Lecture 12: Recursion

Sam McCauley
Data Structures, Spring 2026

Recursion

So far, we have seen methods call other methods in Java. Today we will talk about recursion, where a method calls *itself*. This idea has a surprising amount of depth in computer science: in fact, you are likely to see it many more times if you take more computer science classes.

Recursion can be counterintuitive at times. It is also famously difficult to debug: since the method calls itself, small bugs can "grow" into more significant ones.

The goal of our recursion discussion in this course is twofold. First, we want to give more practice with recursion. Second, we want to start talking about how to approach recursion: how do you think about problems in a way that makes recursive techniques easier to come up with and easier to implement.

Recursion Discussion Recursion has upsides and downsides.

Recursive code is often much shorter than non-recursive code. It can also be much simpler.

However, recursion comes at a cost. It can be quite counterintuitive. As mentioned above, bugs can be magnified by recursion, and can be hard to track down. On larger inputs, recursion can cause the computer's stack to grow too large—we haven't discussed the stack yet, but long story short recursion does not always scale well.

Some Recursion Examples

Examples are perhaps the best way to get a sense of how recursion works.

In these examples, I will ignore normal bounds/error checking—while it would be necessary in the final code, I want to focus on the main recursive idea.

Mathematical Examples Let's start with some mathematical examples of recursion.

The factorial function, written $n!$, is a common function in combinatorics. It counts the number of ways to order n items.

$n!$ is the product of the first n numbers:

$$n! = n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1$$

For example, $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$.

Now, let's define this function recursively. Since

$$n! = n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1$$

which means that

$$(n-1)! = (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1$$

we can do something clever: combine these two formulas to obtain that if $n > 1$,

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

This is very easy to implement recursively. To find $n!$, we will use two steps: first, we'll find $(n-1)!$. We will do this by using our own factorial method. Then, we'll multiply by n. If $n = 1$, we can just return 1 instead.

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

We can do something similar with the Fibonacci numbers. The Fibonacci numbers are: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$. The first two Fibonacci numbers are 1; to obtain the next number in the sequence we add the two previous numbers. So $1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, 3 + 5 = 8$, and so on.

This immediately gives a recursive way to calculate the nth Fibonacci number.

```java
public static int fibonacci(int n) {
    if(n == 1 || n == 2) {
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

Data Structure Examples Perhaps more concrete are examples of methods we've already seen in the course. Let's look at `contains` for an `ArrayList<E>` for example. We already have a perfectly working `contains` method, but for practice, can we implement it recursively?

Let's look at a `contains` method that additionally takes two numbers `start` and end as an argument. `contains(E element, int start, int end)` returns `true` if one of the items in the slots from `start` to end is equal to `element`. We'll include `start` but not end in the range. In other words, we want to check if there is an equal element in the slots $\{start, start+1, \ldots, end-1\}$.

Let's begin with a simple case that we can solve without recursion, the **base case.** I'm going to suggest the following base case: if there are no entries in our range—that is to say, if `start == end`—then we can safely return `false`.[1]

Now, let's write the recursive part of our approach. As usual, the recursion has two parts. First, I will do a small amount of work. Then, I will recurse on a smaller instance of the problem.

In particular, the plan here is to compare the item in slot `start` to `element`. If they are equal, I can return `true`. Otherwise, I can recursively see if the query element is in any slot from `start + 1` to `end`.

