Lec 12: Recursion

Sam McCauley

March 9, 2026

Admin

- Lab out
  - "Pre-lab" problems that you can fully collaborate on with each other, TAs, me, etc.
  - Won't be handed in. Sharing code is OK for `RecursionPreLab.java` (only this file :))
  - Strongly suggested! Only real way to learn recursion is to do it, so the lab asks you to come up with some sophisticated recursions

Javadoc Comments

Code Style

- Will start enforcing code style more strictly
- Requirements listed in a handout on the website
- New: required to have Javadoc comments for every method you write

What Are Javadoc Comments?

- Special comment format for Java
- Every class and every `public` method should have a Javadoc comment placed immediately before it
- Javadoc comments begin with `/**` and end with `*/`
- Every line in between should start with `*`

```
/**
 * A brief description of the method or class goes here.
 * Further details can follow on additional lines.
 */
```

Javadoc Block Tags

- The first line of the comment gives a concise description
- Further lines (optional) give more details
- Methods also require block tags (lines starting with `@`):
  - `@param` — describes a parameter (one per parameter)
  - `@return` — describes the return value (required for non-`void` methods)
  - `@pre` — describes any *preconditions* the method assumes
  - `@post` — describes any *postconditions*: what the method does or changes
  - These last two are special for CS 136

Javadoc Example

```java
/**
 * Returns the index of the first occurrence of element in
   the list.
 * @param element the element to search for
 * @return the index of the first occurrence, or -1 if not
   found
 * @pre element is not null
 * @post the list is unchanged
 */
public int indexOf(E element) {
    for (int index = 0; index < numElems; index++) {
        if (element.equals(arr[index])) {
            return index;
        }
    }
    return -1;
}
```

Javadoc for a Class

- Classes also need a Javadoc comment

```java
/**
 * A generic ArrayList that stores elements of type E.
 * Supports add, get, set, remove, and search operations.
 */
public class ArrayList<E> {
    ...
}
```

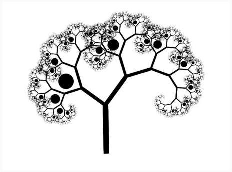
- No `@param` or `@return` needed for classes—just a description

Why Javadocs

- Helps others use your class
- Can use the `javadoc` command to build a webpage explaining your class! Let's look at a demo
- Most Java code is documented this way

Recursion

What Is Recursion?



- So far: our methods call *other* methods
- Recursion: a method calls *itself*
- This idea appears throughout computer science—you'll likely see it in many future courses
- It's just another way to write programs. Essentially any program can be rewritten to use recursion; alternatively, it's possible to entirely remove it from any program—it's another tool in your toolbox

Recursion: Pros and Cons

- Pros:
    - Recursive code is often much shorter and simpler than non-recursive code
- Cons:
    - Can be counterintuitive
    - Bugs are hard to track down—they can be magnified by repeated calls
    - On large inputs, recursion can cause the call stack to grow too large (this should not happen in this class unless something goes wrong)

Recursion Examples

Factorial

- The factorial function $n!$ counts the number of ways to order n items:

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

- For example, $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24$
- Notice:

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

- This gives us a recursive definition!

Factorial in Java

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

- To find $n!$: first find $(n-1)!$ by calling ourselves, then multiply by n
- If $n = 1$, just return 1

Fibonacci

- The Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, . . .
- First two are 1; each subsequent number is the sum of the two before it
- Immediately gives a recursive implementation:

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

Data Structure Example: `contains`

- Can we implement `contains` on an `ArrayList<E>` recursively?
- Add parameters `start` and `end`; we want to know if `E element` appears in any slot between `start` and `end` (not inclusive for `end`)
- Let’s work together on the board to solve this problem recursively.

```java
public boolean contains(E element, int start, int end) {

}
```

Data Structure Example: `contains`

```java
/* Finds if element contained in index between start and end
 * @param element: the element to find in the range
 * @param start: starting element of the range (inclusive)
 * @param end: ending element of the range (not inclusive)
 * @pre start and end are between 0 and numElems - 1
 * @return true if element is in the range; false otherwise
 */
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);
}
```

Parts of a Recursive Method

Three Parts of a Recursive Method

1. Base case — an “easy” version of the problem we can solve immediately
    - Usually occurs when problem size is 0 or 1
    - Several reasonable choices often exist; pick whichever makes sense
2. Recursive call(s) — call the same method on a *smaller* instance
3. Additional work — use the result of recursive call(s) to solve the full problem
    - Usually: handle the piece that was “removed” before recursing

Recursion Checklist

- Every time you write a recursive method, check:

1. Do you have a base case?
    - Without one, the method calls itself forever—infinite loop
2. Do the recursive calls make progress?
    - The problem size must get smaller with each call
    - Eventually, it must reach the base case

Helper Methods

Why Helper Methods?

