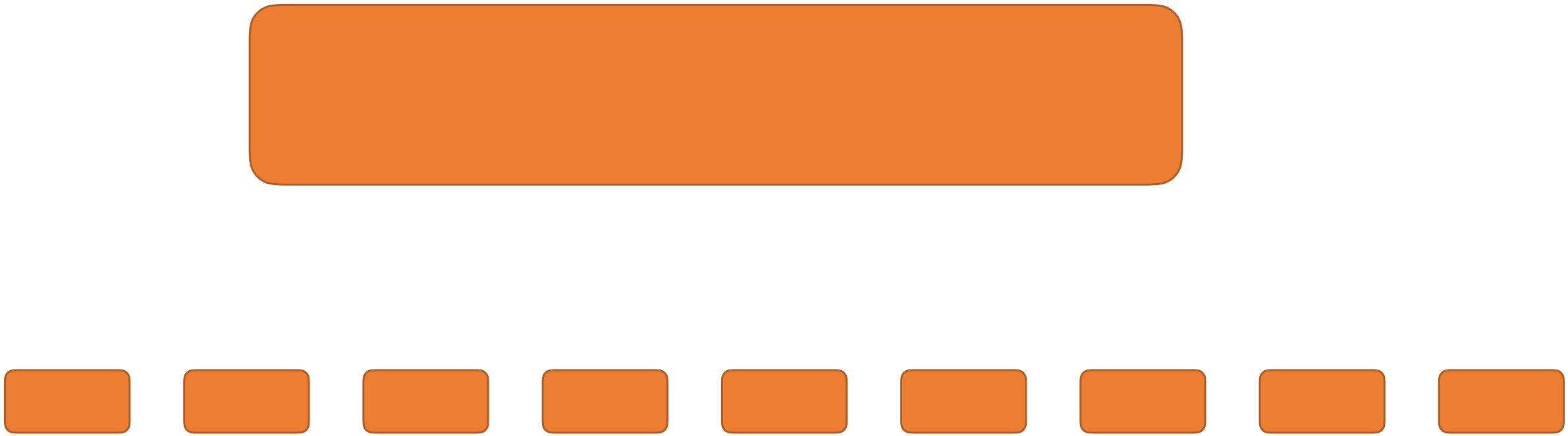
Recursion
I

Mark Matthews

Problem…



When we write an algorithm for solving a particular problem, one of the basic design techniques is to break the task into smaller subtasks.

What on earth is recursion?

Q: What is recursion?

A: When something is defined in terms of itself.

Why learn recursion?

- A new way of thinking
- Powerful programming paradigm
- Gives insights into computation

Many computational artefacts are naturally self-referential.

- File systems with folders within folders
- Fractal graphical patterns
- Divide & conquer algorithms (more later)

Simple addition example

Problem: add consecutive numbers from 1 to N

$$1 + 2 + 3 + \ldots + n = n + [1 + 2 + 3 + \ldots + (n-1)]$$

Example

$$1+2+3+4+5 = 15$$

We could achieve this with a loop...but.....

```
public static int sumL (int n){
                    int res = 0;
                    for(int i = 1; i < n; i++) {
                      res = res + i;

                    }
                    return res;
                  }
```

Loops are so BORING

```
public static int sumL (int n){
                    int res = 0;
                    for(int i = 1; i < n; i++) {
                       res = res + i;

                    }
                    return res;
                }
```

