Reading 10: Recursion

Software in 6.005

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

After today's class, you should:

- be able to decompose a recursive problem into recursive steps and base cases
- know when and how to use helper methods in recursion
- understand the advantages and disadvantages of recursion vs. iteration

Recursion

In today's class, we're going to talk about how to implement a method, once you already have a specification. We'll focus on one particular technique, *recursion* . Recursion is not appropriate for every problem, but it's an important tool in your software development toolbox, and one that many people scratch their heads over. We want you to be comfortable and competent with recursion, because you will encounter it over and over. (That's a joke, but it's also true.)

Since you've taken 6.01, recursion is not completely new to you, and you have seen and written recursive functions like factorial and fibonacci before. Today's class will delve more deeply into recursion than you may have gone before. Comfort with recursive implementations will be necessary for upcoming classes.

A recursive function is defined in terms of *base cases* and *recursive steps* .

- In a base case, we compute the result immediately given the inputs to the function call.
- In a recursive step, we compute the result with the help of one or more *recursive calls* to this same function, but with the inputs somehow reduced in size or complexity, closer to a base case.

Consider writing a function to compute factorial. We can define factorial in two different ways:

Product

$$n! = n \times (n-1) \times \cdots \times 2 \times 1 = \prod_{k=1}^{n} k$$

(where the empty product equals multiplicative identity *1*)

Recurrence relation

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

which leads to two different implementations:

Iterative

```
public static long factorial(int n) {
  long fact = 1;
  for (int i = 1; i <= n; i++) {
    fact = fact * i;
  }
  return fact;
}
```

Recursive

```
public static long factorial(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * factorial(n-1);
  }
}
```

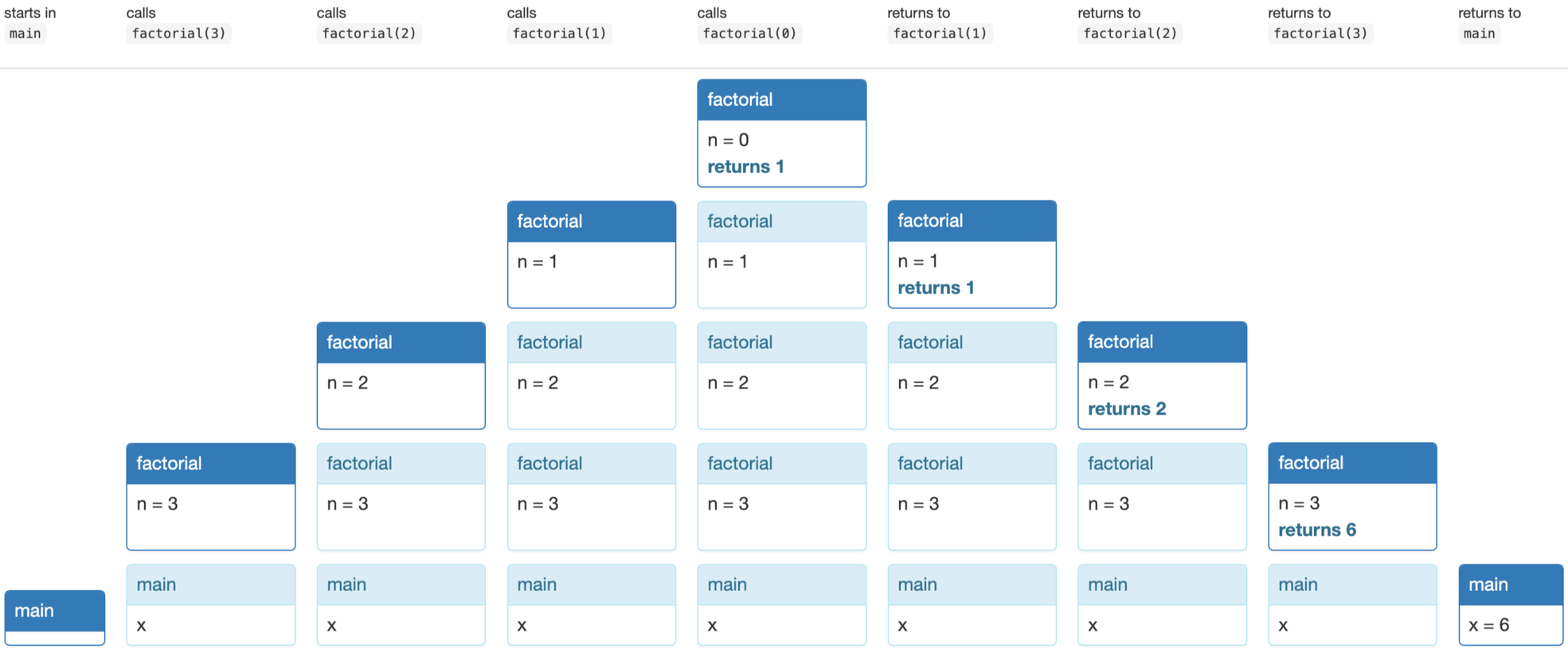
In the recursive implementation on the right, the base case is *n = 0* , where we compute and return the result immediately: *0!* is defined to be *1* . The recursive step is *n > 0* , where we compute the result with the help of a recursive call to obtain *(n-1)!* , then complete the computation by multiplying by *n* .

