Reading 14: Recursion
Recursion
Choosing the right decomposition for a problem
Structure of recursive implementati…
Helper methods
Choosing the right recursive subproblem
Recursive problems vs. recursive data
Mutual recursion
Reentrant code
When to use recursion rather than iteration
Common mistakes in recursive implementati…
Summary
More practice

Reading 14: Recursion

Software in 6.031

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.)

Recursion should not be new to you, and you should have seen and written recursive functions 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	Recurrence relation
$n! = n \times (n-1) \times \cdots \times 2 \times 1 = \prod_{k=1}^{n} k$ (where the empty product equals multiplicative identity *1*)	$n! = \begin{cases} 1 & \text{if } n = 0 \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$

which leads to two different implementations:

Iterative	Recursive

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

```
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*.

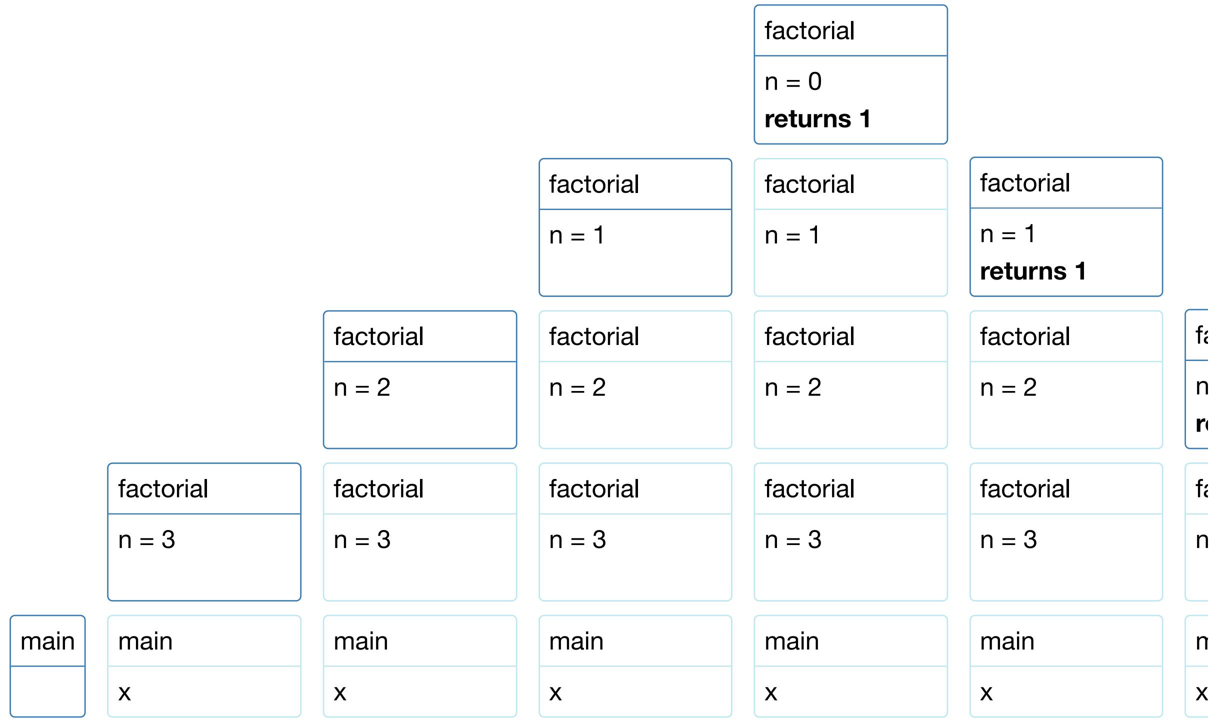
To visualize the execution of a recursive function, it is helpful to diagram the *call stack* of currently-executing functions as the computation proceeds.

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:

starts in `main`	calls `factorial(3)`	calls `factorial(2)`	calls `factorial(1)`	calls `factorial(0)`	returns to `factorial(1)`	re… `f…`



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); // recursi
ve 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.

READING EXERCISES

You are not logged in.

Recursive factorial

Consider this recursive implementation of the factorial function.

