Recursive Methods and Problem Solving

Chris Kiekintveld
CS 2401 (Fall 2010)
Elementary Data Structures and Algorithms
Review: Calling Methods

```java
int x(int n) {
    int m = 0;
    n = n + m + 1;
    return n;
}

int y(int n) {
    int m = 1;
    n = x(n);
    return m + n;
}
```

What does \( y(3) \) return?
Calling Methods

- Methods can call other methods
- Can a method call itself?
- Yes! This is called a recursive method (function)
- “A method within a method”
Example

```java
void test(int n) {
    if (n > 0) {
        System.out.println(n);
        test(n-1);
        System.out.println(n);
    }
}
```

Trace the execution of test(4)
Example (pt 2)

```java
void test(int n) {
    if (n > 0) {
        System.out.println(n);
        test(n-1);
        System.out.println(n);
    }
}
```

Trace the execution of test(-4)
Terminology

- A recursive method is any method that calls itself

- Base case
  - VERY IMPORTANT
  - Stops the recursion (prevents infinite loops)
  - Solved directly to return a value without calling the same method again
Recursive Definitions

- Directly recursive: method that calls itself
- Indirectly recursive: method that calls another method and eventually results in the original method call
- Tail recursive method: recursive method in which the last statement executed is the recursive call
- Infinite recursion: case where every recursive call results in another recursive call
Tracing a Recursive Method

- As always, go line by line
- Recursive methods may have *many copies*
- Every method call creates a new copy and transfers flow of control to the new copy
- Each copy has its own:
  - code
  - parameters
  - local variables
Tracing a Recursive Method

- After completing a recursive call:
  - Control goes back to the calling environment
  - Recursive call must execute completely before control goes back to previous call
  - Execution in previous call begins from point immediately following recursive call
Factorial Numbers

- Factorial numbers (i.e., $n!$) defined recursively:
  - $\text{factorial}(0) = 1$
  - $\text{factorial}(n+1) = \text{factorial}(n) \times (n+1)$

- Examples
  - $0! = 1$
  - $1! = 1 \times 1 = 1$
  - $2! = 2 \times 1 \times 1 = 2$
  - $3! = 3 \times 2 \times 1 \times 1 = 6$
  - $4! = 4 \times 3 \times 2 \times 1 \times 1 = 24$
Iterative Factorial Method

```
public static int fact(int num) {
    int tmp = 1;
    for (int i = 1; i <= num; i++) {
        tmp *= i;
    }
    return tmp;
}
```

Trace fact(5)
Recursive Factorial Method

```java
public static int fact(int num) {
    if (num == 0)
        return 1;
    else
        return num * fact(num - 1);
}

Trace fact(5)
```
Recursive Factorial Trace

Figure 14-1  Execution of the expression fact(4)
Recursion or Iteration?

- **Moral:** There is usually more than one way to solve a problem!
  - Iteration (loops to repeat code)
  - Recursion (nested function calls to repeat code)
- Tradeoffs between two options:
  - Sometimes recursive solution is easier
  - Recursive solution is often slower
Today’s Topic: Recursion (Continued)
Facts about Recursion

- Recursive methods call themselves
- Each call solves an identical problem
  - The code is the same!
  - Successive calls solve smaller/simpler instances
- Every recursive algorithm has at least one base case
  - A known/easy to solve case
  - Often, when we reach 1 or 0
Designing Recursive Algorithms

- General strategy: “Divide and Conquer”
- Questions to ask yourself
  - How can we reduce the problem to smaller version of the same problem?
  - How does each call make the problem smaller?
  - What is the base case?
  - Will you *always* reach the base case?
Similarities

- Closely related to recursive definitions in math
- Also closely related to proof by induction
  - Inductive proofs work in “reverse”
  - Start by proving a base case
  - Then show that if it is true for case n, it must also be true for case n+1
Exercise

Write a recursive function that prints the numbers 1…n in descending order:

```java
public void descending(int n) {
}
```
Exercise

Write a recursive function that prints the numbers 1…n in descending order:

```java
public void descending(int n) {
    if (n <= 0) return;
    System.out.println(n);
    descending(n-1);
}
```
Exercise

Write a recursive function to perform exponentiation
return \( x^m \), assuming \( m \geq 0 \)

```java
public int exp(int x, int m) {
    // implementation goes here
}
```
Exercise

Write a recursive function to perform exponentiation
return \(x^m\), assuming \(m \geq 0\)

```java
public int exp(int x, int m) {
    if (m == 0) { return 1; }
    if (m == 1) { return x; }
    return x * exp(x, m-1);
}
```
public static boolean p(String s, int i, int f) {
    if (i < f) {
        if (s[i] == s[f]) {
            return p(s, i+1, f-1);
        } else {
            return false;
        }
    } else {
        return true;
    }
}

What does p(s, 0, s.length-1) return
a) if s ="UTEPEC"
b) if s ="SAMS"
c) if s ="kayak"
d) if s = "ABBA"
Towers of Hanoi

- The legend of the temple of Brahma
  - 64 golden disks on 3 pegs
  - The universe will end when the priest move all disks from the first peg to the last
The Rules