```java
public boolean contains(E element, int start, int end) {
    if(start == end) {
        return false;
    }
    if(arr[start].equals(element)) {
        return true;
    }
    return contains(element, start+1, end);
}
```

The Parts of a Recursive Method

As we've implied so far, a recursive method has several parts.

First is the **base case**. This is some "easy" version of the problem that we can solve immediately. Usually, the base case occurs when the problem size is 1 or 0; this is the first thing to try. However, base cases can take many forms.

Which base case to choose is not a matter of right or wrong; it is effectively a matter of style. Most problems we see will have a few reasonable base cases; you can pick whichever makes the most sense to you.

Second, we need to make one or more **recursive calls** to a smaller version of the same method.

Third, we often need to do some additional work. How do we use the result of the recursive calls to find a solution to our problem? We're mostly going to see examples where we need to do something

[1] There are other base cases that work perfectly fine: for example, if `start == end - 1` then we can compare the item in slot `start` to `element`.

relatively simple here: in the above examples we multiplied by n, or we compared equality with the element in position `start`.

Oftentimes, the work we do is checking the part we "removed." At the end of the day we're recursing on a smaller instance—that is to say, we're removing a piece of the instance and recursing on the rest. Usually, the work we do before the recursive call is handling that removed piece.

For example, in the `contains` method above, we begin by searching slots `start` to `end`, but our recursive call is on slots `start + 1` to `end`. That is to say, the slot `start` is removed from the instance. While the method runs, we must then handle slot `start`—we check if it matches the element we are querying for.

Things to Look for With Recursion

Every time you write a recursive method, you should use the following checklist to help verify that your method is correct. (This checklist is not a catch-all; it's just an easy way to catch errors.)

1. Make sure you have a base case. If you do not have a base case, the method always calls itself—this means it will loop infinitely.
2. Make sure that the recursive calls *make progress*. In this class, we will only look at recursive calls where the problem *gets smaller*. Again, this is important to make sure that the method finally returns. We want that the problem size we consider gets smaller and smaller until finally reaching the base case.

Helper Methods

Oftentimes when using recursion, we use *helper methods*. These are similar to the helper methods we've seen in the past—they're back-end methods which help other methods accomplish a task.

For recursive methods, helper methods are often used to *add parameters*.

Let's look at one concrete example: let's say we want to implement `contains(E element)` on an `ArrayList` recursively, rather than using a loop or calling `indexOf()`.

We can't implement `contains` recursively because we have no way to "make it smaller": the only parameter of `contains` is `E element`. We have to *add* parameters to allow us to accomplish this task recursively.

These parameters must "make it smaller:" this means that the parameters must allow us to limit `contains` to only search a smaller portion of the list.

In fact, we've already seen how to do this. We already created a `contains(E element, int start, int end)` method that determines if `element` is contained between `start` and end. With this helper method, we can create a recursive `contains(E element)` method by setting `start = 0` and `end = numElems`. That way, our recursive `contains` method searches the entire list.

```java
public boolean contains(E element) {
    return contains(element, 0, numElems);
}
```

Keeping Track of Recursive Programs

At times it can be hard to reason about how recursive programs run. In our previous programs of the class, the answer to the question "what is Java doing right now" was always along the lines of: "it's in the `add()` method" or "it's in that `while` loop." For recursive programs, the answer is more subtle. The program is *always* executing (say) the `contains` method; what matters is *which* `contains` method is executing.

Let's look at two examples of a recursive program, and let's reason about what they do. At the end, we'll discuss strategies to approach recursive programs.

First Example.

```java
public static void printNums(int num) {
    if(num == 0) {
        return;
    }
    System.out.println(num);
    printNums(num - 1);
}
```

Let's start with the base case. If `num == 0` then this method does nothing.

Let's go one "level" up. If `num == 1` then this method prints `1`, then it calls itself with `num == 0` (and does nothing). So if `num == 1`, the method prints:

```
1
```

Let's go one more "level" up. If `num == 2` then this method prints `2`, then it calls itself with `num == 1`; we already know that this prints `1`. So in total, the output is:

```
2
1
```

When we call this method, it prints the value of num. Then, it calls itself with argument `num - 1`, which prints the value `num - 1`; this calls itself with argument `num - 2`, which prints the value `num - 2` and so on. In general, this method prints the numbers from num to `1` in descending order.

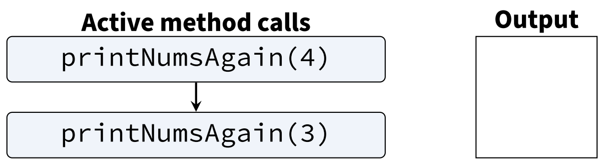
Second Example Let's perturb what the method does just a little bit. We'll swap so that the print statement is *after* the recursive call.

