling itself with simpler inputs. Eventually, the function must stop calling itself. You must deal with the simplest inputs separately. Usually, it is an easy matter to come up with solutions for the simplest inputs.

However, sometimes you get into philosophical questions dealing with *degenerate* inputs: empty strings, shapes with no area, and so on. Then you may want to investigate a slightly larger input that gets reduced to such a trivial input and see what value you should attach to the degenerate inputs so that the simpler value, when used according to the rules you discovered in Step 2, yields the correct answer.

Look at the simplest strings for the palindrome test:

- Strings with two characters
- Strings with a single character
- The empty string

You don't have to come up with a special solution for strings with two characters. Step 2 still applies to those strings—either or both of the characters are removed. But you do need to worry about strings of length `0` and 1. In those cases, Step 2 can't apply. There aren't two characters to remove.

A string with a single character, such as `"I"`, is a palindrome.

The empty string is a palindrome—it's the same string when you read it backwards. If you find that too artificial, consider a string `"oo"`. According to the rule discovered in Step 2, this string is a palindrome if the first and last character of that string match and the remainder—that is, the empty string—is also a palindrome. Therefore, it makes sense to consider the empty string a palindrome.

Thus, all strings of length `0` or 1 are palindromes.

Step 4 Implement the solution by combining the simple cases and the reduction step.

Usually, you want to get the simplest cases out of the way. In the case of the palindrome test, those are the strings with length `0` or 1. If the input isn't one of the simplest cases, then make one or more recursive calls with simpler inputs and use the results of those calls to complete your function.

The following program shows the complete `is_palindrome` function:

ch11/palindrome.cpp

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

/**
   Tests whether a string is a palindrome. A palindrome
   is equal to its reverse, for example “rotor” or “racecar”.
   @param s a string
   @return true if s is a palindrome
*/
bool is_palindrome(string s)
{
   // Separate case for shortest strings
   if (s.length() <= 1) { return true; }

   // Get first and last character, converted to lowercase
   char first = s[0];
   char last = s[s.length() - 1];

   if (first == last)
   {
      string shorter = s.substr(1, s.length() - 2);
      return is_palindrome(shorter);
   }
   else
   {
      return false;
   }
}

int main()
{
   cout << "Enter a string: ";
   string input;
   getline(cin, input);
   cout << input << " is ";
   if (!is_palindrome(input)) { cout << "not "; }
   cout << "a palindrome." << endl;
   return 0;
}
```

Program Run

```
Enter a string: aibohphobia
aibohphobia is a palindrome.
```

SELF CHECK

6. Consider the task of removing all punctuation marks from a string. How can we break the string into smaller strings that can be processed recursively?
7. In a recursive function that removes all punctuation marks from a string, we decide to remove the last character, then recursively process the remainder. How do you combine the results?
8. How do you find solutions for the simplest inputs when removing punctuation marks from a string?

9. Provide pseudocode for a recursive function that removes punctuation marks from a string, using the answers to Self Checks 6–8.

Practice It Now you can try these exercises at the end of the chapter: R11.3, R11.4, P11.3, P11.8.

11.3 Recursive Helper Functions

Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

Sometimes it is easier to find a recursive solution if you change the original problem slightly. Then the original problem can be solved by calling a recursive helper function.

Here is a typical example. Consider the palindrome test of Section 11.2. It is a bit inefficient to construct new `string` objects in every step. Now consider the following change in the problem. Rather than testing whether the entire string is a palindrome, check whether a substring is a palindrome:

```
/*
   Tests whether a substring of a string is a palindrome.
   @param s the string to test
   @param start the index of the first character of the substring
   @param end the index of the last character of the substring
   @return true if the substring is a palindrome
*/
bool substring_is_palindrome(string s, int start, int end)
```

This function turns out to be even easier to implement than the original test. In the recursive calls, simply adjust the `start` and `end` arguments to skip over matching letter pairs. There is no need to construct new string objects to represent the shorter strings.

```
bool substring_is_palindrome(string s, int start, int end)
{
   // Separate case for substrings of length 0 and 1
   if (start >= end) { return true; }

   if (s[start] == s[end])
   {
      // Test substring that doesn't contain the first and last letters
      return substring_is_palindrome(s, start + 1, end - 1);
   }
   else
   {
      return false;
   }
}
```

You should supply a function to solve the whole problem—the user of your function shouldn't have to know about the trick with the substring positions. Simply call the helper function with positions that test the entire string:

```
bool is_palindrome(string s)
{
   return substring_is_palindrome(s, 0, s.length() - 1);
}
```

Note that the `is_palindrome` function is *not* recursive. It just calls a recursive helper function.

Use the technique of recursive helper functions whenever it is easier to solve a recursive problem that is slightly different from the original problem.

SELF CHECK

10. When does the recursive `substring_is_palindrome` function stop calling itself?

11. To compute the sum of the values in an array, add the first value to the sum of the remaining values, computing recursively. Of course, it would be inefficient to set up an actual array of the remaining values. Which recursive helper function can solve the problem?

12. How can you write a recursive function `sum(int a[], int size)` without needing a helper function?

Practice It Now you can try these exercises at the end of the chapter: P11.4, P11.7, P11.15.

11.4 The Efficiency of Recursion

As you have seen in this chapter, recursion can be a powerful tool for implementing complex algorithms. On the other hand, recursion can lead to algorithms that perform poorly. In this section, we will analyze the question of when recursion is beneficial and when it is inefficient.

Occasionally, a recursive solution runs much more slowly than its iterative counterpart. However, in most cases, the recursive solution runs at about the same speed.

The Fibonacci sequence is a sequence of numbers defined by the equation

$$\begin{aligned} f_1 &= 1 \\ f_2 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \end{aligned}$$

That is, each value of the sequence is the sum of the two preceding values. The first ten terms of the sequence are

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55
```

It is easy to extend this sequence indefinitely. Just keep appending the sum of the last two values of the sequence. For example, the next entry is 34 + 55 = 89.

We would like to write a function that computes f_n for any value of n. Here is a program that translates the definition directly into a recursive function:

ch11/fibtest.cpp

```cpp
#include <iostream>

using namespace std;

/**
   Computes a Fibonacci number.
   @param n an integer
   @return the nth Fibonacci number
*/
int fib(int n)
{
   if (n <= 2) { return 1; }
   else { return fib(n - 1) + fib(n - 2); }
}

int main()
{
   cout << "Enter n: ";
   int n;
   cin >> n;
   int f = fib(n);
   cout << "fib(" << n << ") = " << f << endl;
   return 0;
}
```

Program Run

```
Enter n: 6
fib(6) = 8
```

That is certainly simple, and the function will work correctly. But watch the output closely as you run the test program. For small input values, the program is quite fast. Even for moderately large values, though, the program pauses an amazingly long time between outputs. Try out some numbers between 30 and 50 to see this effect.

That makes no sense. Armed with pencil, paper, and a pocket calculator you could calculate these numbers pretty quickly, so it shouldn't take the computer long.

To determine the problem, insert trace messages into the function:

ch11/fibtrace.cpp

```
#include <iostream>

using namespace std;

/**
   Computes a Fibonacci number.
   @param n an integer
   @return the nth Fibonacci number
*/
int fib(int n)
{
   cout << "Entering fib: n = " << n << endl;
   int f;
   if (n <= 2) { f = 1; }
   else { f = fib(n - 1) + fib(n - 2); }
   cout << "Exiting fib: n = " << n
      << " return value = " << f << endl;
   return f;
}

int main()
{
   cout << "Enter n: ";
   int n;
   cin >> n;
   int f = fib(n);
   cout << "fib(" << n << ") = " << f << endl;
   return 0;
}
```

Program Run

```
Enter n: 6
Entering fib: n = 6
Entering fib: n = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Exiting fib: n = 5 return value = 5
Entering fib: n = 4
Entering fib: n = 3
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Entering fib: n = 1
Exiting fib: n = 1 return value = 1
Exiting fib: n = 3 return value = 2
Entering fib: n = 2
Exiting fib: n = 2 return value = 1
Exiting fib: n = 4 return value = 3
Exiting fib: n = 6 return value = 8
fib(6) = 8
```

Figure 3 shows the call tree.

Now it is becoming apparent why the function takes so long. It is computing the same values over and over. For example, the computation of `fib(6)` calls `fib(4)` twice and `fib(3)` three times. That is very different from the computation you would do with pencil and paper. There you would just write down the values as they were

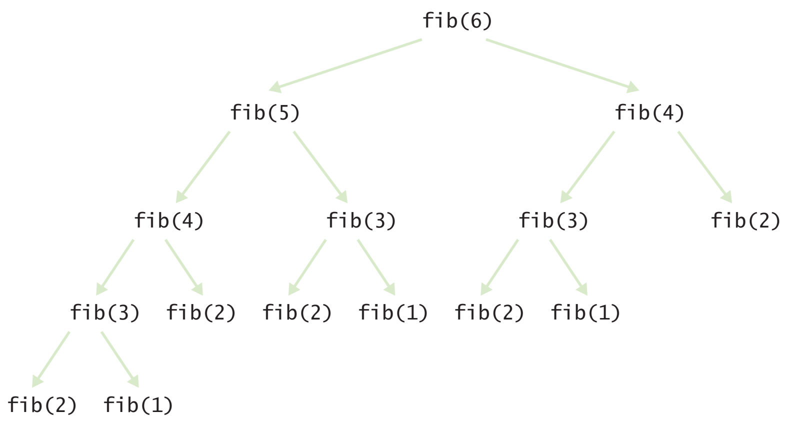


Figure 3 Call Pattern of the Recursive `fib` Function

computed and add up the last two to get the next one until you reached the desired entry; no sequence value would ever be computed twice.