- Sometimes we can’t write a recursive method directly—no way to “make it smaller”
- Example: `contains(E element)` has only one parameter; how do we limit the search?
- Solution: use a helper method that *adds parameters*

Helper Method Example

- We already wrote `contains(E element, int start, int end)`
- Now we can write a recursive `contains(E element)` using it as a helper:

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

- The helper does the recursive work; the public method just calls it with the full range

Tracing Execution of Recursive Programs

Example: `printNums`

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

- What does `printNums(3)` print?
- Prints num, then calls itself with `num - 1`
- Output: 3, 2, 1 (descending)

Example 2: `printNumsAgain`

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

- Same code, but `println` is *after* the recursive call
- What does `printNumsAgain(4)` print?

Tracing `printNumsAgain(4)`

- `printNumsAgain(4)` first calls `printNumsAgain(3)`, which calls `printNumsAgain(2)`, ...all the way to the base case
    - No printing yet!
- Then, as each call returns, it prints *its own* num
- Let's track these calls on the board. As each recursive call is made we'll write it down, and cross it out as each completes.
- Output: 1, 2, 3, 4

Binary Search

- How can we do `contains(Integer element, int start, int end)` on a *sorted* `ArrayList<Integer>`?

- Does someone have an idea for a high-level strategy?
  - Compare to element in middle slot: `(end + start)`/2. Recurse on one side or the other depending on if it is larger or smaller

  - In pairs: how can we write this method recursively?

  - At home: can you write this method with a loop instead? How does the code change?

A scheduling problem: Creating Office Hours

- Let’s say there are 6 enrolled students enrolled in a course. I want to schedule office hours so that every single student has a chance to attend office hours
- I create a doodle poll with 10 options for my office hours
- Each student states which of the office hours they can attend
- What is the minimum number of office hours I can hold so that every student can make at least one hour?

Creating Office Hours

The possible time slots are $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

- 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 $\{1, 3, 4, 5, 9\}$

Creating Office Hours

The possible time slots are $\{1, \mathbf{2}, 3, 4, 5, \mathbf{6}, 7, 8, \mathbf{9}, 10\}$.

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

This is solvable with 3 slots. (I think that's optimal?)

Solving the Office Hours Problem Recursively

- Where to start?
- Can someone come up with a base case?
    - When there’s only one time slot, our only choice is to take it or not take it
    - Second base case option: if there is one student, if they have an hour that matches with a time slot, then 1 slot is optimal. Otherwise, can’t solve.
    - Another option: zero students or zero time slots

Office Hours Scheduling: Breaking into a Smaller Subproblem

- How can we make this subproblem smaller?
- Let’s look at the first possible time slot
- There are two options: either this time slot is in the solution, or it isn’t
    - What happens in each case? How do we make a recursive call?
    - Let’s assume we take the first time slot. Then we can remove that time slot from our list, and remove all students who can attend that time slot. That gives us a new instance of office hours scheduling!
    - Let’s assume we *don’t* take the first time slot. Then we can remove that time slot from our list. That gives us a new instance of office hours scheduling!

Office Hours Scheduling Solution

- If there is only one remaining slot, just determine if it meets all students' needs. Return 1 if so; -1 otherwise.
- Otherwise:
    - Recursively find the office hours scheduling solution with the first slot removed, and with all students whose availability matches that slot removed. Store this optimal solution in `solWithSlot`
    - Recursively find the office hours scheduling solution with the first slot removed. Store this optimal solution in `solWithOutSlot`
- If both `solWithSlot` and `solWithOutSlot` are not -1, return the minimum of $1+$ `solWithSlot` and `solWithOutSlot`
- If just one is -1, return the other
- If both are -1, return -1.

Discussion

- Why does this method work? What do we need to guarantee for a recursion to terminate?
    - Need to make progress towards the base case!
    - Each recursive call reduces the number of slots by 1
- Is this method fast? Is that OK?
    - No, this is not fast at all.
    - In algorithms you will learn that this problem is computationally intensive—there’s no known solution that’s efficient and always correct

Creating Office Hours

Let's look at what our algorithm does on a smaller instance.

The possible time slots are $\{1, 2, 3\}$.

- Student 1 can make slots $\{1, 2\}$
- Student 2 can make slots $\{2, 3, \}$
- Student 3 can make slots $\{3\}$
- Student 4 can make slots $\{1\}$

Smaller example

Let’s look at what our algorithm does on a smaller instance.
The possible time slots are $\{1, 2, 3\}$.

- Student 1 can make slots $\{1, 2\}$
- Student 2 can make slots $\{2, 3, \}$
- Student 3 can make slots $\{3\}$
- Student 4 can make slots $\{1\}$

- If there is only one remaining slot, determine if it suffices for all students; otherwise:
    - Recursively find solution with the first slot removed, and with all students whose availability matches that slot removed.
    - Recursively find solution with the first slot removed. Store this optimal solution in `solWithOutSlot`

What does this print?

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