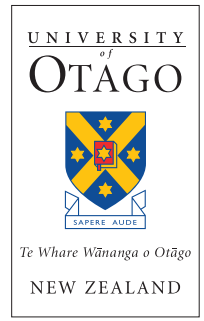
Cosc 241
Programming and Problem Solving
Lecture 4 (5/3/20)
Recursion

Michael Albert
michael.albert@cs.otago.ac.nz

Keywords: recursion, iteration



What happens if?

- We are accustomed to writing programs with methods that call other methods.
- What would happen if a method called itself?

As a general concept, recursion is the process of defining something in terms of itself.

The problem is to avoid circularity which in programming terms leads to infinite loops, or non-termination.

Recursive definition

A string is either:

- empty, or
- a character followed by another string.

Despite the fact that the definition of string uses the concept of string, there is no circularity, because of the ‘basic’ non-recursive case (the empty string).

In a general recursive definition objects are defined as either:

- certain basic objects (non-recursive case), or
- formed from simpler objects by certain rules (recursive case).

How to grade a stack of exam papers recursively

if the stack is empty **then**
    Stop! Have fun!
else
    Grade one paper
    Grade the remaining stack of papers
end if

- Here we are applying a recursive method
- Again there is a basic non-recursive case (a stack of 0 papers).
- And a recursive case ('grade the remaining stack') which is simpler because the stack is one smaller.

Try this

- How does recursion arise in evaluating a complicated mathematical expression like:

$$3 + 5 \times (23 - 120 \div (2 + 6)) + 117$$

- Imagine you have to use a physical dictionary. How does recursion arise in searching for a word like “sesquipedalian”?
- Are there any day-to-day activities that you think of recursively?

Recursion in mathematics

$$\begin{aligned} 0! &= 1 \\ n! &= n \times (n-1)! \quad \text{for } n > 0 \end{aligned}$$

This defines the factorial function in terms of itself – is that a problem? No, we can trace a computation:

$$\begin{aligned} 3! &= 3 \times 2! \\ &= 3 \times 2 \times 1! \\ &= 3 \times 2 \times 1 \times 0! \\ &= 3 \times 2 \times 1 \times 1 \\ &= 6. \end{aligned}$$

Factorial in Java

```
public class FacRec{

   public static void main(String[] args) {
      int n = Integer.parseInt(args[0]);
      System.out.println(n + "! = " + factorial(n));
   }

  public static long factorial(int n) {
      if (n == 0) return 1;
      return n*factorial(n-1);
   }

}
```

This works!

Why not?

```
public class FacIt{

   public static void main(String[] args) {
      int n = Integer.parseInt(args[0]);
      System.out.println(n + "! = " + factorial(n));
   }

  public static long factorial(int n) {
      long fac = 1;
      for(int i = 1; i <= n; i++)
         fac *= i;
      return fac;
   }

}
```

Because this is a lecture on recursion!

So how does it work?

- Method calls get added to a stack.
- Whenever a method calls another one (or itself), its state is stored, execution is temporarily suspended, and the new method begins execution.
- When a `return` statement (in any method) is encountered, that method is removed from the stack and the value returned (if any) supplied to its caller.
- There is some overhead involved.
- More on stacks (in general) later!

Recursion in game playing

- Managing a two player game is a good example of where recursion can simplify code.
- "Run the game" means "check whether the game is over" (base case), or "do a turn and change players" then "run the game"
- Loop structures often turn out to be awkward but this basic play control works in any game with alternating players (and can be modified to allow for more than two players).

Play(*g*, *currentPlayer*, *nextPlayer*):
  if the game is over **then**
    Announce the winner. Halt/return.
  end if
  Get a move in *g* from *currentPlayer*
  Apply the move to *g* (which changes *g* to a simpler game)
  Play(*g*, *nextPlayer*, *currentPlayer*)

Towers of Hanoi

- A puzzle invented by French mathematician Édouard Lucas in 1883.
- There are three rods and a set of discs of differing sizes.
- Initially, the discs are on a single rod in order of size, smallest at the top.
- The problem is to move them all to one of the other rods, shifting one disc at a time, and never placing a larger disc on top of a smaller one.

A plan

- Call the rods A, B and C, and the discs 1, 2, . . . , n (with 1 being the smallest).
- The object is to move all the discs from A to B.
- What has to be the case when we move disc n?
- All the other discs must be on C.
- That is, we have to solve the $n-1$ disc problem from A to C first, then move disc n, then solve the $n-1$ disc problem from C to B.
- That's recursion!

An algorithm

Move(n, *source*, *dest*, *extra*):
  if $n = 0$ **then**
    return
  end if
  Move($n - 1$, *source*, *extra*, *dest*)
  source $\rightarrow$ *dest*
  Move($n - 1$, *extra*, *dest*, *source*)

An implementation

```
public class Hanoi{

   public static void main(String[] args){
      hanoi(Integer.parseInt(args[0]), "A", "B", "C");
   }

   public static void hanoi(int n,
         String source, String dest, String extra) {

      if (n == 0) return;

      hanoi(n-1, source, extra, dest);
      System.out.println(source + " --> " + dest);
      hanoi(n-1, extra, dest, source);

   }

}
```

Recursion vs iteration

- Every problem that can be solved recursively can also be solved iteratively (at worst, by representing the call stack explicitly).
- But, if a problem, solution, or data structure is naturally described recursively, then a recursive implementation is likely to be simpler, and more likely to be correct.
- In some sense, the default should be to use an iterative solution unless the problem is naturally cast recursively.
- Among our examples, factorial is more naturally done iteratively, but Hanoi should be done recursively.

A cautionary tale

- The Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, 21, ...
- Defined by the rule 'each element is the sum of the preceding two'.
- Mathematically:

$$f_0 = 1,\ f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \quad \text{for } n \geq 2$$

```
Fib(n):
  if n ≤ 1 then
    return 1
  else
    return Fib(n − 1) + Fib(n − 2)
  end if
```

What's the problem?

- All the indicators seem right for recursion.
- An implementation will work fine for small n.
- For $n \sim 40$ it starts to get a bit slow.
- For $n \sim 50$ it takes unacceptably long.
- What's going on? After all, to calculate each successive Fibonacci number only requires one extra addition!
- The problem is that the recursive algorithm doesn't 'know' Fib$(n-2)$ even after it has computed Fib$(n-1)$, so it starts from scratch
- The double recursive call creates an exponential blow up in the number of actual method calls.