```
public static long factorial(int n) {
    if (n == 0) {
        return 1; // this is called the base case
    } else {
        return n * factorial(n-1); // this is the rec
ursive step
    }
}
```

For `factorial(3)`, how many times will the base case `return 1` be executed?

✔
- ○ 0 times
- ◉ 1 time ☑
- ○ 2 times
- ○ 3 times
- ○ more than 3 times

› `factorial(3)` calls `factorial(2)`, which calls `factorial(1)`, which calls `factorial(0)`, which executes the base case once. Each of the recursive calls then returns, without any further recursion.

CHECK EXPLAIN

Recursive Fibonacci

Consider this recursive implementation of the Fibonacci sequence.

```
public static int fibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1; // base cases
    } else {
        return fibonacci(n-1) + fibonacci(n-2); // re
cursive step
    }
}
```

For `fibonacci(3)`, how many times will the base case `return 1` be executed?

- ✔ ○ 0 times
- ○ 1 time
- ○ 2 times
- ◉ 3 times ☑
- ○ more than 3 times

`fibonacci(3)` calls `fibonacci(2)`. `fibonacci(2)` calls `fibonacci(1)`, which executes the base case once, and then calls `fibonacci(0)`, which executes the base case again. `fibonacci(2)` then returns to `fibonacci(3)`, which now calls `fibonacci(1)`, which executes the base case one more time, for a total of 3.

Another way to do it: because fibonacci is a pure function, with no side-effects, we can substitute expressions:

```
fibonacci(3) = fibonacci(2) + fibonacci(1)
             = (fibonacci(1) + fibonacci(0)) + fi
bonacci(1)
             reduces to (1 + 1) + 1 = 3 base case
s
```

Still another way to check, which only works for Fibonacci, is to realize that each execution of the base case contributes 1 to the final answer, and none of the recursive steps contribute any additional value. So the number of base case executions in `fibonacci(n)` is the same as the nth Fibonacci number.

CHECK EXPLAIN

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 wor
d
 * 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        final int withTrailingEmptyStrings = -1; // see String.split() spec
12        for (String subsequence : subsequencesOfRest.split(",", withTrailingEmptyStrings)) {
13            result += "," + subsequence;
14            result += "," + firstLetter + subsequence;
15        }
16        result = result.substring(1); // remove extra leading comma
17        return result;
18    }
19 }
```

READING EXERCISES

subsequences("c")

What does `subsequences("c")` return?

- `"c"`
- `""`
- `",c"` ✔
- `"c,"`

`subsequences("c")` first calls `subsequences("")`, which just returns `""`.

We then split this empty string on `","`, which returns an array of one element, the empty string.

The for loop then iterates over this array and constructs the two ways that subsequences of `"c"` can be formed (with or without the letter 'c'). It appends each new subsequence to result, starting it with a comma. This means we'll end up with an extra comma at the beginning of result, which we have to remove after the for loop.

- `result = ""` (line 10)
- `result = ","` (line 13)
- `result = ",,c"` (line 14)
- `result = ",c"` (line 16)

Finally `subsequences("c")` returns `",c"` representing the empty string and the string `"c"`, the two possible subsequences of the one-character string `"c"`.

CHECK EXPLAIN

subsequences("gc")

What does `subsequences("gc")` return?

- ○ `"g,c"`
- ◉ `",g,c,gc"` ✓
- ○ `",gc,g,c"`
- ○ `"g,c,gc"`

`subsequences("gc")` first calls `subsequences("c")`, which returns `",c"` as we saw in the previous question.

We then split this string on `","`, which produces an array of two elements, `""` and `"c"`.