- Only move one disk at a time
- A move is taking one disk from a peg and putting it on another peg (on top of any other disks)
- Cannot put a larger disk on top of a smaller disk
- With 64 disks, at 1 second per disk, this would take roughly 585 billion years
Towers of Hanoi: Three Disk Problem

Figure 14-6  Tower of Hanoi problem with three disks
Towers of Hanoi: Three Disk Solution

Move disk 1 from needle 1 to needle 3

Move disk 2 from needle 1 to needle 2

Move disk 1 from needle 3 to needle 2

Move 1

Move 2

Move 3
Towers of Hanoi: Three Disk Solution

Move disk 3 from needle 1 to needle 3

Move disk 1 from needle 2 to needle 1

Move disk 2 from needle 2 to needle 3

Move disk 1 from needle 1 to needle 3

Move disk 7 from needle 1 to needle 3

Move disk 5 from needle 2 to needle 1

Move disk 1 from needle 3 to needle 2

Move disk 4 from needle 1 to needle 3

Move disk 2 from needle 3 to needle 1

Move disk 6 from needle 3 to needle 1

Move disk 1 from needle 2 to needle 3

Move disk 3 from needle 2 to needle 1
Four disk solution (courtesy wikipedia)
Recursive algorithm idea

- Final step is to move the bottom disk from peg 1 to peg 3
- To do this, the other n-1 disks must be on peg 2
- So, we need an way to move n-1 disks from peg 1 to peg 2
- Base case: moving the smallest disk is easy (you can always move it to any peg in one step)
Pseudocode

solveTowers (count, source, destination, spare) {
    if (count == 1) {
        move directly
    }
    else {
        solveTowers(count-1, source, spare, destination)
        solveTowers(1, source, destination, spare)
        solveTowers(count-1, spare, destination, source)
    }
}
Today’s Topic: Recursion (Continued)
Recursive Fibonacci

\[ rFibNum(a,b,n) = \begin{cases} 
  a & \text{if } n = 1 \\
  b & \text{if } n = 2 \\
  rFibNum(a,b,n-1) + rFibNum(a,b,n-2) & \text{if } n > 2.
\]
Recursive Fibonacci (continued)

```java
public static int fib(int n) {
    if (n <= 2) {
        return 1;
    }
    return fib(n - 1) + fib(n - 2);
}
```
Recursive Fibonacci (continued)

Figure 14-5  Execution of the expression rFibNum(2, 3, 5)
Iterative Solution

- Recursive solution repeats many computations, so it is very inefficient. An iterative approach:

```java
public static int fib(int n) {
    if (n <= 2) return 1;
    int f1 = f2 = 1;
    for (int i = 3; i <= n; i++) {
        int tmp = f1 + f2;
        f1 = f2;
        f2 = tmp;
    }
    return f2;
}
```
Improved Recursion

```java
int[] cache = new int[n];
public static int fib(int n) {
    if (cache[n] <= 0) {
        cache[n] = fib(n-2) + fib(n-1);
    }
    return cache[n];
}
```

Bugs?
Improved Recursion (debugged)

```java
int[] cache = new int[n+1];
public static int fib(int n) {
    if (n <= 2) return 1;
    if (cache[n] <= 0) {
        cache[n] = fib(n-2) + fib(n-1);
    }
    return cache[n];
}
```
Search

- Design a method that returns true if element n is a member of array x[] and false if not
- Iterative approach:

```java
public boolean search(int[] x, int n) {
    for (int i = 0; i < x.length; i++) {
        if (x[i] == n) return true;
    }
    return false;
}
```
Recursive Search

- A recursive variant:

```java
boolean search(int[] x, int size, int n) {
    if (size > 0) {
        if (x[size-1] == n) {
            return true;
        } else {
            return search(x, size-1, n);
        }
    }
    return false;
}
```
Faster Search

- The problem: these methods are slow
- Recall the phone book example
- “Linear search” – need to look at every element
- “Binary search” is much faster on sorted data
Binary Search Pseudocode

search(phonebook, name) {
    if only one page, scan for the name
    else
        open to the middle
        determine if name is before or after this page
        if before
            search (first half of phonebook, name)
        else
            search (second half of phonebook, name)
boolean binarySearch(int[] x, int start, int end, int n) {
    if (end < start) return false;
    int mid = (start+end) / 2;
    if (x[mid] == n) {
        return true;
    } else {
        if (x[mid] < n) {
            return search(x, mid+1, end, n);
        } else {
            return search(x, start, mid-1, n);
        }
    }
}
Programming Example

```java
int test(String s, int last) {
    if (last < 0) {
        return 0;
    }
    if (s.charAt(last) == "0") {
        return 2 * test(s, last-1);
    }
    return 1 + 2 * test(s, last-1);
}
```

Trace test("01101", 4)
What does method test do?
Exercise

Write a recursive function convert a decimal number into a binary number, printing the binary number

```java
public static void decToBin(int num) {
}
```
Exercise
Decimal to Binary

```java
public static void decToBin(int num) {
    if (num > 0) {
        decToBin(num / 2);
        System.out.print(num % 2);
    }
}
```