If you imitate the pencil-and-paper process, then you get the following program:

ch11/fibloop.cpp

```
#include <iostream>

using namespace std;

/**
   Computes a Fibonacci number.
   @param n an integer
   @return the nth Fibonacci number
*/
int fib(int n)
{
   if (n <= 2) { return 1; }
   int fold = 1;
   int fold2 = 1;
   int fnew;
   for (int i = 3; i <= n; i++)
   {
      fnew = fold + fold2;
      fold2 = fold;
      fold = fnew;
   }
   return fnew;
}

int main()
{
   cout << "Enter n: ";
   int n;
   cin >> n;
   int f = fib(n);
   cout << "fib(" << n << ") = " << f << endl;
   return 0;
}
```

The `fib` function in this program runs *much* faster than the recursive version.

In this example of the `fib` function, the recursive solution was easy to program because it exactly followed the mathematical definition, but it ran far more slowly than the iterative solution, because it computed many intermediate results multiple times.

Can you always speed up a recursive solution by changing it into a loop? Frequently, the iterative and recursive solution have essentially the same performance. For example, here is an iterative solution for the palindrome test:

```
bool is_palindrome(string s)
{
   int start = 0;
   int end = s.length() - 1;
   while (start < end)
   {
      if (s[start] != s[end]) { return false; }
      start++;
      end--;
      }
      return true;
   }
```

This solution keeps two index variables: `start` and `end`. The `start` index starts at the beginning of the string, and the `end` index at the end of the string. Whenever the characters at `start` and `end` match, the `start` index is incremented and the `end` index is decremented. When the two index variables meet, then the iteration stops.

The iteration and the recursion do essentially the same work. If a palindrome has *n* characters, the iteration executes the loop *n* / 2 times, comparing a pair of characters in each iteration. Similarly, the recursive solution calls itself *n* / 2 times, because two characters are compared and removed in each step.

In such a situation, the iterative solution tends to be a bit faster, because each recursive function call takes a certain amount of processor time. In principle, it is possible for a smart compiler to avoid recursive function calls if they follow simple patterns, but most compilers don't do that. From that point of view, an iterative solution is preferable.

In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

Often, recursive solutions are easier to understand and implement correctly than their iterative counterparts. There is a certain elegance and economy of thought to recursive solutions that makes them more appealing. As the computer scientist (and creator of the GhostScript interpreter for the PostScript graphics description language) L. Peter Deutsch put it: "To iterate is human, to recurse divine."

SELF CHECK

13. Is it faster to compute the triangle numbers recursively, as shown in Section 11.1, or is it faster to use a loop that computes `1 + 2 + 3 + . . . + side_length`?

14. You can compute the factorial function either with a loop, using the definition that $n! = 1 \times 2 \times \ldots \times n$, or recursively, using the definition that $0! = 1$ and $n! = (n-1)! \times n$. Is the recursive approach inefficient in this case?

15. To compute the sum of the values in an array, you can split the array in the middle, recursively compute the sums of the halves, and add the results. Compare the performance of this algorithm with that of a loop that adds the values.

Practice It Now you can try these exercises at the end of the chapter: R11.9, R11.10, P11.5.

11.5 Permutations

In this section, we consider a more complex example of recursion that would be difficult to program with a simple loop. Our task is to generate all **permutations** of a string. A permutation is simply a rearrangement of the letters. For example, the string `"eat"` has six permutations (including the original string itself):

```
"eat"
"eta"
"aet"
"ate"
"tea"
"tae"
```

We will develop a function

```
vector<string> generate_permutations(string word)
```

that generates all permutations of a word.

Here is how you would use the function. The following code displays all permutations of the string "eat":

```
vector<string> v = generate_permutations("eat");
for (int i = 0; i < v.size(); i++)
{
   cout << v[i] << endl;
}
```

Now you need a way to generate the permutations recursively. Consider the string "eat" and simplify the problem. First, generate all permutations that start with the letter 'e', then those that start with 'a', and finally those that start with 't'. How do you generate the permutations that start with 'e'? You need to know the permutations of the substring "at". But that's the same problem—to generate all permutations—with a simpler input, namely the shorter string "at". Using recursion generates the permutations of the substring "at". You will get the strings

```
"at"
"ta"
```

For each result of the simpler problem, add the letter 'e' in front. Now you have all permutations of "eat" that start with 'e', namely

```
"eat"
"eta"
```

Next, turn your attention to the permutations of "eat" that start with 'a'. You must create the permutations of the remaining letters, "et", namely

```
"et"
"te"
```

Add the letter 'a' to the front of the strings and obtain

```
"aet"
"ate"
```

Generate the permutations that start with 't' in the same way.

