Recursion

- Idea
- A few examples
- “wishful thinking” method
- Recursion in classes
- Ex: palindromes
- Helper functions
- Computational complexity of recursive functions
- Recursive functions with multiple calls

Announcements

- PA2 due Wed.
- Reminder: regrading policy: regrades must be initiated within a week of when you get the work back.
- Lab this week: recursion problems on codingbat.com (see lab description) (no Vocareum)

What is recursion?

- A *recursive* function is one which calls itself.
- multiple calls are active at the same time.
- recursion instead of a loop.
- Sometimes the recursive solution is shorter and more elegant.
- To get the code right it helps to “think recursively”.

Characteristics of recursive functions:

- some kind of conditional. Two cases:
  - Base case (stopping case): smallest version of problem

    Don't make recursive call.
  - Recursive case

    Calls itself (one or more times) with
    a smaller version of the problem.
- Each recursive call brings the computation closer to the base case.

Example: writeStars (code)

```
// writes n asterisks on a line
// PRE: n >= 0
public static void writeStars(int n) {
  if (n==0) {
    System.out.println();
  }
  else {
    System.out.print("*");
    writeStars(n-1);
  }
}
```

Example: writeStars (thinking recursively)

- Given the specification for the function:

```
// writes n asterisks on a line
// PRE: n >= 0
public static void writeStars(int n)
```

- So what does the following call do:

```
writeStars(n-1);
```

Coming up with a recursive solution: the "wishful thinking" method

1. Decide what would make the problem simpler.

2. Pretend you already know how to solve the simpler version of the problem.

3. How can we use the solution to the simpler problem to construct a solution to the original problem?

Example: exponentiation

- How can we compute b^n in terms of a smaller version of the problem?

… in other words …

- If we already know the answer to b^{n-1}, how can we use that to compute b^n?

Example: exponentiation (cont.)

- A recursive function that returns a value

```
// compute b^n
// PRE: n >= 0
public static int expon(int b, int n) {
  if (n == 0) {
    return 1;
  else {
    return b * expon(b,n-1);
  }
|
```

Example: printNto1

- Print the numbers from n *down to* 1
- Example calls:

```
printNto1(3);       // writes 3 2 1
printNto1(1);       // writes 1
```

- Header:

```
// PRE: n >= 0
public static void printNto1(int n)
```

Example: print1ToN

- Print the numbers from 1 *up to* n
- Example calls:

```
print1ToN(3);       // writes 1 2 3
print1ToN(1);       // writes 1
```

Ex: Palindromes

- palindromes read the same in both directions, e.g.:
  - radar
  - Madam, I'm Adam
  - A man, a plan, a canal. Panama!
- Simpler version: single-word palindromes, all lower case.
  - tenet
  - radar
  - deleveled
  - deified
  - racecar
  - naan
- Method to tell us: Is this word a palindrome?

recursive isPalindrome

```
// returns true iff word is a palindrome
// (reads the same forwards and backwards)
public static boolean isPalindrome(String word)
```

Helper (or Auxiliary) Functions

- Sometimes the function the client wants doesn't match the parameters needed to do the job recursively.
- Ex1: a palindrome function that doesn't make a new substring for every recursive call.

Second version of isPalindrome

Counting steps for recursive functions

- How to figure out computational complexity?

(i.e., big-O)

- Count number of steps in one instance of the function …
- … and multiply this by…
- Number of recursive calls total to solve problem of size n
- Let's do this for some of the examples so far…