To visualize the execution of a recursive function, it is helpful to diagram the *call stack* of currently-executing functions as the computation proceeds.

Let's run the recursive implementation of `factorial` in a main method:

```
public static void main(String[] args) {
    long x = factorial(3);
}
```

At each step, with time moving left to right:



In the diagram, we can see how the stack grows as `main` calls `factorial` and `factorial` then calls *itself* , until `factorial(0)` does not make a recursive call. Then the call stack unwinds, each call to `factorial` returning its answer to the caller, until `factorial(3)` returns to `main` .

Here's an **interactive visualization of `factorial`** . You can step through the computation to see the recursion in action. New stack frames grow down instead of up in this visualization.

You've probably seen factorial before, because it's a common example for recursive functions. Another common example is the Fibonacci series:

```
/**
 * @param n >= 0
 * @return the nth Fibonacci number
 */
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1; // base cases
    } else {
        return fibonacci(n-1) + fibonacci(n-2); // recursive step
    }
}
```

Fibonacci is interesting because it has multiple base cases: n=0 and n=1. You can look at an **interactive visualization of Fibonacci** . Notice that where factorial's stack steadily grows to a maximum depth and then shrinks back to the answer, Fibonacci's stack grows and shrinks repeatedly over the course of the computation.

Choosing the Right Decomposition for a Problem

Finding the right way to decompose a problem, such as a method implementation, is important. Good decompositions are simple, short, easy to understand, safe from bugs, and ready for change.

Recursion is an elegant and simple decomposition for some problems. Suppose we want to implement this specification:

```
/**
 * @param word consisting only of letters A-Z or a-z
 * @return all subsequences of word, separated by commas,
 * where a subsequence is a string of letters found in word
 * in the same order that they appear in word.
 */
public static String subsequences(String word)
```

For example, `subsequences("abc")` might return `"abc,ab,bc,ac,a,b,c,"` . Note the trailing comma preceding the empty subsequence, which is also a valid subsequence.

This problem lends itself to an elegant recursive decomposition. Take the first letter of the word. We can form one set of subsequences that *include* that letter, and another set of subsequences that exclude that letter, and those two sets completely cover the set of possible subsequences.

```
1 public static String subsequences(String word) {
2     if (word.isEmpty()) {
3         return ""; // base case
4     } else {
5         char firstLetter = word.charAt(0);
6         String restOfWord = word.substring(1);
7 
8         String subsequencesOfRest = subsequences(restOfWord);
9 
10        String result = "";
11        for (String subsequence : subsequencesOfRest.split(",", -1)) {
12            result += "," + subsequence;
13            result += "," + firstLetter + subsequence;
14        }
15        result = result.substring(1); // remove extra leading comma
16        return result;
17    }
18 }
```

Structure of Recursive Implementations

A recursive implementation always has two parts:

- **base case** , which is the simplest, smallest instance of the problem, that can't be decomposed any further. Base cases often correspond to emptiness – the empty string, the empty list, the empty set, the empty tree, zero, etc.
- **recursive step** , which **decomposes** a larger instance of the problem into one or more simpler or smaller instances that can be solved by recursive calls, and then **recombines** the results of those subproblems to produce the solution to the original problem.

It's important for the recursive step to transform the problem instance into something smaller, otherwise the recursion may never end. If every recursive step shrinks the problem, and the base case lies at the bottom, then the recursion is guaranteed to be finite.

A recursive implementation may have more than one base case, or more than one recursive step. For example, the Fibonacci function has two base cases, n=0 and n=1.

