Computational Thinking with Algorithms - Recursion

Dominic Carr

Atlantic Technological University

dominic.carr@atu.ie

“Begin at the beginning,”” the King said, very gravely, “and go on till you come to the end: then stop.”

We must repeat ourselves

- Computers automate tasks
  - Very good at doing the same thing again and again in the exact same way.
- We have ways of making code repeat itself with varied input i.e. loops, methods etc.
- Up until now we have mostly considered on iterative approaches
- **Recap:** iteration allows some sequence of steps (or block of code) to be executed repeatedly, e.g. using a for loop or a while loop
- Recursion is *another technique* to complete tasks which are repetitive

Recursion

Definition

"Recursion in computer science is a method where the **solution to a problem depends on solutions to smaller instances of the same problem** (as opposed to iteration). The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science"

Mostly this means that the programming language allows **a method to invoke itself**.

Recursion I

- "Normally" methods call other methods
  - For Example: main() method calls sumNumbers()
- A recursive method is one which calls itself
- For Example: A method factorial() contains a call to itself

Recursion II

```
void count(int index){
    print(index);
    if (index < 2){
        count(index+1);
    }
}
```

- count() calls itself!
- The input is *different*
- There is some way to *stop* i.e. index $>= 2$
- count(0) doesn't finish until count(1) finishes, and count(1) needs count(2) to finish

Recursion III

- Stack
  - A data structure i.e. a data organization, management, and storage format that is usually chosen for efficient data access
  - Works like a stack of plates in a cafeteria There are two operations:
    - Push: put something in
    - Pop: remove the top element
  - This gets used for program instructions
  - When a method finishes it gets popped off the stack
  - When a method is called/invoked it is placed on the program stack

Why use recursion?

- You enjoy hurting your brain.
- We would use recursion where the solution to a problem can be solved by solving smaller instances of the same problem.
  - The pain will stop, I promise
- Some problems lend themselves to recursion e.g.
  - Traversing directories in a file system
  - Traversing a tree data structure
  - Some sorting algorithms are recursive
- (once you get your head around it) recursion often leads to cleaner and more concise code

Recursion Vs. Iteration

- If you can do it with recursion then you can do it iteratively
  - ...and vice versa
- Recursion is "expensive"
  - overhead in method calls
- If both of the above statements are true, why would we ever use recursion?
- In many cases, the extra "expense" of recursion is far outweighed by a simpler, clearer algorithm which leads to an implementation that is easier to code.
- Ultimately, the recursion is eliminated when the compiler creates assembly language (it does this by implementing the stack).
- If the recursion tree has a simple form, the iterative version may be better.
- If the recursion tree appears quite "bushy", with very few duplicate tasks, then recursion is likely the natural solution.

Types of Recursion

- Linear: method makes a single call to itself
- Tail: Method makes a single call to itself as the last operation
- Binary: The method makes two calls to itself
- Exponential: The method makes more than two calls to itself
- Infinite: When we don't have a guard to stop the recursion
  - Circular: sub-type of infinite where the values passed become circular

The Golden Rules of Recursion

- There must always be a **base case** which can be solved without further recursion
- Every recursive call must **make progress** toward the base case

Designing Recursive Algorithms

- Think about the task which you wish to accomplish, and try to identify any recurring patterns, e.g. similar operations that must be conducted, like traversing through nested directories on a file system
- Divide the problem up using these recurring operations
- Then:
  - Identify cases you know can be solved without recursion (base cases). Avoid ending with a multitude of special cases; rather, try to identify a simple base case
  - Invoke a new copy of the method within each recursive step
  - Each recursive step resembles the original, larger problem
  - Make progress towards the base case(s) with each successive recursive step/call

The Fibonacci Sequence I

Definition

"In mathematics, the **Fibonacci numbers** are the numbers in the following integer sequence, called the **Fibonacci sequence**, and characterized by the fact that every number after the first two **is the sum of the two preceding ones**"