The for loop then iterates over this array, producing two new subsequences from each element:

- `result = ""` (line 10)
- `result = ","` (line 13)
- `result = ",,g"` (line 14)
- `result = ",,g,c"` (line 13)
- `result = ",,g,c,gc"` (line 14)
- `result = ",g,c,gc"` (line 16)

This final result is returned from `subsequences("gc")`.

CHECK EXPLAIN

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*.

READING EXERCISES

Recursive structure

Recursive methods have a base case and a recursive step. What other concepts from computer science also have (the equivalent of) a base case and a recursive step?

- [x] proof by induction
- [ ] regression testing
- [ ] recessive functions
- [x] binary trees

In proof by induction, the base case is called a base case, and the recursive step is often called the inductive step.

Binary trees have leaves (base case) and internal nodes (recursive step).

Regression testing doesn't involve a base case or recursive step in general, and "recessive functions" aren't a concept in computer science.

CHECK EXPLAIN

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:

```
/**
 * @param partialSubsequence  a subsequence-in-progress, c
onsisting only of letters A-Z or a-z
 * @param word consisting only of letters A-Z or a-z
 * @return all subsequences of word, separated by commas,
with partialSubsequence prefixed to each one
 */
private static String subsequencesAfter(String partialSubs
equence, String word) {
    if (word.isEmpty()) {
        // base case
        return partialSubsequence;
    } else {
        // recursive step
        return subsequencesAfter(partialSubsequence, word.
substring(1))
             + ","
             + subsequencesAfter(partialSubsequence + wor
d.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.

READING EXERCISES

Unhelpful 1

Louis Reasoner doesn't want to use a helper method, so he tries to implement `subsequences()` by storing `partialSubsequence` as a static variable instead of a parameter. Here is his implementation:

```
private static String partialSubsequence = "";
public static String subsequencesLouis(String word) {
    if (word.isEmpty()) {
        // base case
        return partialSubsequence;
    } else {
        // recursive step
        String withoutFirstLetter = subsequencesLoui
s(word.substring(1));
        partialSubsequence += word.charAt(0);
        String withFirstLetter = subsequencesLouis(wo
rd.substring(1));
        return withoutFirstLetter + "," + withFirstLe
tter;
    }
}
```

Suppose we call `subsequencesLouis("c")` followed by `subsequencesLouis("a")`.

What does `subsequencesLouis("c")` return?

- ○ `"c"`
- ○ `""`
- ◉ `",c"` ✓
- ○ `"c,"`

What does `subsequencesLouis("a")` return?

- ○ `"a"`
- ○ `""`
- ○ `",a"`
- ○ `"a,"`
- ◉ `"c,ca"` ✓

The static variable maintains its value across calls to `subsequencesLouis()`, so it still has the final value `"c"` from the call to `subsequencesLouis("c")` when `subsequencesLouis("a")` starts. As a result, every subsequence of that second call will have an extra `c` before it.

CHECK EXPLAIN

Unhelpful 2

Louis fixes that problem by making `partialSubsequence` public:

```
/**
 * Requires: caller must set partialSubsequence to ""
before calling subsequencesLouis().
 */
public static String partialSubsequence;
```

Alyssa P. Hacker throws up her hands when she sees what Louis did. Which of these statements are true about his code?

- ☐ `partialSubsequence` is risky – it should be final
- ☑ `partialSubsequence` is risky – it is a global variable ✓
- ☐ `partialSubsequence` is risky – it points to a mutable object

> `partialSubsequence` is indeed a global variable. It can't be made final, however, because the recursion needs to reassign it (frequently). But at least it doesn't point to a mutable object.

CHECK EXPLAIN

Unhelpful 3

Louis gives in to Alyssa's strenuous arguments, hides his static variable again, and takes care of initializing it properly before starting the recursion:

```
public static String subsequences(String word) {
    partialSubsequence = "";
    return subsequencesLouis(word);
}

private static String partialSubsequence = "";

public static String subsequencesLouis(String word) {
    if (word.isEmpty()) {
        // base case
        return partialSubsequence;
    } else {
        // recursive step
        String withoutFirstLetter = subsequencesLoui
s(word.substring(1));
        partialSubsequence += word.charAt(0);
        String withFirstLetter = subsequencesLouis(wo
rd.substring(1));
        return withoutFirstLetter + "," + withFirstLe
tter;
    }
}
```

Unfortunately a static variable is simply a bad idea in recursion. Louis's solution is still broken. To illustrate, let's trace through the call `subsequences("xy")`. You can step through an **interactive visualization of this version** to see what happens. It will produce these recursive calls to `subsequencesLouis()`:

```
1. subsequencesLouis("xy")
2.     subsequencesLouis("y")
3.         subsequencesLouis("")
4.         subsequencesLouis("")
5.     subsequencesLouis("y")
6.         subsequencesLouis("")
7.         subsequencesLouis("")
```

When each of these calls **starts**, what is the value of the static variable partialSubsequence?

1. `subsequencesLouis("xy")`

   ✔ ~~empty string~~ empty string

2. `subsequencesLouis("y")`

   ✔ empty string empty string

3. `subsequencesLouis("")`

✔ empty string empty string

4. `subsequencesLouis("")`

   ✔ y y

5. `subsequencesLouis("y")`

   ✔ yx yx

6. `subsequencesLouis("")`

   ✔ yx yx

7. `subsequencesLouis("")`

   ✔ yxy yxy

Everything seems fine until call 5, where it becomes clear that the static variable is still clinging to letters like `"y"` that were added to it in deeper levels of recursion and never discarded.

The final (wrong) return value of this implementation can be read off from the base cases, calls 3,4,6,7: `",y,yx,yxy"`.

Static variables and aliases to mutable data are very unsafe for recursion, and lead to insidious bugs like this. When you're implementing recursion, the safest course is to pass in the values it needs, and stick to immutable objects or avoid mutation.

CHECK EXPLAIN

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<=ba
se<=10.
 * @return n represented as a string of digits in the spec
ified base, with
 *           a minus sign if n<0. No unnecessary leading z
eros are included.
 */
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.

READING EXERCISES

Implementing stringValue

Here is the recursive implementation of `stringValue()` with the recursive steps brought together but with the base case still missing:

```
/**
 * @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.  No unnecessary leading zeros are included.
 */
public static String stringValue(int n, int base) {
    if (n < 0) {
        return "-" + stringValue(-n, base);
    } else if (BASE CONDITION) {
        BASE CASE
    } else {
        return stringValue(n/base, base) + "0123456789".charAt(n%base);
    }
}
```

Which of the following can be substituted for the `BASE CONDITION` and `BASE CASE` to make the code correct? It may help to think about the test cases on the right.

```
stringValue(16, 10) → "16"
stringValue(3, 10)  → "3"
stringValue(0, 10)  → "0"
```

✔
- [ ] `else if (n == 0) { return "0"; }`
- [x] `else if (n < base) { return "" + n; }` ☑
- [ ] `else if (n == 0) { return ""; }`
- [x] `else if (n < base) { return "0123456789".substring(n,n+1); }`

The first choice doesn't work because it will add a leading 0 to single-digit numbers, i.e. making `stringValue(3, 10)` return `"03"` instead of just `"3"`.

The second choice works. `return "" + n` is shorthand for converting the single-digit number `n` into a string.

The third choice doesn't work because `stringValue(0, 10)` will return `""` instead of `"0"`.

The fourth choice works. Since `0 <= n < base` (because of the `if` statements), and `base <= 10` (by the precondition), the substring will not be out of bounds.

CHECK EXPLAIN

Calling stringValue

Assuming the code is completed with one of the base cases identified in the previous problem, what does `stringValue(170, 16)` do?

- returns `"AA"`
- returns `"170"`
- returns `"1010"`
- throws `StringIndexOutOfBoundsException`
- doesn't compile, static error
- `StackOverflowError`
- infinite loop

Note first that using base=16 violates the precondition of this method, so it doesn't have to satisfy the postcondition. A valid implementation can do anything. The question is what this particular valid implementation will do.

The recursive step will be invoked, which will split the number 170 by computing 170/16=10 and 170%16=10. The `charAt()` call will attempt to get the 11th character of `"0123456789"`, which is past the end of the string. A `StringIndexOutOfBoundsException` will result.

CHECK EXPLAIN

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 fil
esystem
 */
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.g
etName();
    }
}
```

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.

