Chapter

10

Introducing Recursion

Performing a task repeatedly is iteration. Selecting different alternatives is selection. Most of us learn to use the control statements *for*, *while*, and *if* easily, because they are familiar. In this lesson, you'll look at a different, more abstract problem-solving strategy called recursion.

Recursion is a technique where large problems are solved by reducing them to smaller problems of the same form. This is different than procedural decomposition, where the smaller problems have a different structure. In recursion, the sub-problems have the same form as the original.

To most of us, this does not make much sense when we first hear it. Since it is unfamiliar, learning how to use recursion can be difficult. As a problem-solving tool, recursion is so powerful that it at times seems almost magical.

Recursion makes it possible to write complex programs in simple and elegant ways.

A Recursion Example

You may be familiar with the notation `n!`, pronounced "**n factorial**", the **product** of all of the positive integers less-than, or equal to `n`. In mathematical notation it is written like this.

$$\begin{aligned} n! &= \prod_{k=1}^{n} k \\ &= 1 \cdot 2 \cdot 3 \cdots (n-2) \cdot (n-1) \cdot n \\ &= n(n-1)(n-2) \cdots (2)(1) \end{aligned}$$

Using a loop, we can implement the function in C++: like this:

```
int factorial(int n)
{
    int result = 1;
    for (int i = 1; i <= n; ++i) { result *= i; }
    return result;
}
```

Algorithms that use loops like this are iterative.

A Recursive *factorial()* Function

Another way to the function is as a **recurrence relation**, which **recursively defines** a sequence; each further term is defined as a function of the preceding terms.

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

Without qualification, this is a **circular definition**. The **qualification** is that **`0!`** = **`1`**. We can translate this **recursive definition** into code as well:

```
int factorial(int n)
{
    if (n == 0) { return 1; }      // qualification
    return n * factorial(n - 1);   // recursion
}
```

The condition **`(n == 0)`** is the simplest condition, called the **base case**.

Thinking Recursively

The structure of a recursive problem looks like this:

```
If answer is known Then return it      // base case
Call the function with simpler inputs  // recursive case
Return the combined simpler results
```

This **pattern** is called the **recursive paradigm**. You can apply this technique as long as:

1. You can identify simple cases for which the answer is known.
2. You can find a **recursive decomposition** breaking any complex instance of the problem into simpler problems **of the same form**.

Because this depends on dividing complex problems into simpler instances of the same problem, such recursive solutions are often called **divide-and-conquer algorithms**.

The Recursive Leap of Faith

The computer treats recursive functions just like all other functions. It is useful to **put the underlying details aside** and focus on **a single level of the operation**; **assume** that any recursive call automatically gets the right answer as long as the arguments are in **some sense simpler** than the original.

This psychological strategy—assuming that any simpler recursive call will work correctly—is called **the recursive leap of faith**. Learning to apply this strategy is essential to using recursion in practical applications.

Consider what happens when you call `fact(4)`; the function must compute the expression `n * fact(n - 1)`, and by substituting the current value of `n` into the expression, you know that the result is `4 * fact(3)`.

Stop right there. Computing `fact(3)` is simpler than computing `fact(4)`. Because it is simpler, the recursive leap of faith allows you to **assume that it works**. Thus, you should assume that the call to `fact(3)` will correctly compute the value of `3!`, which is `3 × 2 × 1`, or `6`. The result of calling `fact(4)` is therefore `4 × 6`, or `24`.

As you look at the examples in the rest of this chapter, try to **focus on the big picture** instead of the details. Once you have made the recursive decomposition and identified the simple cases, be satisfied that the computer can handle the rest.

The Fibonacci Sequence

In 1202, the Italian mathematician Leonardo Fibonacci experimented with how a population of rabbits would grow from generation to generation, **give a set of rules**. His rules lead to a **sequence of terms**, which today are called the **Fibonacci sequence**: each term is the sum of the two numbers preceding it. Expressed as a recurrence relation:

$$t_n = t_{n-1} + t_{n-2}$$

This alone is not sufficient, however; you can define new terms, but **the process has to start somewhere!** You need **at least two terms already available**, which means that the first two terms in the sequence—t_0 and t_1—must be defined explicitly.

