UNIVERSITY of WISCONSIN
LA CROSSE
COMPUTER SCIENCE

Week 11:
Recursion II

CS 220: Software Design II — D. Mathias

Recursion vs Iteration

iterative programming is the method of programming you've been using

- i.e., loops are exclusively used to repeat, make progress

recursive programming is a complementary method of programming

- i.e., recursion is used—sometimes in conjunction with loops—to make progress
- some programming languages use only recursion without loops
  - e.g., Scheme, Lisp, Haskell

Every iterative program can be written recursively and vice versa[1]

1: https://en.wikipedia.org/wiki/Church%E2%80%93Turing_thesis

Example: Recursion vs Iteration

Calculating factorials can be defined (iteratively) as below:

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

Which can be rewritten recursively:

$$f(0) = 1,\ f(n) = n! = n \cdot f(n - 1)$$

```java
public static int factorialIter(int n) {
    int sum = 1;
    if (n <= 1) { return sum; }

    while (n > 1) {
        sum *= n;
        n--;
    }
    return sum;
}
```

```java
public static int factorialRecur(int n) {

    if (n <= 1) {
        return 1;
    }
    /* else (n > 1) */
    return n * factorialRecur(n - 1);
}
```

Why Recursion?

- Pros
  - some algorithms are more elegant/concise/understandable recursively
    - particularly true for some 340 data structures
- Cons
  - takes up more space (i.e., memory) on the stack
    - rarely a problem if recursion is done well
    - some languages allow for *tail-call optimization*, which mitigates this; not supported in Java
  - can be difficult to understand if written poorly
    - but this is true of all code!

Parts of a Loop

```
public static int factorialIter(int n) {
    int sum = 1;
    if (n <= 1) { return sum; }

    while (n > 1) {
        sum *= n;
        n--;
    }
    return sum;
}
```

Every loop has four parts

Parts of a Loop

```
public static int factorialIter(int n) {
    int sum = 1;
    if (n <= 1) { return sum; }

    while (n > 1) {
        sum *= n;
        n--;
    }
    return sum;
}
```

Every loop has four parts

- initialization
  - set up a variable that will control the loop

Parts of a Loop

```
public static int factorialIter(int n) {
    int sum = 1;
    if (n <= 1) { return sum; }

    while (n > 1) {
        sum *= n;
        n--;
    }
    return sum;
}
```

Every loop has four parts

- initialization
  - set up a variable that will control the loop
- condition
  - a boolean expression to control when the loop stops

Parts of a Loop

```
public static int factorialIter(int n) {
    int sum = 1;
    if (n <= 1) { return sum; }

    while (n > 1) {
        sum *= n;
        n--;
    }
    return sum;
}
```

Every loop has four parts

- initialization
  - set up a variable that will control the loop
- condition
  - a boolean expression to control when the loop stops
- work
  - the code the loop will repeat

Parts of a Loop

```
public static int factorialIter(int n) {
    int sum = 1;
    if (n <= 1) { return sum; }

    while (n > 1) {
        sum *= n;
        n--;
    }
    return sum;
}
```

Every loop has four parts

- initialization
  - set up a variable that will control the loop
- condition
  - a boolean expression to control when the loop stops
- work
  - the code the loop will repeat
- progress
  - how the loop moves closer to termination

Parts of a Recursive Method

Every recursive method has **five** parts

```
public static int factorialRecur(int n) {

    if (n <= 1) {
        return 1;
    }
    /* else (n > 1) */
    return n * factorialRecur(n - 1);
}
```

Parts of a Recursive Method

```
public static int factorialRecur(int n) {

    if (n <= 1) {
        return 1;
    }
    /* else (n > 1) */
    return n * factorialRecur(n - 1);
}
```

Every recursive method has **five** parts

- initialization

Parts of a Recursive Method

```
public static int factorialRecur(int n) {

    if (n <= 1) {
        return 1;
    }
    /* else (n > 1) */
    return n * factorialRecur(n - 1);
}
```

Every recursive method has **five** parts

- initialization
- recursive case
  - one or more boolean expressions to control when to make a recursive call

Parts of a Recursive Method

```
public static int factorialRecur(int n) {

    if (n <= 1) {
        return 1;
    }
    /* else (n > 1) */
    return n * factorialRecur(n - 1);
}
```

Every recursive method has **five** parts

- initialization
- recursive case
  - one or more boolean expressions to control when to make a recursive call
- smallest value case
  - one or more boolean expressions to control when to solve a small problem directly

Parts of a Recursive Method

```
public static int factorialRecur(int n) {

    if (n <= 1) {
        return 1;
    }
    /* else (n > 1) */
    return n * factorialRecur(n - 1);
}
```