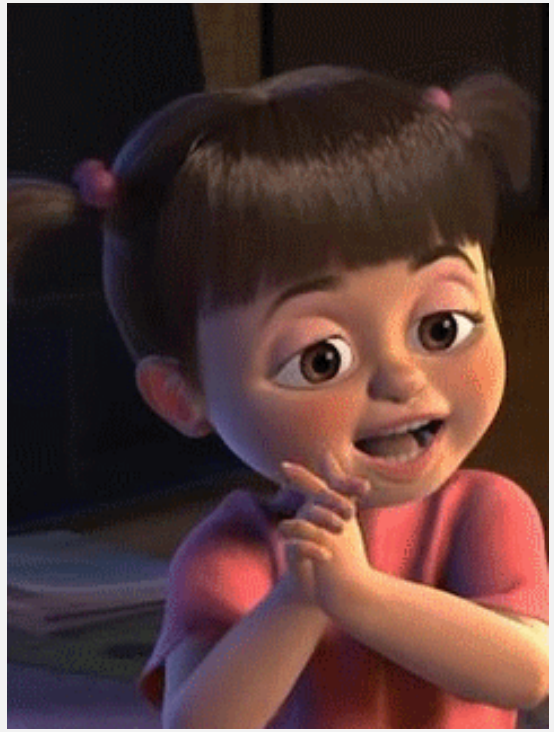
Simple addition example

Problem: add consecutive numbers from 1 to N

$$1 + 2 + 3 + \ldots + n = n + [1 + 2 + 3 + \ldots + (n-1)]$$

$$sumR(n) = n + sumR(n-1)$$

We could do something much cooler....



```
public static int sumR(int n) {
        if(n == 1) {
            return 1;
        }  else {
            return n + sumR(n-1);
    }
    }
```

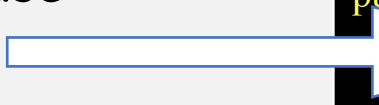
There's an elegance to recursion

```
public static int sumL (int n){
                int res = 0;
                for(int i = 1; i < n; i++) {
                  res = res + i;

                }
                return res;
              }
```

```
public static int sumR(int n) {
                if(n == 1) {
                        return 1;
                } else {
                        return n + sumR(n-1);
        }
        }
```

We could do something much cooler....

Base case



```
public static int sumR(int n) {
        if(n == 1) {
                return 1;
        }  else {
                return n + sumR(n-1);
    }
    }
```

Recursion requires base cases in order to prevent infinite recursion.

Base Case: Metaphor

Recursion and the Stack

```
sumR(5)
    sumR(4)
        sumR(3)
            sumR(2)
                sumR(1)
                    return 1
                return 2 + 1
            return 3 + 2 + 1
        return 4 + 3 + 2 + 1
    return 5 + 4 + 3 + 2 + 1
```

Basic structure of a recursive algorithm?

We can distill the idea of recursion into two simple rules:

1. Each recursive call should be on a smaller instance of the same problem, that is, a smaller subproblem.
2. The recursive calls must eventually reach a base case, which is solved without further recursion.

Recursive Leap of Faith

1. Redefine the problem in terms of a smaller sub-problem (e.g., in SumR we have know how to sum-up n-1 integers.
2. Next, you have to work out how the solution to smaller subproblems will give you a solution to the problem as a whole. (recursive leap of faith)
3. Before using a recursive call, you must be sure that the recursive call will do what it is supposed to. You do not need to think how recursive calls works, just assume that it gives you the correct result.

Hello World of Recursion: Factorial

- We indicate factorial by n!
- It is simply the product of all the integers from 1 to N

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

$$= 120$$

- Useful for trying to count how many different orders there are or different combinations