Given this qualification, the Fibonacci sequence is:

$$t_n = \begin{cases} n & \text{if } n \text{ is 0 or 1} \\ t_{n-1} + t_{n-2} & \text{otherwise} \end{cases}$$

A Recursive *fibonacci()*

To write a recursive implementation of a *fibonacci(n)* function, you need only plug in the simple cases, plus the recurrence relation, and you're done.

```
int fibonacci(int n)
{
    if (n < 2) { return n; }    // base case
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

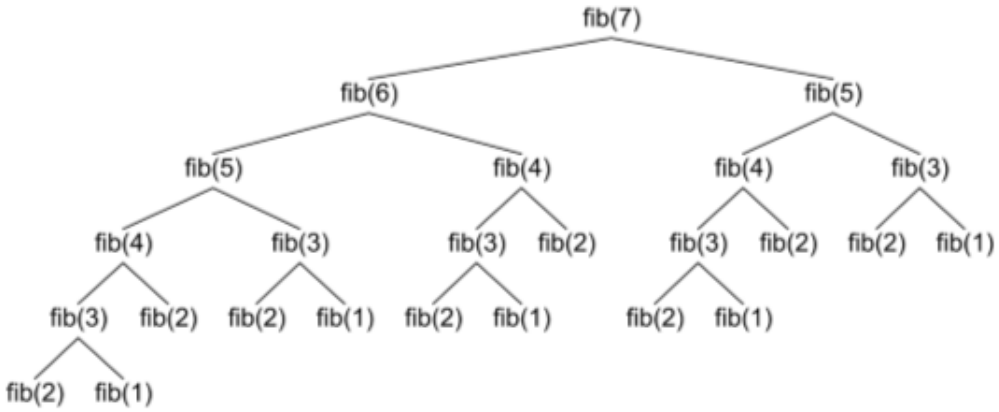
How do you convince yourself that the `fib()` function works? If you begin by tracing through the logic, I guarantee that you'll be confused. Instead, **regard this entire mechanism as irrelevant detail**.

Since the argument values are smaller, **each of these calls represents a simpler case**, and so, applying the recursive leap of faith, **you can assume that the program correctly computes each of these values**. **Case closed.** You don't need the details.

Introducing Recursive Efficiency

So, how **efficient** is the `fib()` function? How much memory does it take and how fast does it run? (Those are called the **space and time** measures of efficiency.) You can get a quick idea by simply calling `fib(42)`. It seems to take forever! Why?

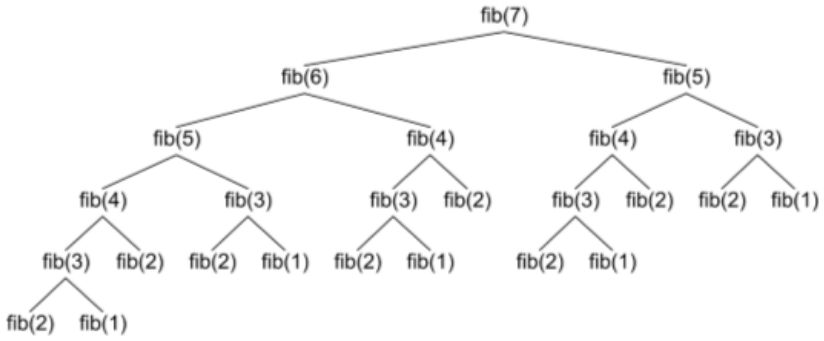
The `fib()` calculations are extremely inefficient, because the function makes many **redundant calls**, calculating **exactly the same term in the sequence** several times. Given that the Fibonacci sequence can be implemented quickly and efficiently using iteration, this is more than a little disturbing.



The problem here is **not recursion**, but the naïve way in which is implemented. In this case, we are repeatedly calculating the same value. By using a different strategy, we can write a recursive version of `fib()` where all of these redundant calls disappear. You'll learn how to do that in the next lesson.

Efficient Recursion

The naïve version of the recursive Fibonnaci you met in the last lesson was very inefficient. As the numbers get larger, it takes an increasingly large amount of time to generate each one.



This is because for each number we find, we have to generate **all** of the Fibonnaci numbers preceeding it. In Computer Science, we say that this implementation has an **exponential**, or ***O(2ⁿ)*** runtime performance; as n gets larger, we double the number of calcualations at each step. That means that it could **literally take years** to calculate a Fibonacci number of even a moderate size.

Wrappers and Helpers

We can reduce those years to a fraction of a second by learning about **wrappers** and **helpers**. A wrapper is a **non-recursive** function that **calls** a **recursive** function. A helper is the recursive function that the wrapper calls. Let's apply that to the Fibonacci sequence.

We can instantly calculate ***fib(n)*** when we know the values of ***fib(n-1)*** and ***fib(n-2)***. If you don't know the values of ***fib(n-1)*** and ***fib(n-2)***, that takes a lot of time, but when you do, then it's really fast. Are there values for ***fib(n-1)*** and ***fib(n-2)*** that we **do know**? Let's write out the sequence:

```
     n->0 1 2 3 4 5  6  7  8  9 10 . . .
fib(n)->0 1 1 2 3 5  8 13 21 34 55 . . .
```

Since ***fib(0)*** is ***0*** and ***fib(1)*** is ***1***, we can start there. For our **recursive helper**, just write a function that accepts ***n*** and the two terms: ***t0*** and ***t1***. If ***n*** is ***0*** return or ***t0*** and if ***1***, return ***t1***. Otherwise call the function recursively with ***n - 1***.

However, instead of passing the ***t0*** and ***t1*** to the recursive call, calculate the **next two terms** and pass those instead. Here's what the helper should look like:

```
int helper(int n, int t0, int t1)
{
    if (n == 0) return t0;
    if (n == 1) return t1;
    return helper(n - 1, t1, t0 + t1);
}
```

For the ***fib()*** **wrapper function**, just call the helper, kick-starting it with the first two terms, ***0*** and ***1***. Here's what the function looks like:

```
int fib(int n)
{
    return helper(n, 0, 1);
}
```

Checking Palindromes

A **palindrome** is a string that reads identically backward and forward, such as "level" or "noon". Although it is easy to check whether a string is a palindrome by **iterating** through its characters, palindromes can also be defined recursively.

Any palindrome longer than a single character must contain a shorter palindrome in its interior. For example, the string "`level`" consists of the palindrome "`eve`" with an "`l`" at each end. Thus, to check whether a string is a palindrome—assuming the string is sufficiently long that it does not constitute a simple case—all you need to do is

1. Check to see that the first and last characters are the same.
2. Check to see whether the substring generated by removing the first and last characters is itself a palindrome.

If both apply, then the string is a palindrome. So, **what are the simple or base-cases?** A single-character string is a palindrome because reversing a one-character string has no effect. The one-character string therefore represents a simple case, **but it is not the only one**. The empty string—which contains no characters at all—is also a palindrome.

Here is a recursive `isPalindrome()`. It returns `true` if its argument is a palindrome.

```
bool isPalindrome(const string& str)
{
    if (str.size() < 2) { return true; } // base cases
    return str.front() == str.back() &&
        isPalindrome(str.substr(1, str.size() - 2));
}
```

If the length of the string is less than 2, it is a palindrome. If not, the function checks to make sure that the string meets both of the necessary criteria.

Measuring Efficiency

Like `fibonacci()` this implementation is also **very inefficient**, even though it is easy to follow. You can improve the performance by making these changes:

- Calculate the size of the string only once.
- Don't make a new substring on each call.

The main inefficiency is the repeated `substr()` calls. You can avoid this by passing indices to keep track of the positions instead of creating new substrings.

```
bool palHelp(const string& str, int i1, int i2)
{
    if (i1 >= i2) { return true; }
    return str.at(i1) == str.at(i2) &&
        palHelp(str, i1 + 1, i2 - 1);
}

bool isPalindrom(const string& str)
{
    return palHelp(str, 0, str.size() - 1);
}
```

As with the `fib()` function, this `isPalindrome()` is a **recursive wrapper**. It "wraps up" and calls the **recursive helper function** `palHelper()` to do its work.

Mutual Recursion

Most recursive functions **call themselves directly.** However, the definition of recursion is broader. To be recursive, a function must call itself **at some point during its evaluation**. If a function `f()` calls a function `g()`, which in turn calls `f()`, those function calls are still considered to be recursive. Because the functions `f()` and `g()` call each other, this type of recursion is called **mutual recursion**.

As a simple example, it turns out to be easy—**although wildly inefficient**—to use recursion to test whether a number is even or odd:

```
bool isEven(unsigned n);    // prototype
bool isOdd(unsigned n) { return ! isEven(n); }
bool isEven(unsigned n)
{
    if (n == 0) return true;
    return isOdd(n - 1);
}
```

This code implements the **`isEven()`** and **`isOdd()`** functions, by taking advantage of the following informal definition:

- A number is even if its predecessor is odd.
- A number is odd if is not even.
- The number **`0`** is even by definition.

Even though these rules seem simple, they form the basis of an effective strategy for distinguishing odd and even numbers, as long as those numbers are nonnegative. We ensure that by having **`isEven()`** and **`isOdd()`** take arguments of type **`unsigned`**, which C++ uses to represent an integer that can never be less than zero.

Finish Up

- Complete the **reading exercises (REX)** for this chapter.
- Complete the homework using the **CS50 IDE**. The link is on Canvas.
  a. Make sure you submit the assignment using **`make submit`**.
  b. Make sure you check the CS150 Homework Console to see that your scores got reported, before the beginning of the next lecture.
- Take the **pre-class reading quiz** on Canvas. You have two attempts.

See you in class or on the Canvas discussion board.