Helper Methods

The recursive implementation we just saw for `subsequences()` is one possible recursive decomposition of the problem. We took a solution to a subproblem – the subsequences of the remainder of the string after removing the first character – and used it to construct solutions to the original problem, by taking each subsequence and adding the first character or omitting it. This is in a sense a *direct* recursive implementation, where we are using the existing specification of the recursive method to solve the subproblems.

In some cases, it's useful to require a stronger (or different) specification for the recursive steps, to make the recursive decomposition simpler or more elegant. In this case, what if we built up a partial subsequence using the initial letters of the word, and used the recursive calls to *complete* that partial subsequence using the remaining letters of the word? For example, suppose the original word is "orange". We'll both select "o" to be in the partial subsequence, and recursively extend it with all subsequences of "range"; and we'll skip "o", use "" as the partial subsequence, and again recursively extend it with all subsequences of "range".

Using this approach, our code now looks much simpler:

```
/**
 * Return all subsequences of word (as defined above) separated by commas,
 * with partialSubsequence prepended to each one.
 */
private static String subsequencesAfter(String partialSubsequence, String word) {
    if (word.isEmpty()) {
        // base case
        return partialSubsequence;
    } else {
        // recursive step
        return subsequencesAfter(partialSubsequence, word.substring(1))
             + ","
             + subsequencesAfter(partialSubsequence + word.charAt(0), word.substring(1));
    }
}
```

This `subsequencesAfter` method is called a **helper method** . It satisfies a different spec from the original `subsequences` , because it has a new parameter `partialSubsequence` . This parameter fills a similar role that a local variable would in an iterative implementation. It holds temporary state during the evolution of the computation. The recursive calls steadily extend this partial subsequence, selecting or ignoring each letter in the word, until finally reaching the end of the word (the base case), at which point the partial subsequence is returned as the only result. Then the recursion backtracks and fills in other possible subsequences.

To finish the implementation, we need to implement the original `subsequences` spec, which gets the ball rolling by calling the helper method with an initial value for the partial subsequence parameter:

```
public static String subsequences(String word) {
    return subsequencesAfter("", word);
}
```

Don't expose the helper method to your clients. Your decision to decompose the recursion this way instead of another way is entirely implementation-specific. In particular, if you discover that you need temporary variables like `partialSubsequence` in your recursion, don't change the original spec of your method, and don't force your clients to correctly initialize those parameters. That exposes your implementation to the client and reduces your ability to change it in the future. Use a private helper function for the recursion, and have your public method call it with the correct initializations, as shown above.

Choosing the Right Recursive Subproblem

Let's look at another example. Suppose we want to convert an integer to a string representation with a given base, following this spec:

```
/**
 * @param n integer to convert to string
 * @param base base for the representation. Requires 2<=base<=10.
 * @return n represented as a string of digits in the specified base, with
 *           a minus sign if n<0.
 */
public static String stringValue(int n, int base)
```

For example, `stringValue(16, 10)` should return `"16"`, and `stringValue(16, 2)` should return `"10000"`.

Let's develop a recursive implementation of this method. One recursive step here is straightforward: we can handle negative integers simply by recursively calling for the representation of the corresponding positive integer:

```
if (n < 0) return "-" + stringValue(-n, base);
```

This shows that the recursive subproblem can be smaller or simpler in more subtle ways than just the value of a numeric parameter or the size of a string or list parameter. We have still effectively reduced the problem by reducing it to positive integers.

The next question is, given that we have a positive n, say n=829 in base 10, how should we decompose it into a recursive subproblem? Thinking about the number as we would write it down on paper, we could either start with 8 (the leftmost or highest-order digit), or 9 (the rightmost, lower-order digit). Starting at the left end seems natural, because that's the direction we write, but it's harder in this case, because we would need to first find the number of digits in the number to figure out how to extract the leftmost digit. Instead, a better way to decompose n is to take its remainder modulo base (which gives the *rightmost* digit) and also divide by base (which gives the subproblem, the remaining higher-order digits):

```
return stringValue(n/base, base) + "0123456789".charAt(n%base);
```

Think about several ways to break down the problem, and try to write the recursive steps. You want to find the one that produces the simplest, most natural recursive step.

It remains to figure out what the base case is, and include an if statement that distinguishes the base case from this recursive step.

Recursive Problems vs. Recursive Data

The examples we've seen so far have been cases where the problem structure lends itself naturally to a recursive definition. Factorial is easy to define in terms of smaller subproblems. Having a *recursive problem* like this is one cue that you should pull a recursive solution out of your toolbox.