```java
public static void printNumsAgain(int num) {
    if(num == 0) {
        return;
    }
    printNumsAgain(num - 1);
    System.out.println(num);
}
```

For `num == 0` or `num == 1` the output is the same as the previous method.

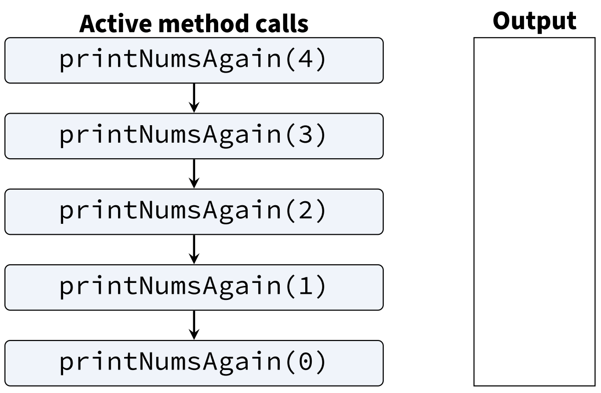
However, as num gets larger, this method becomes more difficult to reason about. Let's look at what happens with `num == 4`; we'll discuss why it's harder to reason about, and talk about how to draw a "recursion tree" diagram to help us keep track of what the recursion is doing.

When we call `printNumsAgain(4)`, the first thing it does is call `printNumsAgain(3)`. It won't call `System.out.println(4)` until `printNumsAgain(3)` is done. In other words, this method is still processing; let's keep track of this by writing it down.



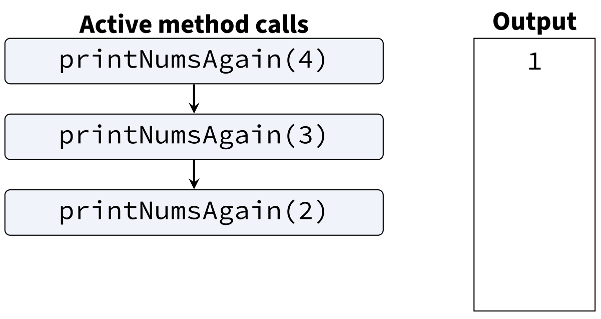
Now, `printNumsAgain(3)` is running. (`printNumsAgain(4)` is too! It's waiting for `printNumsAgain(3)` to finish.) The first thing it does is call `printNumsAgain(2)`.

Now, `printNumsAgain(2)` is running. The first thing it does is call `printNumsAgain(1)`. Then, `printNumsAgain(1)` calls `printNumsAgain(0)`.



Finally, we've reached our base case. `printNumsAgain(0)` does nothing, and returns to `printNumsAgain(1)`. When control returns to `printNumsAgain(1)`, it runs `System.out.println(1)` and returns.

Right now, our current status looks like this. All of the methods on the left are running, and on the right hand side is the current output.



`printNumsAgain(2)` prints 2 to the screen and returns; then `printNumsAgain(3)` prints 3 to the screen and returns, and finally `printNumsAgain(4)` prints 4. Our final output is:

```
1
2
3
4
```

Practice. What does this method print?

```java
public static void printNumsTwice(int num) {
    if(num == 0) {
        return;
    }
    System.out.println(num);
    printNumsAgain(num - 1);
    System.out.println(num);
}
```

What about this one? (We'll see a similar example next which also makes multiple recursive calls; it may be a good idea to look at that one before trying this.) The pattern for this one is not simple—it's not "count down from num to 1" or anything like that—but try to trace through what is printed for `num == 3`.

```java
public static void printNumsTwoCalls(int num) {
    if(num == 0) {
        return;
    }
    printNumsTwoCalls(num - 1);
    System.out.println(num);
    printNumsTwoCalls(num - 1);
}
```

Another Recursion Example

Let's look at a more complex recursion example.

The Problem Suppose there are 6 students enrolled in a course and you want to schedule office hours so that every student has a chance to attend. You create a poll with 10 possible time slots, and each student indicates which slots they can make:

- Student 1 can make slots $\{1, 6, 8\}$
- Student 2 can make slots $\{2, 5, 8\}$
- Student 3 can make slots $\{3, 4, 9, 10\}$
- Student 4 can make slots $\{6, 7, 8, 9\}$
- Student 5 can make slots $\{2, 3, 4\}$
- Student 6 can make slots $\{1, 3, 4, 5, 9\}$

Our goal is to answer the following: what is the *minimum* number of office hours slots you need to hold so that every student can attend at least one? If it is not possible to cover every student, we'll return -1.

For this instance, the answer is 3: holding slots $\{2, 6, 9\}$ covers all six students.

Solution. To solve this problem recursively, we first need a **base case**. If there is only one remaining time slot, we check whether holding it covers all remaining students. If so, we return 1; otherwise, there is no valid solution, and we return -1.

For the **recursive step**, we look at the last remaining time slot. While we don't know what the final solution is, one of the following must be true: either the last remaining time slot is in the final solution, or it is not.

Of course, we don't know which of these is the case. So we'll try both! We'll recursively find the best solution that does include the last remaining time slot, and also find the best solution that does not include the last remaining time slot. After both return, we'll take whichever found the best solution.

Let's look at how we will solve each of these cases recursively.

- **Take the slot:** Remove the last slot from our list *and* remove all students whose availability includes that slot—those students are now covered. Then, recursively solve the smaller problem. If this returns a valid solution `solWithSlot`, the total cost is $1 +$ `solWithSlot` (the cost for this slot, plus whatever further slots were needed recursively).
- **Skip the slot:** Remove the last slot from our list but keep all students. Recursively solve the smaller problem, storing the result in `solWithoutSlot`.

After both recursive calls return, we pick the better of the two options. If both are -1 (impossible), we return -1. If only one is valid, we return that one. If both are valid, we return the minimum.

Implementing this Method We are given a variable `slots` which represents the largest slot in the instance (`slots` is 10 in the above example). We are also given the availability of each student. For simplicity, let's assume that we have a `Student` class that stores the available slots for each student.

The following code uses as a subroutine a method `coversAll` that checks if all students can make a given slot, and another method `removeCovered` that returns a new array of students after removing all students that are covered by a given slot.

In our recursive calls, we need to "remove the last slot". Since the available slots are $\{1, 2, \ldots,$ `slots`$\}$, we can do this by using `slots - 1` as the argument.