Every recursive method has **five** parts

- initialization
- recursive case
  - one or more boolean expressions to control when to make a recursive call
- smallest value case
  - one or more boolean expressions to control when to solve a small problem directly
- work

Parts of a Recursive Method

```
public static int factorialRecur(int n) {

    if (n <= 1) {
        return 1;
    }
    /* else (n > 1) */
    return n * factorialRecur(n - 1);
}
```

Every recursive method has **five** parts

- initialization
- recursive case
  - one or more boolean expressions to control when to make a recursive call
- smallest value case
  - one or more boolean expressions to control when to solve a small problem directly
- work
- progress
  - how the recursion moves closer to termination

Parts of a Recursive Method

```
public static int factorialRecur(int n) {

    if (n <= 1) {
        return 1;
    }
    /* else (n > 1) */
    return n * factorialRecur(n - 1);
}
```

Every recursive method has **five** parts

- initialization
- recursive case
  - one or more boolean expressions to control when to make a recursive call
- smallest value case
  - one or more boolean expressions to control when to solve a small problem directly
- work
- progress
  - how the recursion moves closer to termination

How to Write a Recursive Method

1. Identify the recursive structure in the problem and how to leverage it to solve the problem.
2. Identify the smallest value case(s). What instances are too small to make smaller?
3. Consider a larger case (but not too large!). Assume you have a method that can solve a problem that is smaller than that one.

    you don't **yet** have to know what that method is

4. If you can **assume** you have a method to solve that case, how can you write the code to solve the original case?

Example: Fibonacci

The Fibonacci sequence is as follows:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

This sequence can be described mathematically:

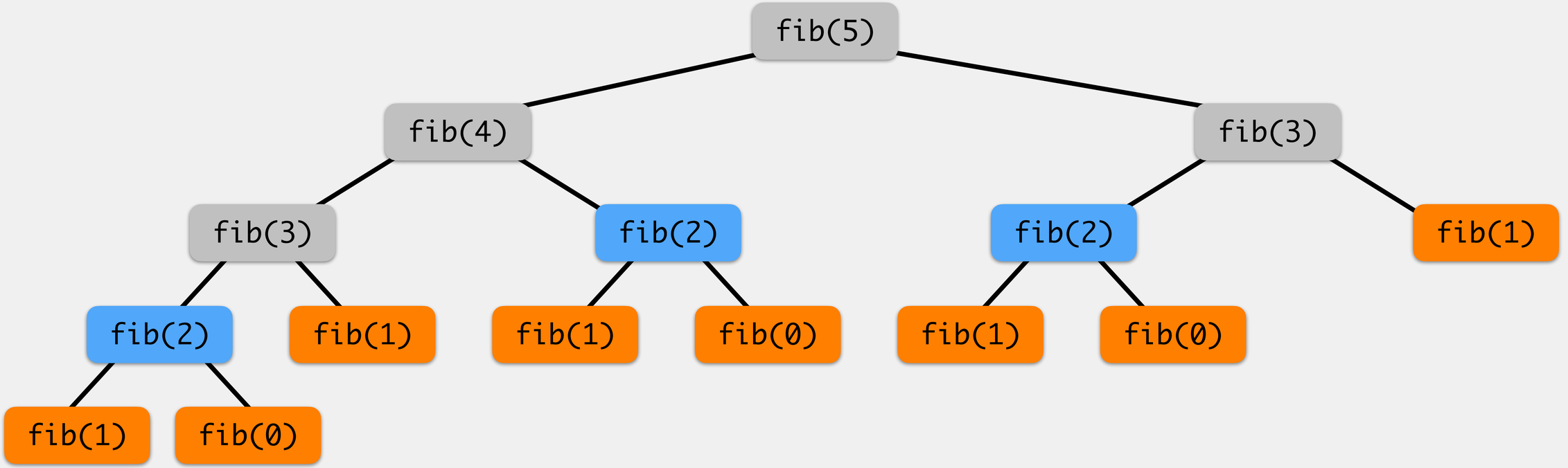
$$f(0) = 0,\ f(1) = 1,\ f(n) = f(n - 1) + f(n - 2)$$

1. Base case(s): f(0) = 0, f(1) = 1
2. Consider f(4) = f(3) + f(2). We're assuming we can already solve f(3) and f(2).
3. Let's write (general) code to solve f(4)!

```
public static int fib(int n) {

    if (n == 0 || n == 1){
        return n;
    }
    /* else (n > 1) */
    return fib(n - 1) + fib(n - 2);
}
```

Method Calls for fib(5)



Lots of repeat calculations!