Mutual recursion

Sometimes multiple helper methods cooperate in a recursive implementation. If method A calls method B, which then calls method A again, then A and B are *mutually recursive*.

Mutual recursion is often found in code that operates over recursive data. Using the filesystem as an example, here are two mutually recursive methods that walk down the tree of files and folders:

```
/**
 * @param file a file in the filesystem
 */
public static void visitNode(File file) {
  if (file.isDirectory()) {
    visitChildren(file.listFiles());
  }
}

/**
 * @param files a list of files
 */
public static void visitChildren(File[] files) {
  for (File file : files) {
    visitNode(file);
  }
}
```

One of the advantages of this approach is that both methods are simple and short, and the client can choose to start the tree traversal from either single point (`visitNode`) or from multiple points (`visitChildren`) without needing any redundant code.

We will see many examples of mutually recursive methods when we talk about recursive data types in a few classes.

READING EXERCISES

Finding the files

Suppose you want to change the `visitNode` / `visitChildren` traversal so that it prints every visited file or folder that starts with a particular pattern. Here is the line of code:

```
if (file.getName().startsWith(pattern)) { System.out.println(file); }
```

Which places should this code be added?

- [x] as the first line of `visitNode`
- [ ] as the first line of the `if` statement body
- [ ] as the first line of `visitChildren`
- [ ] as the first line of the `for` loop body

To keep the code DRY (and avoid redundant printing), this code only needs to go at the start of `visitNode`. The other places would be either insufficient or redundant.

Where should the `pattern` variable be declared?

- [ ] as a static variable outside both methods
- [x] as a parameter of `visitNode`
- [x] as a parameter of `visitChildren`

The information about the pattern being searched for should be passed as a parameter to both `visit` methods. The methods then pass it back and forth as they mutually recurse.

Using a parameter rather than a static variable follows the general principle of scope minimization. Although it may be tempting to put it in a static variable, for apparent convenience and efficiency, that approach is very unsafe from bugs, because it means that these methods would no longer be reentrant (as discussed in the next section). Two different parts of the program would be unable to use `visitNode` / `visitChildren` at the same time without stomping on each other through the single shared `pattern` variable.

After these changes, the code looks like this:

```
/**
 * Prints files or folders whose names start with
pattern,
 * searching recursively from a root file or fold
er.
 * @param file a file in the filesystem
 * @param pattern pattern to look for
 */
public static void visitNode(File file, String pa
ttern) {
  if (file.getName().startsWith(pattern)) { Syste
m.out.println(file); }
  if (file.isDirectory()) {
    visitChildren(file.listFiles(), pattern);
  }
}

/**
 * Prints files or folders whose names start with
pattern,
 * searching recursively from a list of root file
s or folders.
 * @param files a list of files
 * @param pattern pattern to look for
 */
public static void visitChildren(File[] files, St
ring pattern) {
  for (File file : files) {
    visitNode(file, pattern);
  }
}
```

CHECK EXPLAIN

Accumulating results

Instead of printing the matching files, it would be better to return them, perhaps by gathering them up into a `Set`.

This exercise considers one way to do this, using a single mutable `Set` object that accumulates the results. This is analogous to what you would do if this file traversal were done iteratively in a loop. The next exercise will consider doing it with immutable `Set` values instead, since immutability is generally a better pattern for safe and understandable recursion.

