

Recursion (Part 1)



EECS2101 E:
Fundamentals of Data Structures
Summer 2026

CHEN-WEI WANG

Beyond this lecture ...



- Fantastic resources for developing your recursive skills:
<http://codingbat.com/java/Recursion-1>
<http://codingbat.com/java/Recursion-2>
- The **best** long-term approaches for mastering recursion are:
 - learning a **functional programming** language
[e.g., Haskell: <https://www.haskell.org/tutorial/>]
 - learning to develop a **compiler** (after learning **trees** in this course)
[e.g., ANTLR4 from [EECS4302](#)]

Background Study: Basic Recursion



- It is assumed that, in EECS2030, you learned about the basics of **recursion** in Java:
 - What makes a method recursive?
 - How to trace recursion using a **call stack**?
 - How to define and use **recursive helper methods** on **arrays**?
 - If needed, review the above assumed basics from the relevant parts of EECS2030:
 - From [F'21](#): Parts A – C, Lecture 8, Week 12
 - From [F'24](#): Lecture 24, Sec. E (Tower of Hanoi)
- Tips.**
- Skim the **slides**: watch lecture videos if needing explanations.
 - Recursion lab from EECS2030-F22: [here](#) [Solution: [here](#)]
 - Ask questions related to the assumed basics of **recursion**!
- Assuming that you know the basics of **recursion**, we will:
 - Look at an advanced example of **recursion on arrays** together.
 - Have you complete an assignment on the more advanced recursion problems.

Learning Outcomes of this Lecture



This module is designed to help you:

- Quickly review the **recursion basics**.
- Know about the **resources** on **recursion basics**.
- Get used to the **more advanced** use of recursion.

Recursion: Principle



- **Recursion** is useful in expressing solutions to problems that can be **recursively** defined:
 - **Base Cases:** Small problem instances immediately solvable.
 - **Recursive Cases:**
 - Large problem instances *not immediately solvable*.
 - Solve by reusing *solution(s) to strictly smaller problem instances*.
- Similar idea learnt in high school: [**mathematical induction**]

5 of 12

Tracing Method Calls via a Stack



- When a method is called, it is **activated** (and becomes **active**) and **pushed** onto the stack.
- When the body of a method makes a (helper) method call, that (helper) method is **activated** (and becomes **active**) and **pushed** onto the stack.
 - ⇒ The stack contains activation records of all **active** methods.
 - **Top** of stack denotes the **current point of execution**.
 - Remaining parts of stack are (temporarily) **suspended**.
- When entire body of a method is executed, stack is **popped**.
 - ⇒ The **current point of execution** is returned to the new **top** of stack (which was **suspended** and just became **active**).
- Execution terminates when the stack becomes **empty**.

6 of 12

Tracing Method Calls via a Stack



- Can you identify the pattern of a Fibonacci sequence?

$$F = 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \dots$$

- Here is the formal, **recursive** definition of calculating the n_{th} number in a Fibonacci sequence (denoted as F_n):

$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ F_{n-1} + F_{n-2} & \text{if } n > 2 \end{cases}$$

- Your tasks are then to review how to
 - **implement** the above mathematical, recursive function in Java
 - **trace**, via a stack, the recursive execution at runtimeby studying **this video** (≈ 20 minutes):

7 of 12

Making Recursive Calls on an Array



- For **efficiency**, we exploit the feature of **call by value**, by:
 - passing the **reference** of the same array
 - specifying the **range of indices** to be considered

```
void m(int[] a, int from, int to) {  
    if(from > to) { /* base case */ }  
    else if(from == to) { /* base case */ }  
    else { m(a, from + 1, to) } }
```

- $m(a, 0, a.length - 1)$ [Initial call; entire array]
- $m(a, 1, a.length - 1)$ [1st r.c. on array of size $a.length - 1$]
- $m(a, a.length - 1, a.length - 1)$ [Last r.c. on array of size 1]

- **Required Task:**

Study the two examples `allPositive` and `isSorted` from the background study.

8 of 12

Recursion: All Positive (1)

Problem: Determine if an array of integers are all positive.

```
System.out.println(allPositive({})); /* true */
System.out.println(allPositive({1, 2, 3, 4, 5})); /* true */
System.out.println(allPositive({1, 2, -3, 4, 5})); /* false */
```

Base Case: Empty array → Return *true* immediately.

The base case is *true* ∴ we can *not* find a counter-example (i.e., a number *not* positive) from an empty array.

Recursive Case: Non-Empty array →

- 1st element positive, **and**
- *the rest of the array is all positive*.

Exercise: Write a method `boolean somePositive(int[]`

`a)` which *recursively* returns *true* if there is some positive number in `a`, and *false* if there are no positive numbers in `a`.

Hint: What to return in the base case of an empty array? [*false*]

∴ No witness (i.e., a positive number) from an empty array

9 of 12

Recursion: All Positive (2)

```
boolean allPositive(int[] a) {
    return allPositiveHelper(a, 0, a.length - 1);
}

boolean allPositiveHelper(int[] a, int from, int to) {
    if (from > to) { /* base case 1: empty range */
        return true;
    }
    else if (from == to) { /* base case 2: range of one element */
        return a[from] > 0;
    }
    else { /* recursive case */
        return a[from] > 0 && allPositiveHelper(a, from + 1, to);
    }
}
```

10 of 12

A More Advanced Example on Recursion

Assuming that you will review the assumed basic, let's go over an advanced example from paper to Eclipse:

- **Problem Description:**

<https://www.eecs.yorku.ca/~wangcw/teaching/lectures/2026/S/EECS2101/exercises/EECS2101-S26-Problem-Recursion-splitArray-Spec.pdf>

- **Java starter project** (with hints and **JUnit tests**):

https://www.eecs.yorku.ca/~wangcw/teaching/lectures/2026/S/EECS2101/exercises/ExtraRecursionProblemSplitArray_Starter.zip

11 of 12

Index (1)

Beyond this lecture ...

Background Study: Basic Recursion

Learning Outcomes of this Lecture

Recursion: Principle

Tracing Method Calls via a Stack

Tracing Method Calls via a Stack

Making Recursive Calls on an Array

Recursion: All Positive (1)

Recursion: All Positive (2)

A More Advanced Example on Recursion

12 of 12