```
/* Find the minimum number of slots necessary to accommodate all
   students
 * @param slots gives the number of slots (assume 1 ... slots)
 * @param students is an array of students
 * @return the minimum number of slots necessary to accommodate all
     students
```

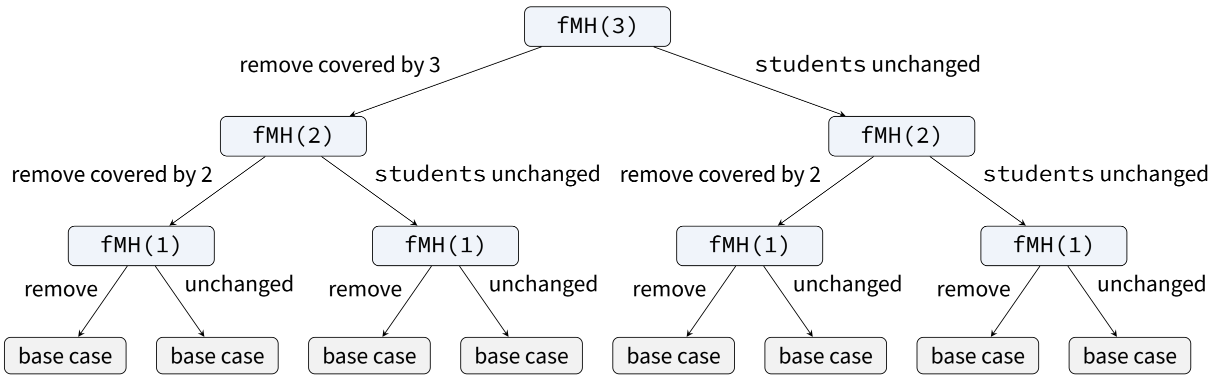
```java
 */
public int findMinHours(int slots, Student[] students) {
    if(slots == 1) {
        //assume the following method call checks if all students can
           make slot 1
        if( coversAll(1, students) ) {
            return 1;
        } else {
            return -1;
        }
    }
    //the following method removes all Students from students that are
    //covered from the given slot
    Student[] newStudents = removeCovered(slots, students);

    int solWithSlot = findMinHours(slots - 1, newStudents);
    int solWithoutSlot = findMinHours(slots - 1, students);
    if(solWithSlot == -1) {
        return solWithoutSlot;
    }
    if(solWithOutSlot == -1) {
        return 1 + solWithSlot;
    }
    if(1 + solWithSlot < solWithoutSlot) {
        return 1 + solWithSlot;
    } else {
        return solWithoutSlot;
    }
}
```

Why Does This Terminate? Every recursive call reduces the number of remaining time slots by one, so the problem strictly gets smaller with each call. Eventually we reach the base case with one slot remaining.

Is This Slow? Yes, this approach is fairly slow—it won't work beyond 20-30 slots. At each step we branch into two recursive calls, so the total number of calls can be exponential in the number of time slots. In fact, this type of problem (called a *set cover* problem) is known to be computationally hard: there is no known efficient algorithm that is always correct.

Drawing the Recursive Calls. The diagram below shows every recursive call made on an instance with `slots = 3`. At each internal node we branch into two calls: one where we *take* the current slot (removing covered students), and one where we *skip* it. The leaves are the base cases.

The first call is at the top of the diagram, where `findMinHours` is called with `slots = 3`. This method makes two calls, to `findMinHours` with `slots = 2` (these calls are denoted using the two arrows coming out of the top node). For one call we remove all `Students` who have `3` as an available hour; for the other we recurse on the same `students` array without changing it.


fMH(3)
remove covered by 3
students unchanged
fMH(2)
fMH(2)
remove covered by 2
students unchanged
remove covered by 2
students unchanged
fMH(1)
fMH(1)
fMH(1)
fMH(1)
remove
unchanged
remove
unchanged
remove
unchanged
remove
unchanged
base case
base case
base case
base case
base case
base case
base case
base case