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 similar to functional decomposition, yet different as well. In functional decomposition, 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.

The Factorial Function

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

Algorithms that use loops like this are **iterative**. Let's see how to write the function **recursively**.

A Recursive Example

Another way to think of the *factorial* 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(n)
{
    if (n == 0) { return 1; }       // qualification
    return n * factorial(n - 1);    // recursion
}
```

The condition **`(n == 0)`** is the simplest possible condition. It is called the **base case**. If **`n`** is not zero, then the function multiplies **`n`** times the result of *(n - 1)!*. It does this, by **calling itself** again to simplify the problem.

The solution to **any** recursive problem can be organized like this:

```
If the answer is known then return it      // the base case
If not, then
  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.

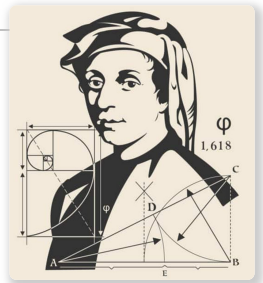
This psychological strategy—assuming that any simpler recursive call will work correctly—is called **the recursive leap of faith**. Learning to apply this strategy is essential to using recursion in practical applications.

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

Stop right there. Computing `factorial(3)` is **simpler** than computing `factorial(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 `factorial(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, base 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 can be expressed as:

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

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

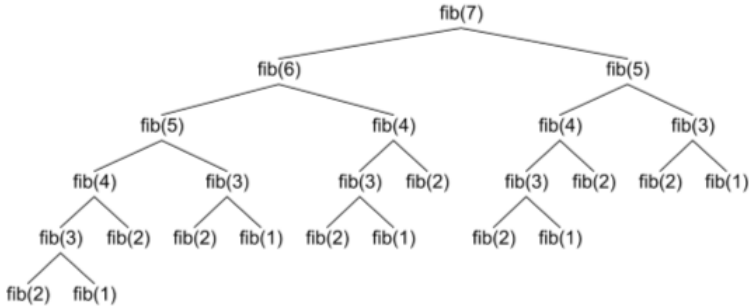
How do you convince yourself that the `fib()` function works? If you begin by tracing through the logic, I guarantee that you'll be confused. Instead, **regard this entire mechanism as irrelevant detail**.

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.

Recursive Efficiency

So, how efficient is the `fib()` function? What we mean by that is, how much memory does it use 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?**

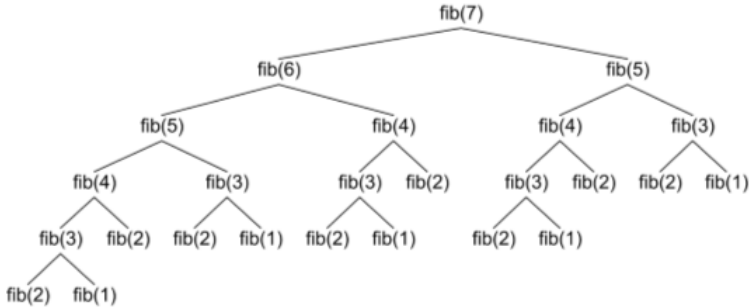
This particular recursive implementation of `fib()` is **extremely inefficient**, because the function makes many **redundant calls**, calculating **exactly the same term in the sequence** several times. Given that the Fibonacci sequence can be implemented quickly and efficiently using iteration, this 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 Fibonacci function which 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 Fibonacci numbers preceding it.



In Computer Science, we say that this implementation has an **exponential**, or $O(2^n)$ runtime performance; as n gets larger, we double the number of calculations at each step. That means that it could **literally take years** to calculate a Fibonacci number of even a moderate size using this function.

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.

Wrappers & Helpers

We can instantly calculate `fib(n)` when we know the values of `fib(n-1)` and `fib(n-2)`. When you don't know those values, the calculation 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 and see:

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 `t0` and if it is `1`, return `t1`. Otherwise call the function recursively with `n - 1`.