Hello World of Recursion: Factorial

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1 & n >= 1 \end{cases}$$

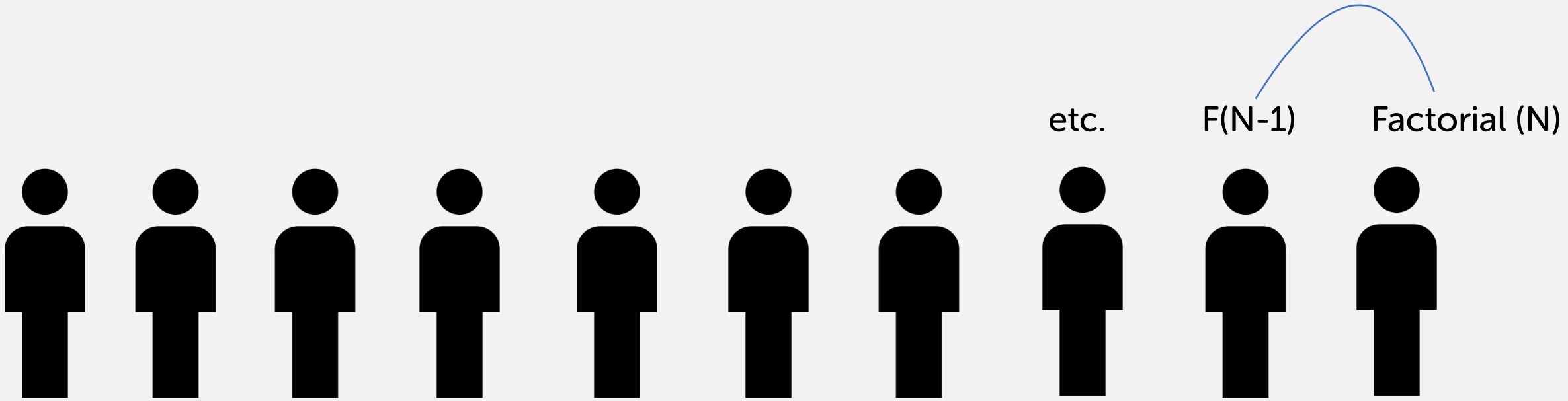
Recursive factorial function

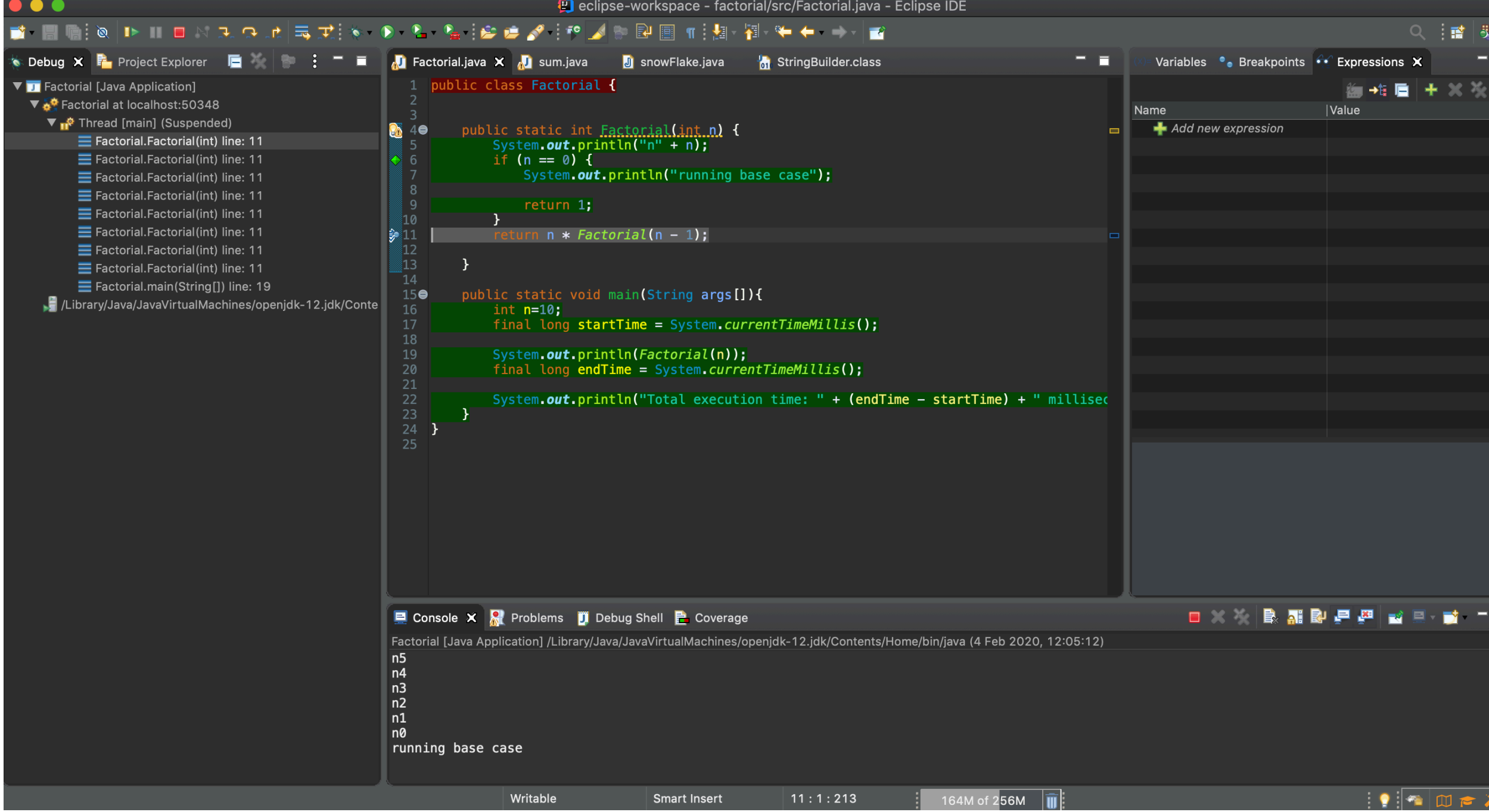
```
factorial(n):
    if n is 0
        return 1
    return n * factorial(n-1)
```