How should the code be changed to accumulate the results in a mutable `Set`?

First declare `resultSet` as a:

✔
- [ ] static variable
- [x] parameter of `visitNode`
- [x] parameter of `visitChildren`
- [x] `Set<File>`
- [ ] `HashSet<File>`

The mutable `resultSet` can be passed as a parameter, minimizing its scope for the same reason as the previous exercise.

It should be declared as a `Set` rather than `HashSet` to allow the client the freedom to choose the kind of set implementation.

Replace `System.out.println(file);` with the following line(s):

✔
- [ ] `resultSet = new HashSet<>();`
- [x] `resultSet.add(file);`
- [ ] `resultSet.remove(file);`
- [ ] `resultSet.clear();`

We don't want to empty out the set during the recursion, only add new matching files we find.

Finally:

✔
- [ ] change the return type of both `visitNode` and `visitChildren` from `void` to `Set<File>`
- [x] change the calls to `visitNode` and `visitChildren` to pass in `resultSet`
- [x] document the fact that `resultSet` is mutated in the specs for `visitNode` and `visitChildren`

It isn't necessary to return the set, because `Set` is a mutable datatype. Each recursive call has a chance to contribute its results by mutating the set that they all share.

After these changes, the code looks like this:

```
/**
 * Finds files or folders whose names start with pattern,
 * searching recursively from a root file or folder.
 * @param file a file in the filesystem
 * @param pattern pattern to look for
 * @param resultSet all matching files found are added to this set
 */
public static void visitNode(File file, String pattern, Set<File> resultSet) {
  if (file.getName().startsWith(pattern)) { resultSet.add(file); }
  if (file.isDirectory()) {
    visitChildren(file.listFiles(), pattern, resultSet);
  }
}

/**
 * Finds files or folders whose names start with pattern,
 * searching recursively from a list of root files or folders.
 * @param files a list of files
 * @param pattern pattern to look for
 * @param resultSet all matching files found are added to this set
 */
public static void visitChildren(File[] files, String pattern, Set<File> resultSet) {
  for (File file : files) {
    visitNode(file, pattern, resultSet);
  }
}
```

This is an example of a design pattern sometimes called an *accumulator*, a mutable datatype passed around through a recursive process that accumulates results. Using mutable datatypes is less safe from bugs than immutable datatypes, however. It would be better to design our specs and implementations to remove mutation. The next exercise will do that.

CHECK EXPLAIN

Immutable results

Instead of mutating a result set provided by the caller, let's build our implementation around immutable sets. Select the best pieces of code to complete the implementation:

```
public static Set<File> visitNode(File file, String p
attern) {
    Set<File> resultSet = ▶▶A◀◀;
    if (file.getName().startsWith(pattern)) { resultS
et.add(file); }
    if (file.isDirectory()) {
        ▶▶B◀◀(visitChildren(file.listFiles(), patter
n));
    }
    return ▶▶C◀◀;
}

public static Set<File> visitChildren(File[] files, S
tring pattern) {
    Set<File> resultSet = ▶▶A◀◀;
    for (File file : files) {
        ▶▶B◀◀(visitNode(file));
    }
    return ▶▶C◀◀;
}
```

The same three pieces will complete both `visitNode` and `visitChildren` …

A ✔
- (•) `new HashSet<>()` ☑
- ( ) `Collections.unmodifiableSet(new HashSet<>())`
- ( ) `Set.of()`

B ✔
- ( ) `resultSet.add`
- (•) `resultSet.addAll` ☑
- ( ) `resultSet.retainAll`

C ✔
- ( ) `resultSet`
- ( ) `new HashSet<>(resultSet)`
- (•) `Collections.unmodifiableSet(resultSet)` ☑

The implementations of both functions accumulate their results by mutating a mutable `Set` whose scope is limited to that function alone. This requires a mutable `resultSet` in (A). Both `Collections.unmodifiableSet` and `Set.of` would return immutable sets.