When calling the function recursively, however, instead of passing the `t0` and`t1`, 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 function which returns true when given a palindrome.

```
bool isPalindrome(const string& str)
{
    if (str.size() < 2) { return true; }     // base case
    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 first checks to see that the first and last characters are the same, and, if they are, it calls itself again with a shorter substring, removing the first and last characters from **`str`**.

Palindrome Efficiency

Like our original,naïve *fibonacci* function this implementation is also very inefficient, but for a different reason.

- The `fib()` function was inefficient because every time we calculated a term, we first had to (re)calculate all of the lower terms. It was expensive in terms of **time**.
- The `isPalindrome()` function is inefficient because every time we enter the recursive call, we first have to create a new substring, which not only takes time, but also uses extra memory. This function is expensive in terms of **space**.

We 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.

Of course, that means we'll need a **helper** and a **wrapper**. Here they are:

```
bool palHelper(const string&, int i1, int i2)
{
    if (i1 >= i2) { return true; }
    return str.at(i1) == str.at(i2) &&
        palHelper(str, i1 + 1, i2 - 1);
}
// Wrapper
bool isPalindrome(const string& str)
{
    return palHelper(str, 0, str.size() - 1);
}
```

A Recursion Checklist I

Now that you've seen several examples of recursion, let's apply recursion to a few problems, similar to those on a programming exam. To help you, here is a **checklist** that will help you identify the most common sources of errors**.**

☐ Check the Simple (Base) Cases

Does your recursive implementation begin by checking for simple base-cases? First check to see if the problem is so simple that no recursion is necessary. Recursive functions always start with the keyword `if`; if yours doesn't, look carefully.

☐ Have You Solved the Base Cases Correctly?

A surprising number of bugs in arise from **incorrect solutions to the simple base-cases**. If the simple cases are wrong, the rest of the function will inherit the same mistake. If you had defined `factorial(0)` as returning `0` instead of `1`, **any** argument would end up returning `0`.

☐ Does Decomposition Make it Simpler?

Does your recursive decomposition **make the problem simpler**? Problems must get simpler as you go along; the problem **must get smaller** as it proceeds. If the problem does not get simpler, you have the recursive analogue of the infinite loop, which is called **nonterminating or infinite recursion**.

A Recursion Checklist II

Here's the rest of the recursion checklist.

☐ Have You Covered All Possibilities?

Does the simplification process always reach the base case, or **have you left out some of the possibilities**? A common error is forgetting one of the base-cases. You need to check the empty string in the `isPalindrome()` function, as well as the single-character string. Since the function reduces the size of the string by two each time, only having the one-character base case, would mean some strings would fail.

☐ Are Sub-problems Identical in Form?

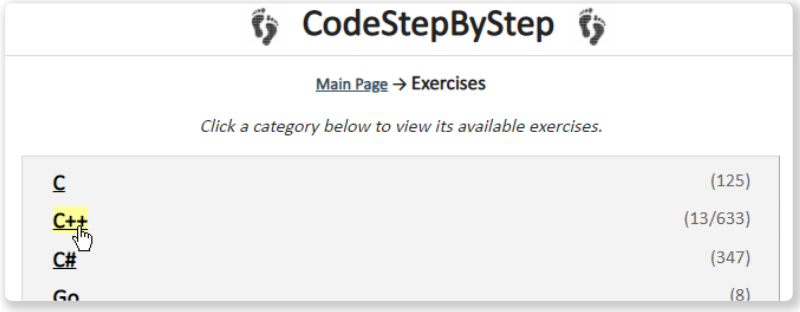
Are the recursive sub-problems **truly identical in form to the original?** It is essential that the sub-problems be of the same form. If the recursive calls change the nature of the problem then the entire process can break down.

☐ Are You a Believer?

When you apply the recursive leap of faith, do the solutions to the recursive sub-problems **provide a complete solution to the original problem**? Work through all the steps in the current function call, but assume that every recursive call returns the correct answer. If this process yields the right solution, your program should work.

Try It Out

On the Canvas course home page, you'll find a link to Code Step-by-Step, a Web site providing practice problems in several different programming languages, including C++. Go ahead and create your own account, and then let's walk through a problem.



Click the C++ link as shown in the screenshot above, and then find the **recursion** section. Here are the instructions for the first problem, `collapseSequences`.

> *Write a **recursive function** named `collapseSequences` that accepts a `string s` and `char c` as parameters and returns a new `string` that is the same as `s` but with any sequences of **consecutive occurrences** of `c` compressed into a single occurrence of `c`. For example, if you collapse sequences of character `'a'` in the `string "aabaaccaaaaada"`, you get `"abaccada"`.*
>
> *Your function is **case-sensitive**; if the character `c` is, for example, a lowercase `'f'`, your function should not collapse sequences of uppercase `'F'` characters. In other words, you do not need to write code to handle case issues in this problem.*

The following table shows two examples and their expected return values:

Call	Returns
collapseSequences("aabaaccaaaaaada", 'a')	"abaccada"
collapseSequences("mississsipppi", 's')	"misisipppi"

Solving the Problem I

Let's walk through the steps listed in the checklist. Normally, of course, you'll compress these steps together in an intuitive way. However, if you have difficulty with solving a recursive problem, going back and checking them individually is a good debugging tool.

☐ Check the Simple (Base) Cases

What is the simplest base case? In other words, what values for `s` require **no compression**? Well, obviously, if we don't have any characters, there can be no compression. Similarly, if we have only a single character, there can be no compression. The **simplest thing that can work, without recursion** is:

```
string collapseSequences(const string& s, char c)
{
    if (size() < 2) return s;  // base case
}
```

☐ Have You Solved the Base Cases Correctly?

Have we handled all of the base cases? The empty string returns `""`, and a single character returns that character. It doesn't matter if the character matches the parameter `c`, since a single-character cannot be a sequence. Those should be all of the base cases. If we have two charcters, we may have a sequence `"cc"` that requires compression.

☐ Does Decomposition Make it Simpler?

Does decomposition make it simpler? Or, in other words, how can we solve a simpler version of the problem? Or, how can we approach the base case? How? By **passing a smaller** string each time we call the function recursively.

Solving the Problem II

Let's continue with the remaining steps in the checklist.

☐ Have You Covered All Possibilities?

We have (**at least**) two characters, (because otherwise, we would have gone into the base case). In any event, we only need to think about the first two characters.

- They **may** both match `c`. If that is the case, then we need to **ignore** the first character, and pass only a shortened string to the function.
- They **may not** both match `c`. If so, them we must **include** the first character in the returned string before passing a shortened string to our function.

This should **cover all possibilities.**

☐ Are the Subproblems Identical in Form?

Depending on whether both characters match `c` or not, we do a different action in each case; in one we include the first character, and in other we do not, but the **form** of the problem **is the same in both cases.**

☐ Are You a Believer?

Walk through a solution using the simplest recursive cases. If it works for those, then **it must work for all of them**.

1. Call `collapseSequences("vv", 'v')`
    - Both characters match `c`, so the first `v` is **ignored** and the function is **called again** with the string `"v"`.
    - On the second recursive call, the base case returns `"v"`
    - Thus, the sequence `"vv"` is compressed to `"v"`, **so it works**.
2. Now call `collapseSequences("va", 'v')`
    - The two characters **do not match**, so the first `'v'` is returned along with the value returned on the recursive call.
    - The second recursive call returns `"a"` from the base case.
    - Thus, the sequence `"va"` **is not compressed** (even though one of the characters matched the character `c`). **So, it works.**

Implement the algorithm described here and you'll see that all of the tests should pass. Now you can try a few problems on your own.