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34... \tag{1}$$

The Fibonacci Sequence II

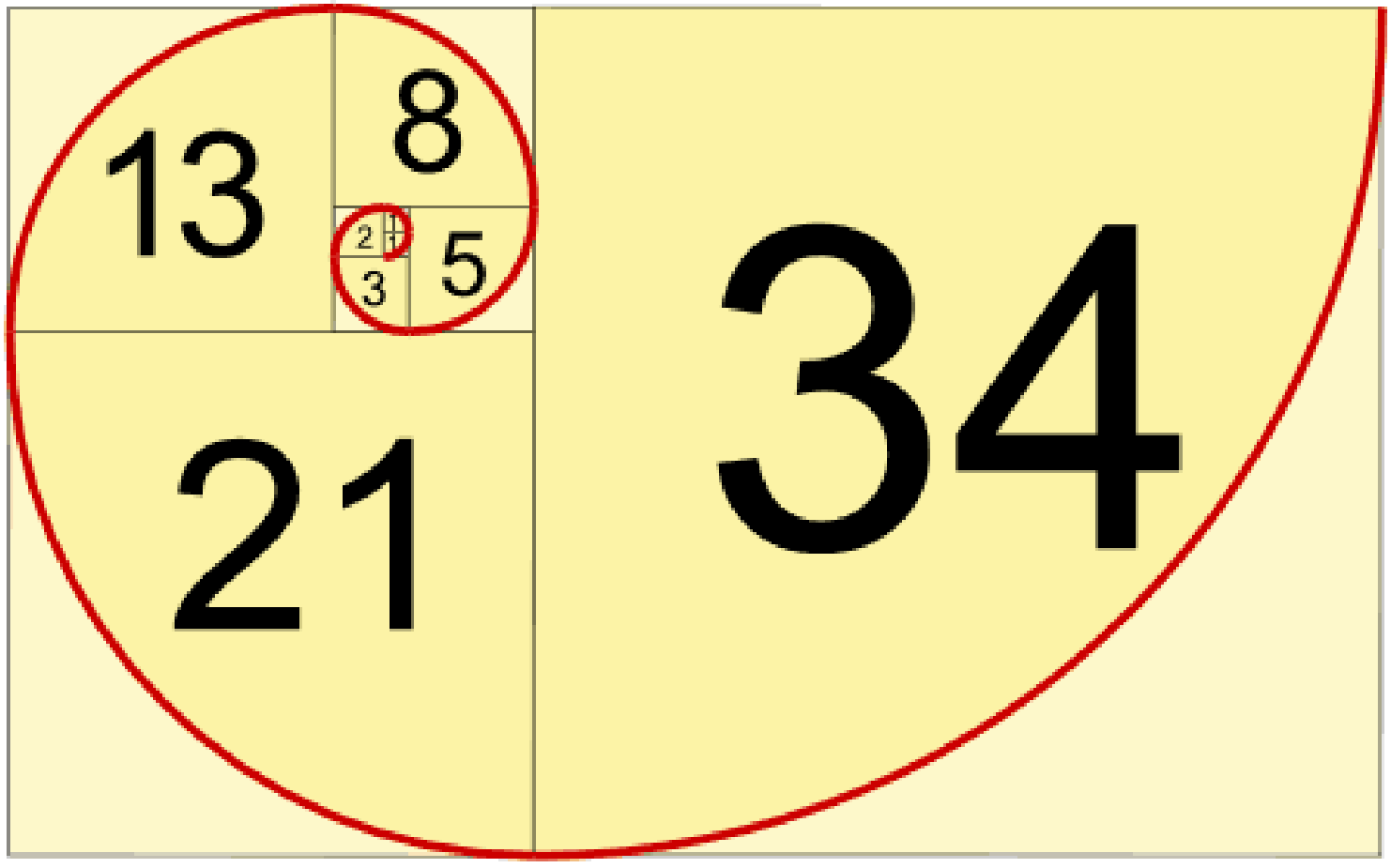

Figure: This is the Fibonacci Sequence

The Fibonacci Sequence III

Listing 1: Algorithm to determine the nth fibonacci number

```
public static int fib(int n) {
    if (n==0)
        return 0;
    int a = 0, b = 1;
    for (int i = 2; i <= n; i++) {
        int c = a + b;
        a = b;
        b = c;
    }
    return b;
}

public static void main(String[] args) {
    for (int i = 0; i < 20; i++) {
        System.out.println("fib("+i+") is " + fib(
```

The Fibonacci Sequence IV

```
        }
    }
```

This is an iterative solution to calculating the nth Fibonacci number, put simply this means that it is a method / function **which loops to repeat some part of the code**. A recursive method **is one which calls itself to repeat the code.**

Listing 2: Output of the program

```
fib(0) is 0
fib(1) is 1
fib(2) is 1 (sum of the previous 2!)
fib(3) is 2
fib(4) is 3
fib(5) is 5
fib(6) is 8
fib(7) is 13
```

The Fibonacci Sequence V

```
fib(8) is 21
fib(9) is 34
fib(10) is 55
fib(11) is 89
```

The Fibonacci Sequence VI

So do we achieve a recursive solution for this problem?

- From the definition of the Fibonacci sequence we know that fib(3) is fib(2) + fib(1)
- So maybe fib(n) calls fib(n-1) and fib(n-2)? **let's give it a go**.

The Fibonacci Sequence VII

- That didn't work!

```
Exception in thread "main" java.lang.StackOverflow
        at fibonaccitest.FibonacciTest.fib(Fibonac
        at fibonaccitest.FibonacciTest.fib(Fibonac
        at fibonaccitest.FibonacciTest.fib(Fibonac
        at fibonaccitest.FibonacciTest.fib(Fibonac
        at fibonaccitest.FibonacciTest.fib(Fibonac
        at fibonaccitest.FibonacciTest.fib(Fibonac
```

- We don't have anything to indicate that we should stop, so the algorithm runs until we run out of space.
- **"In computing, recursion termination is when certain conditions are met and a recursive algorithm stops calling itself and begins to return values"**

The Fibonacci Sequence VIII

For the Fibonacci we can define three conditions:

- if n is 0, returns 0.
- if n is 1, returns 1.
- otherwise, return fib(n-1) + fib(n-2)

Listing 3: Recursive implementation of fib

```
public static int fib(int n) {
    if (n == 0 || n == 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

The Fibonacci Sequence IX

What we have done in this specific case can be applied to all recursive algorithms. As all such algorithms must have the following properties to operate.

- Base Case (i.e. when to stop)
- Work toward Base Case
- Recursive Call

Factorial I

Definition

"In mathematics, the factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n."

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \tag{2}$$

Factorial II

Listing 4: Iterative calculation of the factorial of n

```
    // static means that a method is of the class
    public static int factorial(int n){
    int factorial = 1;
    for (int i = 2; i <= n; i++) {
        factorial *= i;
    }
    return factorial;
}
```

How to do it recursively?

- 1! and 0! are both 1.
- that is a stop condition
- factorial(5) is 5 * factorial(4)

Factorial III

Listing 5: Recursive solution to factorial of N)

```
public static int factorial(int N) {
    if (N == 1) {
        return 1;
    }

    return N * factorial(N - 1);
}
```

The End