In (B), the recursive call has just returned a set of results, and we add all of those results to our current result set.

And in (C), we have the opportunity to defend against mutability bugs by wrapping `resultSet` in an immutable wrapper. Once the function returns, that wrapper will have the only reference to the original mutable `resultSet`, preventing any future mutation. When we write the specs for `visitNode`/`visitChildren`, we can choose whether to guarantee immutability of the returned sets as part of the specs.

CHECK EXPLAIN

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. Mutual recursion is often intentional, 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 are 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 are 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. If the maximum depth of recursion grows only logarithmically with the size of the input (as in, say, a recursive binary search), then this is rarely a problem. But if the maximum depth grows *linearly* with the input (as in factorial and Fibonacci), then the stack size may become a limit on the size of the problem that your recursive implementation can solve.

READING EXERCISES

subsequences("123456")

Recall the implementation of `subsequences()` from the start of this reading:

```
public static String subsequences(String word) {
    if (word.isEmpty()) {
        return ""; // base case
    } else {
        char firstLetter = word.charAt(0);
        String restOfWord = word.substring(1);

        String subsequencesOfRest = subsequences(rest
OfWord);

        String result = "";
        final int withTrailingEmptyStrings = -1; // s
ee String.split() spec
        for (String subsequence : subsequencesOfRest.
split(",", withTrailingEmptyStrings)) {
            result += "," + subsequence;
            result += "," + firstLetter + subsequenc
e;
        }
        if (result.startsWith(",")) result = result.s
ubstring(1); // remove extra leading comma
        return result;
    }
}
```

For `subsequences("123456")`, how deep does its recursive call stack get — i.e., how many calls to `subsequences()` are on the stack at the same time?

✔ 7 7

❯ `subsequences()` is called on “123456”, “23456”, “3456”, “456”, “56”, “6”, “”, for a total of 7.

CHECK EXPLAIN

Common mistakes in recursive implementations

Here are three 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.
- Aliases to a mutable data structure are inadvertently shared, and mutated, among the recursive calls.

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 sometimes fails faster.

READING EXERCISES

Danger zone

Here is a buggy recursive implementation:

```
/**
 * @param list must be nonempty
 * @return the maximum integer in list
 */
public static int maxList(List<Integer> list) {
  return maxOfRange(list, 0, list.size()-1);
}

// helper method -- finds the max of list[start]...li
st[end]
private static int maxOfRange(List<Integer> list, int
start, int end) {
  if (start == end) {
    return list.remove(start);
  } else {
    int midpoint = (start + end + 1)/2;
    return Math.max(maxOfRange(list, start, midpoin
t), maxOfRange(list, midpoint+1, end));
  }
}
```

Which of the following bugs does this implementation suffer from? If a single bug could be described in more than one way, select all the matching choices.

- [x] recursive case may fail to reduce to a smaller subproblem
- [x] a base case is missing
- [x] some inputs lead to infinite recursion
- [ ] mutual recursion between `maxList` and `maxOfRange` leads to infinite recursion
- [x] aliases to a mutable object are shared among recursive calls, and harmful mutation occurs

When the recursion reaches a range with only two elements (`start` + 1 == `end`), then `midpoint` will be the same as `end`, and `maxOfRange` will call itself with the same arguments again. This recursive case did not shrink the subproblem.

Another way to characterize the same bug is that the recursion is missing a base case where the range has only two elements.

As a result of this bug, calling `maxList` on a two-element list leads to an infinite recursion.

There is no mutual recursion between `maxList` and `maxOfRange` – `maxOfRange` only calls itself recursively.

The base case mutates `list`, which invalidates the `start` and `end` indexes that will be used by other recursive calls.

CHECK EXPLAIN

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 unreassignable 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.

More practice

If you would like to get more practice with the concepts covered in this reading, you can visit the question bank. The questions in this bank were written in previous semesters by students and staff, and are provided for review purposes only – doing them will not affect your classwork grades.