Recursion (Part 1)

EECS2011 X:
Fundamentals of Data Structures
Winter 2023

CHEN-WEI WANG

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 (`https://www.eecs.yorku.ca/~jackie/teaching/lectures/index.html#EECS2030_F21`):
  - Parts A – C, Lecture 8, Week 12

  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 a basic 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***.

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*]
- Recursion can be easily expressed programmatically in Java:

```
m (i) {
  if(i == ...) { /* base case: do something directly */ }
  else {
    m (j);/* recursive call with strictly smaller value */
  }
}
```

  - In the body of a method *m*, there might be *a call or calls to m itself*.
  - Each such self-call is said to be a *recursive call*.
  - Inside the execution of $m(i)$, a recursive call $m(j)$ must be that $j < i$.

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*.

Tracing Method Calls via a Stack

- Can you identify the pattern of a Fibonacci sequence?

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

- 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 runtime

  by studying **this video** ($\approx$ 20 minutes):

Making Recursive Calls on an Array

- Recursive calls denote solutions to *smaller* sub-problems.
- *Naively*, explicitly create a new, smaller array:

```
void m(int[] a) {
  if(a.length == 0) { /* base case */ }
  else if(a.length == 1) { /* base case */ }
  else {
    int[] sub = new int[a.length - 1];
    for(int i = 1; i < a.length; i ++) { sub[i - 1] = a[i]; }
    m(sub) } }
```

- For *efficiency*, we pass the ***reference*** of the same array and specify 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]

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 somePostive(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

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);
  }
}
```

Recursion: Is an Array Sorted? (1)

Problem: Determine if an array of integers are sorted in a non-descending order.

```
System.out.println(isSorted({}));  true

System.out.println(isSorted({1, 2, 2, 3, 4}));  true

System.out.println(isSorted({1, 2, 2, 1, 3}));  false
```

Base Case: Empty array $\longrightarrow$ Return *true* immediately.
The base case is *true* $\because$ we can *not* find a counter-example (i.e., a pair of adjacent numbers that are *not* sorted in a non-descending order) from an empty array.
Recursive Case: Non-Empty array $\longrightarrow$

- 1st and 2nd elements are sorted in a non-descending order, **and**
- *the rest of the array*, starting from the 2nd element, *are sorted in a non-descending order* .

Index (1)