That's the idea. To carry it out, you must implement a loop that iterates through the character positions of the word. Each loop iteration creates a shorter word that omits the current position:

```
vector<string> generate_permutations(string word)
{
   vector<string> result;
   ...
   for (int i = 0; i < word.length(); i++)
   {
      string shorter_word = word.substr(0, i) + word.substr(i + 1);
      ...
   }
   return result;
}
```

The next step is to compute the permutations of the shorter word.

```
vector<string> shorter_permutations = generate_permutations(shorter_word);
```

For each of the shorter permutations, add the omitted letter:

```
for (int j = 0; j < shorter_permutations.size(); j++)
{
   string longer_word = word[i] + shorter_permutations[j];
   result.push_back(longer_word);
}
```

The permutation generation algorithm is recursive—it uses the fact that we can generate the permutations of shorter words. When does the recursion stop? The simplest possible string is the empty string, which has a single permutation—itself.

```
vector<string> generate_permutations(string word)
{
   vector<string> result;
   if (word.length() == 0)
   {
      result.push_back(word);
      return result;
   }
   . . .
}
```

For generating permutations, it is much easier to use recursion than iteration.

Could you generate the permutations without recursion? There is no obvious way of writing a loop that iterates through all permutations. Exercise P11.13 shows that there is an iterative solution, but it is far more difficult to understand than the recursive algorithm.

Here is the complete permutation program:

ch11/permute.cpp

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

/**
   Generates all permutations of the characters in a string.
   @param word a string
   @return a vector that is filled with all permutations
   of the word
*/
vector<string> generate_permutations(string word)
{
   vector<string> result;
   if (word.length() == 0)
   {
      result.push_back(word);
      return result;
   }

   for (int i = 0; i < word.length(); i++)
   {
      string shorter_word = word.substr(0, i)
         + word.substr(i + 1);
      vector<string> shorter_permutations
         = generate_permutations(shorter_word);
      for (int j = 0; j < shorter_permutations.size(); j++)
      {
         string longer_word = word[i] + shorter_permutations[j];
         result.push_back(longer_word);
      }
   }
   return result;
}

int main()
{
   cout << "Enter a string: ";
   string input;
   getline(cin, input);
   vector<string> v = generate_permutations(input);
   for (int i = 0; i < v.size(); i++)
   {
      cout << v[i] << endl;
   }
   return 0;
}
```

Program Run

```
Enter a string: arm
arm
amr
ram
rma
mar
mra
```

SELF CHECK

16. What are all permutations of the four-letter word beat?

17. Our recursion for computing permutations stops at the empty string. What simple modification would make the recursion stop at strings of length 0 or 1?

18. Why isn't it easy to develop an iterative solution for computing permutations?

Practice It Now you can try these exercises at the end of the chapter: R11.5, P11.10, P11.11.

11.6 Mutual Recursion

In a mutual recursion, a set of cooperating functions calls each other repeatedly.

In the preceding examples, a function called itself to solve a simpler problem. Sometimes, a set of cooperating functions calls each other in a recursive fashion. In this section, we will explore a typical situation of such a *mutual recursion*.

We will develop a program that can compute the values of arithmetic expressions such as

```
3 + 4 * 5
(3 + 4) * 5
1 - (2 - (3 - (4 - 5)))
```

Computing such an expression is complicated by the fact that * and / bind more strongly than + and -, and that parentheses can be used to group subexpressions.

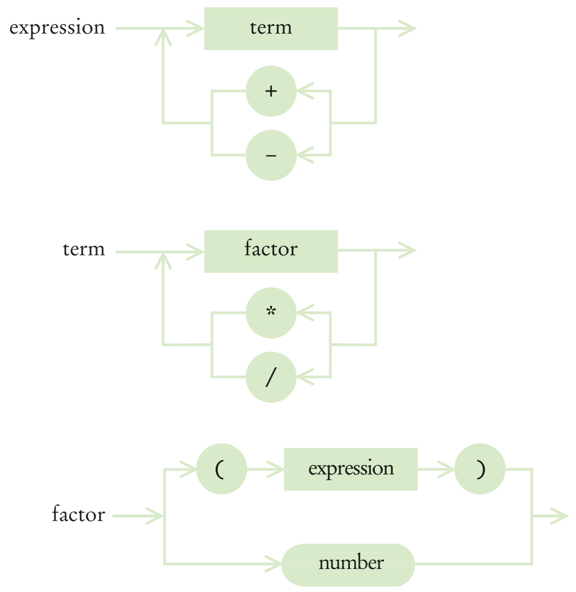


Figure 4 Syntax Diagrams for Evaluating an Expression

Figure 4 shows a set of *syntax diagrams* that describes the syntax of these expressions. An expression is either a term, or a sum or difference of terms. A term is either a factor, or a product or quotient of factors. Finally, a factor is either a number or an expression enclosed in parentheses.

Figure 5 shows how the expressions `3 + 4 * 5` and `(3 + 4) * 5` are derived from the syntax diagram.

Why do the syntax diagrams help us compute the value of the tree? If you look at the syntax trees, you will see that they accurately represent which operations should be carried out first. In the first tree, 4 and 5 should be multiplied, and then the result should be added to 3. In the second tree, 3 and 4 should be added, and the result should be multiplied by 5.

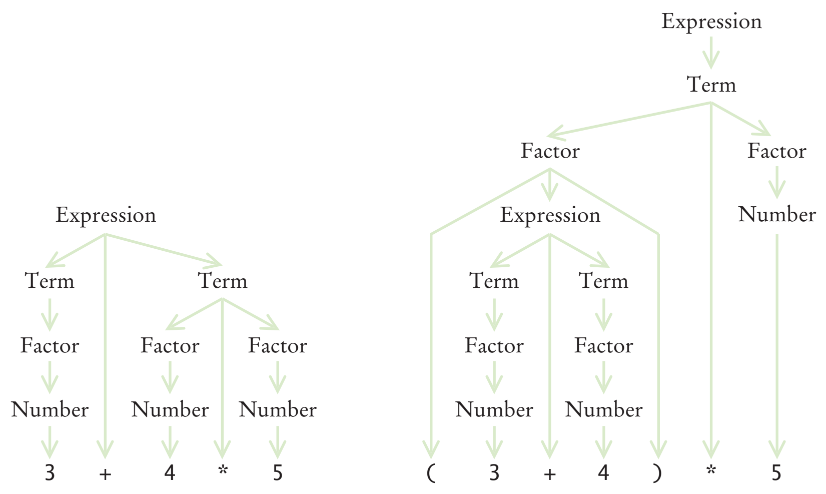


Figure 5 Syntax Trees for Two Expressions

To compute the value of an expression, we implement three functions: expression_value, term_value, and factor_value. The expression_value function first calls term_value to get the value of the first term of the expression. Then it checks whether the next input character is one of + or -. If so, it calls term_value again and adds or subtracts it:

```
int expression_value()
{
   int result = term_value();
   bool more = true;
   while (more)
   {
      char op = cin.peek();
      if (op == '+' || op == '-')
      {
         cin.get();
         int value = term_value();
         if (op == '+') { result = result + value; }
         else { result = result - value; }
      }
      else { more = false; }
   }
   return result;
}
```

The term_value function calls factor_value in the same way, multiplying or dividing the factor values.

Finally, the factor_value function checks whether the next input character is a '(' or a digit. In the latter case, the value is simply the value of the number. However, if the function sees a parenthesis, the factor_value function makes a recursive call to expression_value. Thus, the three functions are mutually recursive.

```
int factor_value()
{
   int result = 0;
   char c = cin.peek();
   if (c == '(')
   {
      cin.get();
      result = expression_value();
      cin.get(); // Read ")"
   }
   else // Assemble number value from digits
   {
      while (isdigit(c))
      {
         result = 10 * result + c - '0';
         cin.get();
         c = cin.peek();
      }
   }
   return result;
}
```

As always with a recursive solution, you need to ensure that the recursion terminates. In this situation, that is easy to see. If expression_value calls itself, the second call works on a shorter subexpression than the original expression. At each recursive call, at least some of the characters of the input are consumed, so eventually the recursion must come to an end.

ch11/eval.cpp

```
#include <iostream>
#include <cctype>

using namespace std;

int term_value();
int factor_value();

/**
   Evaluates the next expression found in cin.
   @return the value of the expression
*/
int expression_value()
{
   int result = term_value();
   bool more = true;
   while (more)
   {
      char op = cin.peek();
      if (op == '+' || op == '-')
      {
         cin.get();
         int value = term_value();
         if (op == '+') { result = result + value; }
         else { result = result - value; }
      }
      else { more = false; }
   }
   return result;
}

/**
   Evaluates the next term found in cin.
   @return the value of the term.
*/
int term_value()
{
   int result = factor_value();
   bool more = true;
   while (more)
   {
      char op = cin.peek();
      if (op == '*' || op == '/')
      {
         cin.get();
         int value = factor_value();
         if (op == '*') { result = result * value; }
         else { result = result / value; }
      }
      else { more = false; }
   }
   return result;
}

/**
   Evaluates the next factor found in cin.
   @return the value of the factor.
*/
```

```
int factor_value()
{
   int result = 0;
   char c = cin.peek();
   if (c == '(')
   {
      cin.get();
      result = expression_value();
      cin.get(); // Read ")"
   }
   else // Assemble number value from digits
   {
      while (isdigit(c))
      {
         result = 10 * result + c - '0';
         cin.get();
         c = cin.peek();
      }
   }
   return result;
}

int main()
{
   cout << "Enter an expression: ";
   cout << expression_value() << endl;
   return 0;
}
```

Program Run