Hello World of Recursion: Factorial

```
public class Factorial {
    public static int Factorial(int n) {
        System.out.println("n" + n);
        if (n == 0) {
            System.out.println("running base case");

            return 1;
        }
        return n * Factorial(n - 1);

    }

    public static void main(String args[]){
        int n=10;
        final long startTime = System.currentTimeMillis();

        System.out.println(Factorial(n));
        final long endTime = System.currentTimeMillis();

        System.out.println("Total execution time: " + (endTime - startTime) + " milliseconds");
    }
}
```

Human Factorial





Debugging your code in Eclipse: https://www.vogella.com/tutorials/EclipseDebugging/article.html

Pitfalls of Recursion

1. Missing a base case: function will repeatedly call itself & never return
2. No guarantee of convergence: the sub-problem is not smaller than the original problem
3. Excessive memory requirements: excessive self-calls before returning, memory Java needs to keep track may be too much
4. Excessive Re-computation: some times a seemingly simple recursive program can require exponential time although this can be fixed....

Memoisation

A technique to speed up algorithms by keeping track of expensive operations and returning the cached result if the same computation is required again

```
1:  function FACTORIAL (n)
2:      Input: n non-negative integer
3:      Output: factorial of n: n!
4:      A ← new array of n integers
5:      s ← 0
6:      if n = 0 then
7:          return 1
8:      end if
9:      return n × Factorial(n − 1)
10: end function
```

https://en.wikipedia.org/wiki/Memoization

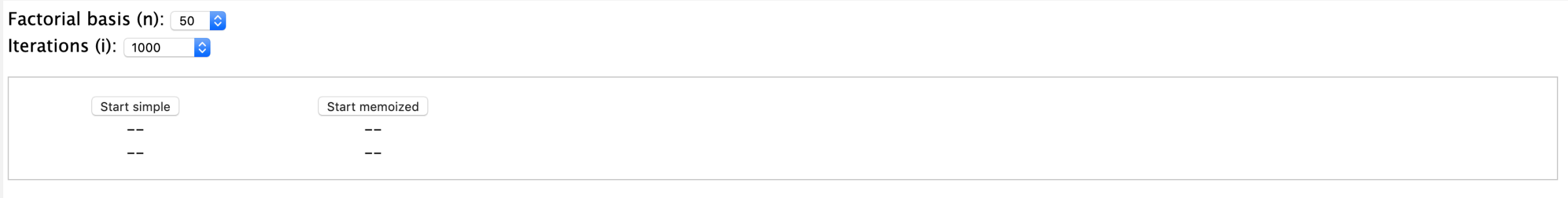
Memoisation

A technique to speed up algorithms by keeping track of expensive operations and returning the cached result if the same computation is required again

Use a lookup table

```
1:  function FACTORIAL(A, n)
2:      n a non-negativ integer
3:      Output: factorial of n
4:      if n = 0 then
5:          return 1
6:      else if n in lookup table then
7:          return lookuptable[n]
8:      else
9:          x ← n × Factorial(n − 1)
10:         store x in lookup table
11:         return x
12:     end if
13: end function
```

Fibonacci: careless duplication of work



https://andrew.hedges.name/experiments/memoization/

Fibonacci numbers

The Fibonacci Sequence is the series of numbers:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

The next number is found by adding up the two numbers before it.

The next number in the sequence above is 21+34 = 55

https://andrew.hedges.name/experiments/memoization/