COMP 202
Recursion

CONTENTS:

• Recursion
• Recursion vs Iteration
• Indirect recursion
• Runtime stacks
Recursive Thinking

• A *recursive definition* is one which uses the word or concept being defined in the definition itself
  – GNU
    • Gnu's Not Unix
  – LAME
    • Lame Ain't an MP3 Encoder
Recursive Definitions

• Consider the following list of numbers:
  
  24, 88, 40, 37

• Such a list can be defined as

  A LIST is a: number
  or a: number comma LIST

• That is, a LIST is defined to be a single number, or a number followed by a comma followed by a LIST

• The concept of a LIST is used to define itself
Recursive Definitions

• The recursive part of the LIST definition is used several times, terminating with the non-recursive part:

```
24 , 88, 40, 37
```

```
88 , 40, 37
```

```
40 , 37
```

```
37
```
Infinite Recursion

- All recursive definitions have to have a non-recursive part
- If they didn't, there would be no way to terminate the recursive path
- Such a definition would cause infinite recursion
- This problem is similar to an infinite loop
- The non-recursive part is often called the base case
Recursive Definitions

• N!, for any positive integer N, is defined to be the product of all integers between 1 and N inclusive.

• This definition can be expressed recursively as:

\[
\begin{align*}
1! &= 1 \\
N! &= N \times (N-1)! 
\end{align*}
\]

• The concept of the factorial is defined in terms of another factorial.

• Eventually, the base case of 1! is reached.
Recursive Definitions

\[ 5! = 5 \times 4! = 4 \times 3! = 3 \times 2! = 2 \times 1! \]

\[ 120 \]
\[ 24 \]
\[ 6 \]
\[ 2 \]
\[ 1 \]
Recursive Programming

• A method in Java can invoke itself; if set up that way, it is called a *recursive method*. 

• The code of a recursive method must be structured to handle both the base case and the recursive case.

• Each call to the method sets up a new execution environment, with new parameters and local variables.

• As always, when the method completes, control returns to the method that invoked it (which may be an earlier invocation of itself).
Recursive Programming

- Consider the problem of computing the sum of all the numbers between 1 and any positive integer N

- Sum of 5 = 5 + 4 + 3 + 2 + 1
Recursive Programming

```java
int sum(int n)
{
    int result = 0;
    if (n == 1) // base case
        result = 1;
    else // recursive part
        result = n + sum(n-1);
    return result;
}
```
Recursive vs. Iterative

int sum_recursive(int n)
{
    int result = 0;
    if (n == 1) // base case
        result = 1;
    else if (n > 1) // recursive part
        result = n + sum_recursive(n-1);
    return result;
}

int sum_iterative(int n)
{
    int result = 0;
    for (int i = 1; i <= n; i++)
        result += i;
    return result;
}
Recursive Programming

• Note that just because we can use recursion to solve a problem, doesn't mean we should (there is a lot of overhead: method calls, variable declarations, etc.)
• For instance, we usually would not use recursion to solve the sum of 1 to N problem, because the iterative version is easier to understand
• However, for some problems, recursion provides an elegant solution, often cleaner than an iterative version
• You must carefully decide whether recursion is the correct technique for any problem
public class PalindromeTesters {

    public static boolean iterativeTester (String str) {
        boolean result = false;
        int left = 0;
        int right = str.length() - 1;

        while (left < right && str.charAt(left) == str.charAt(right)) {
            left++;
            right--;
        }

        if (left >= right) result = true;
        return result;
    }

    public static boolean recursiveTester (String str)  {
        boolean result = false;

        if (str.length() <= 1) result = true;
        else result = (str.charAt(0) == str.charAt(str.length() - 1)) &&
                      recursiveTester(str.substring(1,str.length()-1));

        return result;
    }
}

Palindrom Testing
When to use recursion…

• Notice that we have many ways to iterate:
  – Do…while
  – While
  – For
  – Recursion

• They all do the same thing, so selecting between them should be based on some benefit:
  – Easier to program using that loop
  – Runs faster with that particular loop

• Ideally you want to optimize on both criteria
Designing For Recursion

• Solution requires iteration
• Algorithm always looks like this:
  – Base Case
    • The part of the loop that has the stop condition. It also returns the default (simplest case) result
  – Incrementing Part
    • The part of the program that moves us on to the next data value.
      – Incrementing variable
      – Reading data
      – Moving to a new data item in a structure (like array)
  – Recursion Part
    • The part of the program that initiates the iteration
• Note that the Incrementing and Recursion Parts are often together in the same statement (but not always so)
Indirect Recursion

- A method invoking itself is considered to be *direct recursion*

- A method could invoke another method, which invokes another, etc., until eventually the original method is invoked again

- For example, method \( m_1 \) could invoke \( m_2 \), which invokes \( m_3 \), which in turn invokes \( m_1 \) again

- This is called *indirect recursion*, and requires all the same care as direct recursion

- It is often more difficult to trace and debug
Indirect Recursion

m1 ← m2 ← m3

m1 → m2 → m3
Part 2

The Run-Time Stack
An Executing Program in RAM

Dynamic Memory

- Run-time Heap: used for dynamic data (more next week)
- Run-time Stack: stores local variables and function call management

Static Code

Your compiled code

Static Data

All non-local data and all literals
Function Call “Frame”

- At every call to a function a frame is added to the TOP of the stack. This is referred to as a PUSH.
- When the function terminates the frame is removed from the top of the stack. This is referred to as a POP.
- Stacks function much like a stack of plates. You put them on the top and you remove them from the top.
Problem

Write the factorial program recursively and then construct the run-time stack. Write a main method that invokes the method factorial. Now draw the run-time stack from the moment the main method is invoked to the moment the main method terminates. Show how it updates and how it produces the correct results.