```
Enter an expression: 1+12*12*12
1729
```

SELF CHECK

19. What is the difference between a term and a factor? Why do we need both concepts?

20. Why does the expression evaluator use mutual recursion?

21. What happens if you try to evaluate the illegal expression `3+4*)5`?

Practice It Now you can try these exercises at the end of the chapter: R11.11, P11.17.

CHAPTER SUMMARY

Understand the control flow in a recursive computation.

- A recursive computation solves a problem by using the solution to the same problem with simpler inputs.
- For a recursion to terminate, there must be special cases for the simplest inputs.

Design a recursive solution to a problem.

- The key step in finding a recursive solution is reducing the input to a simpler input for the same problem.
- When designing a recursive solution, do not worry about multiple nested calls. Simply focus on reducing a problem to a slightly simpler one.

Identify recursive helper methods for solving a problem.

- Sometimes it is easier to find a recursive solution if you make a slight change to the original problem.

Contrast the efficiency of recursive and non-recursive algorithms.

- Occasionally, a recursive solution runs much more slowly than its iterative counterpart. However, in most cases, the recursive solution runs at about the same speed.
- In many cases, a recursive solution is easier to understand and implement correctly than an iterative solution.

Review a complex recursion example that cannot be solved with a simple loop.

- For generating permutations, it is much easier to use recursion than iteration.

Recognize the phenomenon of mutual recursion in an expression evaluator.

- In a mutual recursion, a set of cooperating functions calls each other repeatedly.

REVIEW EXERCISES

R11.1 Define the terms

a. recursion

b. iteration

c. infinite recursion

d. mutual recursion

R11.2 Give pseudocode for a recursive algorithm that replaces all digits with value 8 in a number with zeroes.

R11.3 Outline, but do not implement, a recursive solution for finding the smallest value in an array.

R11.4 Outline, but do not implement, a recursive solution for sorting an array of numbers. *Hint:* First find the smallest value in the array.

R11.5 Outline, but do not implement, a recursive solution for generating all subsets of the set {1, 2, … , n}.

R11.6 Exercise P11.13 shows an iterative way of generating all permutations of the sequence (0, 1, … , $n - 1$). Explain why the algorithm produces the right result.

R11.7 Write a recursive definition of x^n, where $x \geq 0$, similar to the recursive definition of the Fibonacci numbers. *Hint:* How do you compute x^n from x^{n-1}? How does the recursion terminate?

R11.8 Write a recursive definition of $n! = 1 \times 2 \times \ldots \times n$, similar to the recursive definition of the Fibonacci numbers.

R11.9 Find out how often the recursive version of `fib` calls itself. Keep a global variable `fib_count` and increment it once in every call of `fib`. What is the relationship between `fib(n)` and `fib_count`?

R11.10 How many moves are required to move n disks in the "Towers of Hanoi" problem of Exercise P11.15? *Hint:* As explained in the exercise,

$$\text{moves}(1) = 1$$
$$\text{moves}(n) = 2 \cdot \text{moves}(n-1) + 1$$

R11.11 Trace the expression evaluator program from Section 11.6 with inputs `3 + 4 * 5` and `3 + (4 * 5)`.

PROGRAMMING EXERCISES

P11.1 If a string has n letters, then the *number* of permutations is given by the factorial function:

$$n! = 1 \times 2 \times 3 \times \ldots \times n$$

For example, $3! = 1 \times 2 \times 3 = 6$ and there are six permutations of the three-character string `"eat"`. Implement a recursive `factorial` function, using the definitions

$$n! = (n-1)! \times n$$

and

$$0! = 1$$

P11.2 Write a recursive function that prints an integer with decimal separators. For example, 12345678 should be printed as `12,345,678`.

P11.3 Write a recursive function `void reverse()` that reverses a sentence. For example:

```
Sentence greeting = new Sentence("Hello!");
greeting.reverse();
cout << greeting.get_text() << "\n";
```

prints the string `"!olleH"`. Implement a recursive solution by removing the first character, reversing a sentence consisting of the remaining text, and combining the two.

P11.4 Redo Exercise P11.3 with a recursive helper function that reverses a substring of the message text.

P11.5 Implement the `reverse` function of Exercise P11.3 as an iteration.

P11.6 Use recursion to implement a function `bool find(string s, string t)` that tests whether a string `t` is contained in a string `s`:

```
bool b = s.find("Mississippi!", "sip"); // Returns true
```

Hint: If the text starts with the string you want to match, then you are done. If not, consider the sentence that you obtain by removing the first character.

P11.7 Use recursion to implement a function `int index_of(string s, string t)` that returns the starting position of the first substring of the string `s` that matches `t`. Return –1 if `t` is not a substring of `s`. For example,

```
int n = s.index_of("Mississippi!", "sip"); // Returns 6
```

Hint: This is a bit trickier than Exercise P11.6, because you need to keep track of how far the match is from the beginning of the sentence. Make that value an argument for a helper function.

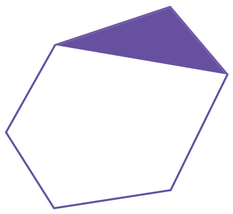
P11.8 Using recursion, find the largest element in a vector of integer values:

```
int maximum(vector<int> values)
```

Hint: Find the largest element in the subset containing all but the last element. Then compare that maximum to the value of the last element.

P11.9 Using recursion, compute the sum of all values in an array.

P11.10 Using recursion, compute the area of a polygon. Cut off a triangle and use the fact that a triangle with corners (x_1, y_1), (x_2, y_2), (x_3, y_3) has area $(x_1y_2 + x_2y_3 + x_3y_1 - y_1x_2 - y_2x_3 - y_3x_1)/2$.



P11.11 Implement a function

```
vector<string> generate_substrings(string s)
```

that generates all substrings of a string. For example, the substrings of the string `"rum"` are the seven strings

```
"r", "ru", "rum", "u", "um", "m", ""
```

Hint: First enumerate all substrings that start with the first character. There are n of them if the string has length n. Then enumerate the substrings of the string that you obtain by removing the first character.

P11.12 Implement a function

```
vector<string> generate_subsets(string s)
```

that generates all subsets of characters of a string. For example, the subsets of characters of the string `"rum"` are the eight strings

```
"rum", "ru", "rm", "r", "um", "u", "m", ""
```

Note that the subsets don't have to be substrings—for example, `"rm"` isn't a substring of `"rum"`.

P11.13 The following program generates all permutations of the numbers 0, 1, 2, ... , $n - 1$, without using recursion.

```
using namespace std;

void swap(int& x, int& y)
{
   int temp = x;
   x = y;
   y = temp;
}

void reverse(vector<int>& a, int i, int j)
{
   while (i < j)
   {
      swap(a[i], a[j]); i++; j--;
   }
}

bool next_permutation(vector<int>& a)
{
   for (int i = a.size() - 1; i > 0; i--)
   {
      if (a[i - 1] < a[i])
      {
         int j = a.size() - 1;
         while (a[i - 1] > a[j]) { j--; }
         swap(a[i - 1], a[j]);
         reverse(a, i, a.size() - 1);
         return true;
      }
   }
   return false;
}

void print(const vector<int>& a)
{
   for (int i = 0; i < a.size(); i++)
   {
      cout << a[i] << " ";
   }
   cout << endl;
}

int main()
{
   const int n = 4;
   vector<int> a(n);
   for (int i = 0; i < a.size(); i++)
   {
      a[i] = i;
   }
   print(a);
   while (next_permutation(a))
   {
      print(a);
   }
   return 0;
}
```

The algorithm uses the fact that the set to be permuted consists of distinct numbers. Thus, you cannot use the same algorithm to compute the permutations of the characters in a string. You can, however, use this technique to get all permutations of the character positions and then compute a string whose `i`th character is `s[a[i]]`. Use this approach to reimplement the `generate_permutations` function without recursion.

P11.14 Refine the `is_palindrome` function to work with arbitrary strings, by ignoring non-letter characters and the distinction between upper- and lowercase letters. For example, if the input string is

```
"Madam, I'm Adam!"
```

then you'd first strip off the last character because it isn't a letter, and recursively check whether the shorter string

```
"Madam, I'm Adam"
```

is a palindrome.

P11.15 *Towers of Hanoi*. This is a well-known puzzle. A stack of disks of decreasing size is to be transported from the left-most peg to the right-most peg. The middle peg can be used as a temporary storage. (See Figure 6.) One disk can be moved at one time, from any peg to any other peg. You can place smaller disks only on top of larger ones, not the other way around.

Write a program that prints the moves necessary to solve the puzzle for n disks. (Ask the user for n at the beginning of the program.) Print moves in the form

```
Move disk from peg 1 to peg 3
```

Hint: Write a helper function

```
void hanoi(int from, int to, int n)
```

that moves the top n disks from the peg `from` to the peg `to`. If n is 1, simply move the disk. If $n > 1$, first move the pile of the top $n - 1$ disks to the third peg, move the nth disk to the destination, and then move the pile from the third peg to the destination peg, this time using the original peg as the temporary storage.

Extra credit if you write the program to actually draw the moves using "ASCII art" or `ch06/animation.cpp`.

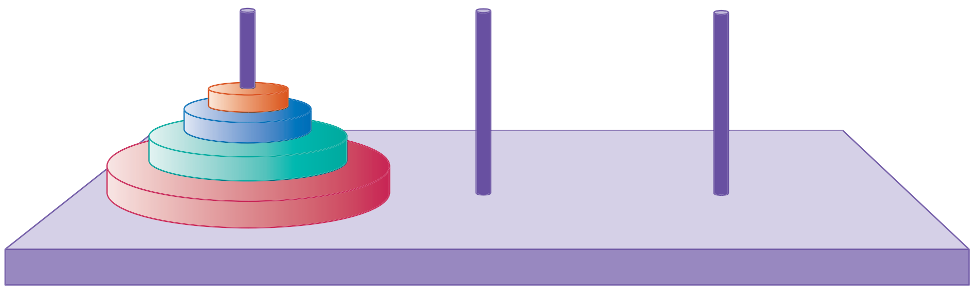


Figure 6 Towers of Hanoi

P11.16 *Escaping a Maze*. You are currently located inside a maze. The walls of the maze are indicated by asterisks (*).