Can be avoided through *memoization*, an optimization technique which *caches* (i.e., saves) the results from a computation to be used in the future

Memoization

```
public class Fibonacci {
    // the index will be n and the value at index n will be f(n)
    private static ArrayList<Integer> cache = new ArrayList<Integer>();

    public static void main(String[] args) {
        cache.add(0, 0);
        cache.add(1, 1);
    }

    public static int fib(int n) {
        if (cache.contains(n)) { // our base case is now "has fib(n) already calculated?"
            return cache.get(n); // if so, return that calculated value
        }
        /* else (n > 1) */
        int result = fib(n - 1) + fib(n - 2);
        cache.add(n, result);    // haven't calculated fib(n) before? store it
        return result;
    }
}
```

Example: Palindrome

Palindromes are strings that are the same forwards and backwards

we'll assume ours don't contain any spaces, all lowercase

e.g., "a", "i", "mom", "tat", "did", "anna", "", "racecar", "amanaplanacanalpanama"

1. Smallest value case(s): strings of length 0 or 1 are palindromes; strings where the first and last chars do not match are **not.**
2. Consider the string "nn". Assume we have a method to determine whether or not "nn" is a palindrome.
3. Solve whether or not "a**a" is a palindrome

```
public static boolean palindrome(String s) {
    if (s.length() == 0 || s.length() == 1){
        return true;
    } else {
        boolean result = false;
        if (s.charAt(0) ==
                s.charAt(s.length()-1))
            result = palindrome(
                s.substring(1, s.length()-1));
        return result;
    }
}
```

Exercises: Recursion

Mersenne Numbers

$f(1) = 1, f(n) = 2 \cdot f(n\text{-}1) + 1$

e.g., f(2) = 3, f(3) = 7, f(4) = 15

Sum values in an int array

e.g., input = [12, 3, 42, 77, 9, 101]

Convert a number in base 10 to base 2

e.g. input = 227

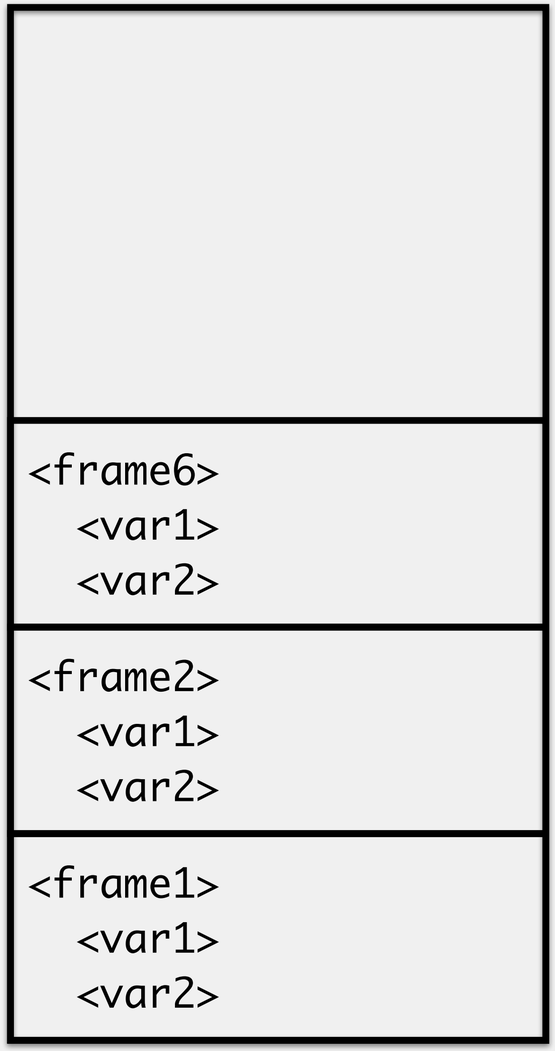
hints: look at the patterns in how the anagrams are arranged; how might you use a second method to help?

Memory Management Revisited

stack: tracks where the program's current location in execution originated and values of local primitive variables (aka *static memory*)

heap: stores global variables, and objects (aka *dynamic memory*)

The Stack Revisited



Bounded in size by the compiler

- can be adjusted
- filling the stack produces a `StackOverflowError`

Faster to access data on than the heap

Error

Thrown when the stack fills up

Usually produced by runaway recursion

- i.e., by an incorrect/lack of base case

Errors **cannot** be recovered from

- must correct program, restart

Why don’t we get this error with infinite loops?

```
public static void printAndIncIter(int num) {

    do {
        System.out.println(num);
        num++;
    } while (true);
}
```

```
public static void printAndIncRecur(int num) {

    System.out.println(num);
    printAndIncRecur(num + 1);
}
```