Comp 152

Recursion
Solving problems

- In Math
  - many problems are solved using smaller versions of the same problem
  - Fibonacci numbers
- In Philosophy
  - inductive proofs
  - show base case is true
  - show that each later case follows from simple step and earlier case
Use the same technique in CS

• We can use the same inductive technique in our programming
  • recursion.
Simple example

- Triangle numbers
  - Compute the area of a triangle of width \( n \)
  - Assume each [] square has an area of 1
  - Also called the \( n \)th triangle number
  - The third triangle number is 6
  - []
  - []
  - [][]
  - [][][]
Class to solve: OutLine

- public class Triangle{
  - public Triangle(int aWidth){
    - width = aWidth;
  - }
  -
  - private int width;
  - }

  - what really interesting method is glossed over here?
The base – easy case

- First check the easy case
  - The triangle consists of a single square
    - Its area is 1
  - Add the code to getArea method for width 1
    - public int getArea()
    - {
    -   if (width == 1) return 1;
    -   ...
    - }
the General Case

• Remember: A small change and an easier case
  
  • Assume we know the area of the smaller, colored triangle
  
  - []
  - [][]
  - [][]
  - [][]
  - [][]

• Area of larger triangle can be calculated as:
  
  - smallerArea + width

• so how do we calculate the smallerArea?
• To get the area of the smaller triangle
• Make a smaller triangle and ask it for its area
• Triangle smallerTriangle = new Triangle(width - 1);
• int smallerArea = smallerTriangle.getArea();
The complete getArea Method

- public int getArea()
- {
-     if (width == 1) return 1;
-     Triangle smallerTriangle = new Triangle(width - 1);
-     int smallerArea = smallerTriangle.getArea();
-     return smallerArea + width;
- }
Walkthrough of width 4

- getArea method makes a smaller triangle of width 3
  - It calls getArea on that triangle
    - That method makes a smaller triangle of width 2
      - It calls getArea on that triangle
        - That method makes a smaller triangle of width 1
        - It calls getArea on that triangle
        - That method returns 1
    - The method returns smallerArea + width = 1 + 2 = 3
  - The method returns smallerArea + width = 3 + 3 = 6
- The method returns smallerArea + width = 6 + 4 = 10
Recursion in Programming

- A recursive computation solves a problem by using the solution of the same problem with simpler values

- For recursion to terminate,
  - there must be special cases for the simplest inputs.
  - To complete our Triangle example, we must handle width $\leq 0$
    - `if (width <= 0) return 0;`
  - Two key requirements for recursion success:
    - Every recursive call must simplify the computation in some way
    - There must be special cases to handle the simplest computations directly
There are other ways to compute it

- The area of a triangle equals the sum
  - $1 + 2 + 3 + \ldots + \text{width}$
- Using a simple loop:
  - `double area = 0;`
  - `for (int i = 1; i <= width; i++)`
  - `area = area + i;`
- Using math:
  - $1 + 2 + \ldots + n = n \times (n + 1)/2$
  - $=> \text{width} \times (\text{width} + 1) / 2$
- but recursive power useful for many types of problems
a nice problem for recursion.

- Problem: test whether a sentence is a palindrome
  - Palindrome: a string that is equal to itself when you reverse all characters
    - A man, a plan, a canal–Panama!
    - Go hang a salami, I'm a lasagna hog
    - Madam, I'm Adam
  - how would you design a recursive solution to this problem?
Sample code minus solution

- public class Sentence{
-   public Sentence(String aText){
-     text = aText;
-   }
-   public boolean isPalindrome() {
-     ...
-   }
-   private String text;
- }
Thinking Recursively

• We need
  • a base case
  • a simpler version of the problem.

• Consider various ways to simplify inputs
  • Here are several possibilities:
    - Remove the first character
    - Remove the last character
    - Remove both the first and last characters
    - Remove a character from the middle
    - Cut the string into two halves

• do any sound good?
Recursive solutions: Simplification

- Combine solutions with simpler inputs into a solution of the original problem
  - Most promising simplification: remove first and last characters
    - "adam, I'm Ada", is a palindrome too!
  - Thus, a word is a palindrome if
    - The first and last letters match, and
    - Word obtained by removing the first and last letters is a palindrome
Simplification scenarios

- What if first or last character is not a letter?
  - Ignore it
- If the first and last characters are letters, check whether they match;
  - if so, remove both and test shorter string
  - If last character isn't a letter, remove it and test shorter string
  - If first character isn't a letter, remove it and test shorter string
Base Cases

- **Find solutions to the simplest inputs**
  - **Strings with two characters**
    - No special case required; step two still applies
  - **Strings with a single character**
    - They are palindromes
  - **The empty string**
    - It is a palindrome
So let's write it.

• on the board.
Recursive Efficiency I

- That was a lot of new String objects
  - more than really needed.
- check if substring is palindrome
- public boolean isPalindrome(int start, int end)
Questions! Hooray!!

- Do we have to give the same name to both isPalindrome methods?
  - is it a good idea?
  - what is it called?
- When does the recursive isPalindrome method stop calling itself?
Fibonacci

- I talked about this before – but how could we have a recursive Fibonacci solution?
- Recall
  - **Fibonacci sequence is a sequence of numbers defined by**
    - $f(1) = 1$
    - $f(2) = 1$
    - $f(n) = f(n-1) + f(n-2)$
    - First ten terms
      - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55
How Do we produce a recursive fib calculator?

- What are the base cases?
- What about the recursive case?
- new wrinkle here.
Typical Recursive Solution

- /** Computes a Fibonacci number.
-   @param n an integer
-   @return the nth Fibonacci number
- */
- public static long fib(int n)
  {
    if (n <= 2) return 1;
    else return fib(n - 1) + fib(n - 2);
  }
Efficiency of Recursion II

• Recursive implementation of fib is straightforward
• if you ran test program you would see:
  • First few calls to fib are quite fast
  • For larger values, the program pauses an amazingly long time between outputs
• See chart next slide
Recursive Call tree for fib(6)

Do you see the problem?

![Call Pattern of the Recursive fib Method](image)

Figure 2 Call Pattern of the Recursive fib Method
Efficiency of Recursion III

- Occasionally, a recursive solution runs much slower than its iterative counterpart
  - In most cases, the recursive solution is only slightly slower
  - The iterative isPalindrome performs only slightly better than recursive solution
  - Each recursive method call takes a certain amount of processor time
• Smart compilers can avoid recursive method calls if they follow simple patterns
• Most compilers don't do that
• In many cases,
  • a recursive solution is easier to understand and implement correctly than an iterative solution
  • easy to understand == easier maintainance
    – usually better for much of your program.
• "To iterate is human, to recurse divine.", L. Peter Deutsch
Only one question? Say it isn't so!

- You can compute the factorial function either with a loop, using the definition that $n! = 1 \times 2 \times \ldots \times n$, or recursively, using the definition that $0! = 1$ and $n! = (n - 1)! \times n$. Is the recursive approach inefficient in this case?
“Mutually Recursive functions”

- Sometimes have solution that is not direct recursion
  - method a calls method b
  - method b called method a
  - still have to make sure each has
    - base case
    - simplified case.
Old Favorite

- Binary Search
  - Describe
  - Solve with students.
File / directory traversal

• Show directory traveling
  • Theoretical approach.
Reading

- Read Chapter 14 in Starting out with Java