```
* *******
*     * *
* ***** *
* * *   *
* * *** *
*   *   *
*** * * *
*     * *
******* *
```

Use the following recursive approach to check whether you can escape from the maze: If you are at an exit, return `true`. Recursively check whether you can escape from one of the empty neighboring locations without visiting the current location. This method merely tests whether there is a path out of the maze. Extra credit if you can print out a path that leads to an exit.

P11.17 Extend the expression evaluator in Section 11.6 so that it can handle the `%` operator as well as a "raise to a power" operator `^`. For example, `2 ^ 3` should evaluate to 8. As in mathematics, raising to a power should bind more strongly than multiplication: `5 * 2 ^ 3` is 40.

ANSWERS TO SELF-CHECK QUESTIONS

1. Suppose we omit the statement. When computing the area of a triangle with `side_length` 1, we compute the area of the triangle with `side_length` 0 as 0, and then add 1, to arrive at the correct area.
2. You would compute the smaller area recursively, then return `smaller_area + side_length + side_length - 1`.

```
[][][][]
[][][][]
[][][][]
[][][][]
```

Of course, it would be simpler to compute the area simply as `side_length * side_length`. The results are identical because

$$1 + 0 + 2 + 1 + 3 + 2 + \cdots + n + n - 1 = \frac{n(n+1)}{2} + \frac{(n-1)n}{2} = n^2.$$

3. There is no provision for stopping the recursion. When a number < 10 isn't 8, then the function should return `false` and stop.
4.
```
int pow2(int n)
{
   if (n <= 0) { return 1; } // 2⁰ is 1
   else { return 2 * pow2(n - 1); }
}
```
5.
```
mystery(4) calls mystery(3)
   mystery(3) calls mystery(2)
      mystery(2) calls mystery(1)
         mystery(1) calls mystery(0)
            mystery(0) returns 0.
         mystery(1) returns 0 + 1 * 1 = 1
      mystery(2) returns 1 + 2 * 2 = 5
   mystery(3) returns 5 + 3 * 3 = 14
mystery(4) returns 14 + 4 * 4 = 30
```
6. In this problem, *any* decomposition will work fine. We can remove the first or last character and then remove punctuation marks from the remainder. Or we can break the string in two substrings, and remove punctuation marks from each.

7. If the last character is a punctuation mark, then you simply return the shorter string with punctuation marks removed. Otherwise, you reattach the last character to that result and return it.

8. The simplest input is the empty string. It contains no punctuation marks, so you simply return it.

9.
```
if str is empty, return str.
last = last letter in str
simpler_result = remove_punctuation(str with last letter removed)
if (last is a punctuation mark)
    return simpler_result.
else
    return simpler_result + last.
```

10. When `start >= end`, that is, when the investigated string is either empty or has length 1.

11. A `sum_helper(int a[], int start, int size)`. The function calls `sum_helper(a, start + 1, size)`.

12. Call `sum(a, size - 1)` and add the *last* element, `a[size - 1]`.

13. The loop is slightly faster. Of course, it is even faster to simply compute `side_length * (side_length + 1) / 2`.

14. No, the recursive solution is about as efficient as the iterative approach. Both require $n - 1$ multiplications to compute $n!$.

15. The recursive algorithm performs about as well as the loop. Unlike the recursive Fibonacci algorithm, this algorithm doesn't call itself again on the same input. For example, the sum of the array 1 4 9 16 25 36 49 64 is computed as the sum of 1 4 9 16 and 25 36 49 64, then as the sums of 1 4, 9 16, 25 36, and 49 64, which can be computed directly.

16. They are `b` followed by the six permutations of `eat`, `e` followed by the six permutations of `bat`, `a` followed by the six permutations of `bet`, and `t` followed by the six permutations of `bea`.

17. Simply change `if (word.length() == 0)` to `if (word.length() <= 1)`, because a word with a single letter is also its sole permutation.

18. An iterative solution would have a loop whose body computes the next permutation from the previous ones. But there is no obvious mechanism for getting the next permutation. For example, if you already found permutations `eat`, `eta`, and `aet`, it is not clear how you use that information to get the next permutation. Actually, there is an ingenious mechanism for doing just that, but it is far from obvious—see Exercise P11.13.

19. Factors are combined by multiplicative operators (`*` and `/`), terms are combined by additive operators (`+`, `-`). We need both so that multiplication can bind more strongly than addition.

20. To handle parenthesized expressions, such as `2+3*(4+5)`. The subexpression `4+5` is handled by a recursive call to `expression_value`.

21. The inputs `3+4*` are processed normally. Then `factor_value` is called and returns a result of `0` when it finds no digits, and `term_value` returns `0` for the product. Then `expression_value` peeks at the next character, finds neither a `+` or `-`, and returns the result 3.