Another cue is when the data you are operating on is inherently recursive in structure. We'll see many examples of recursive data a few classes from now, but for now let's look at the recursive data found in every laptop computer: its filesystem. A filesystem consists of named *files* . Some files are *folders* , which can contain other files. So a filesystem is recursive: folders contain other folders which contain other folders, until finally at the bottom of the recursion are plain (non-folder) files.

The Java library represents the file system using `java.io.File` . This is a recursive data type, in the sense that `f.getParentFile()` returns the parent folder of a file `f` , which is a `File` object as well, and `f.listFiles()` returns the files contained by `f` , which is an array of other `File` objects.

For recursive data, it's natural to write recursive implementations:

```
/**
 * @param f a file in the filesystem
 * @return the full pathname of f from the root of the filesystem
 */
public static String fullPathname(File f) {
    if (f.getParentFile() == null) {
        // base case: f is at the root of the filesystem
        return f.getName();
    } else {
        // recursive step
        return fullPathname(f.getParentFile()) + "/" + f.getName();
    }
}
```

Recent versions of Java have added a new API, `java.nio.Files` and `java.nio.Path` , which offer a cleaner separation between the filesystem and the pathnames used to name files in it. But the data structure is still fundamentally recursive.

Reentrant Code

Recursion – a method calling itself – is a special case of a general phenomenon in programming called **reentrancy** . Reentrant code can be safely re-entered, meaning that it can be called again *even while a call to it is underway.* Reentrant code keeps its state entirely in parameters and local variables, and doesn't use static variables or global variables, and doesn't share aliases to mutable objects with other parts of the program, or other calls to itself.

Direct recursion is one way that reentrancy can happen. We've seen many examples of that during this reading. The `factorial()` method is designed so that `factorial(n-1)` can be called even though `factorial(n)` hasn't yet finished working.

Mutual recursion between two or more functions is another way this can happen – A calls B, which calls A again. Direct mutual recursion is virtually always intentional and designed by the programmer. But unexpected mutual recursion can lead to bugs.

When we talk about concurrency later in the course, reentrancy will come up again, since in a concurrent program, a method may be called at the same time by different parts of the program that are running concurrently.

It's good to design your code to be reentrant as much as possible. Reentrant code is safer from bugs and can be used in more situations, like concurrency, callbacks, or mutual recursion.

When to Use Recursion Rather Than Iteration

We've seen two common reasons for using recursion:

- The problem is naturally recursive (e.g. Fibonacci)
- The data is naturally recursive (e.g. filesystem)

Another reason to use recursion is to take more advantage of immutability. In an ideal recursive implementation, all variables are final, all data is immutable, and the recursive methods are all pure functions in the sense that they do not mutate anything. The behavior of a method can be understood simply as a relationship between its parameters and its return value, with no side effects on any other part of the program. This kind of paradigm is called *functional programming* , and it is far easier to reason about than *imperative programming* with loops and variables.

In iterative implementations, by contrast, you inevitably have non-final variables or mutable objects that are modified during the course of the iteration. Reasoning about the program then requires thinking about snapshots of the program state at various points in time, rather than thinking about pure input/output behavior.

One downside of recursion is that it may take more space than an iterative solution. Building up a stack of recursive calls consumes memory temporarily, and the stack is limited in size, which may become a limit on the size of the problem that your recursive implementation can solve.

Common Mistakes in Recursive Implementations

Here are two common ways that a recursive implementation can go wrong:

- The base case is missing entirely, or the problem needs more than one base case but not all the base cases are covered.
- The recursive step doesn't reduce to a smaller subproblem, so the recursion doesn't converge.

Look for these when you're debugging.

On the bright side, what would be an infinite loop in an iterative implementation usually becomes a `StackOverflowError` in a recursive implementation. A buggy recursive program fails faster.

Summary

We saw these ideas:

- recursive problems and recursive data
- comparing alternative decompositions of a recursive problem
- using helper methods to strengthen a recursive step
- recursion vs. iteration

The topics of today's reading connect to our three key properties of good software as follows:

- **Safe from bugs.** Recursive code is simpler and often uses immutable variables and immutable objects.
- **Easy to understand.** Recursive implementations for naturally recursive problems and recursive data are often shorter and easier to understand than iterative solutions.
- **Ready for change.** Recursive code is also naturally reentrant, which makes it safer from bugs and ready to use in more situations.