Recursive functions with multiple calls

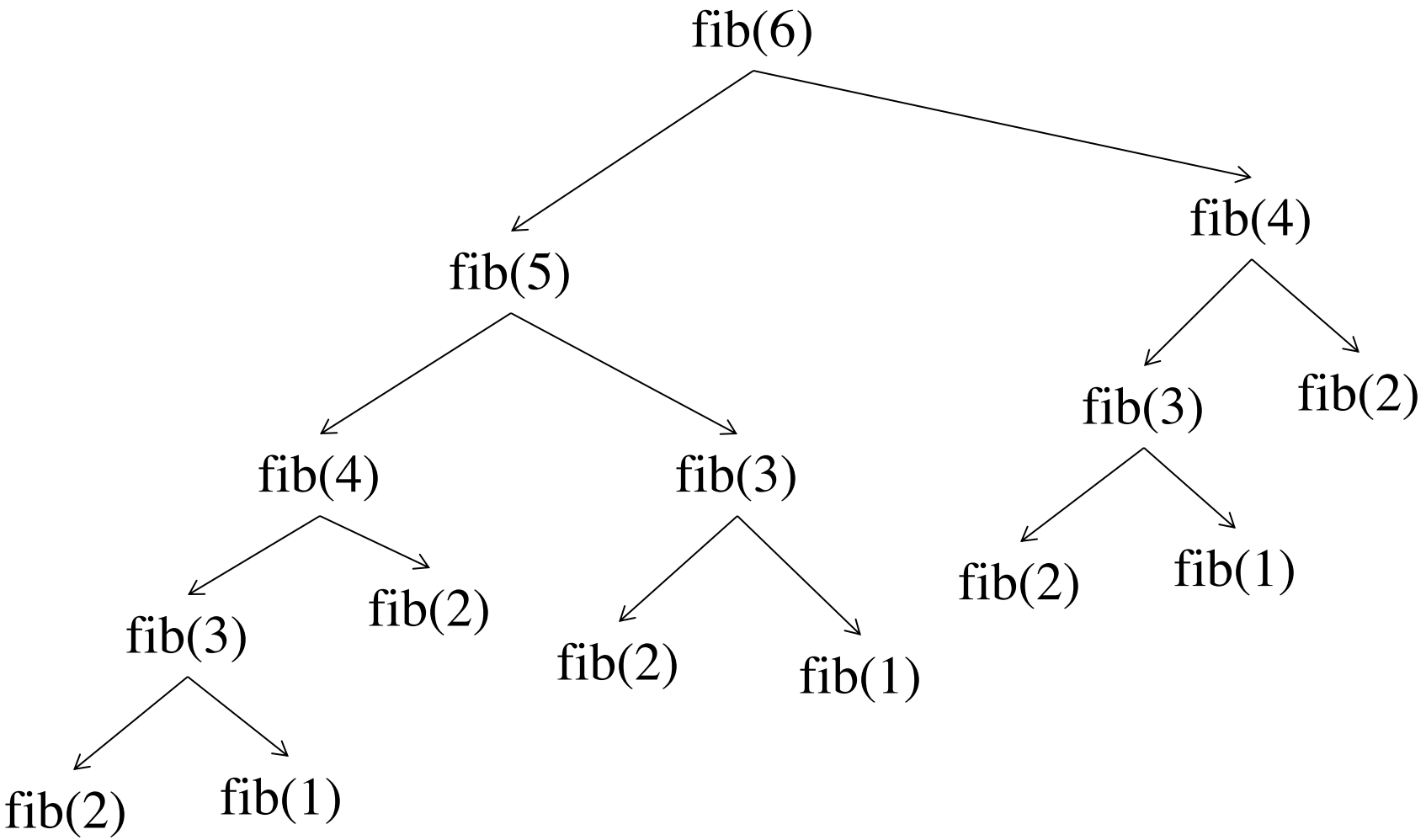
- Simple ex: Fibonacci numbers
- Mathematical definition:

  $f_1 = 1$

  $f_2 = 1$

  $f_n = f_{n-1} + f_{n-2}$
- straightforward translation into a recursive function
- fib(n) returns the n^{th} Fibonacci number

Call tree of fib(6)



Recursive `fib` is SLOOOWWWW…

- Much duplicated work
- Exponential time! ($O(2^n)$)
- Space complexity of recursive solution is only O(n) (depth of call tree).
- Iterative solution?
- Not all recursive functions with multiple calls are bad – we'll see some more in later lectures.

Don't make two **identical** recursive calls

- Similarly . . .

```
if (isPalindrome(smallerString)) {
   . . .
}
else if (!isPalindrome(smallerString)) {
   . . .
}
```

- Is much slower than . . .

```
if (isPalindrome(smallerString)) {
   . . .
}
else {
   . . .
}
```

Using recursion in a Java `class`

- Often can't do recursion on the object itself.
- Usually recursive function will be a static helper function (i.e., private).
- It may operate on a recursive data structure (we'll see some later)

Summary

- This week's examples simple, to understand the concept:
  - Loop versions roughly as easy
  - Same big-O times for both
  - O(n) space for these recursive versions
- Can’t ignore big-O:
  - e.g., recursive fib much worse than alternative.
- Later lectures, assignments:
  - Problems where recursive solution is much simpler.
  - not less space or time efficient than loop solution.