COMPX301 Week 1

What you should know from this week:

- Assignment/test dates.
- What is an algorithm?
- What pseudocode is.
- What Collatz conjecture is.
- What recursion is.
- Different types of recursion.

What is an algorithm?

An algorithm is a step-by-step set of instructions to be followed in calculations or problem-solving operations.

An algorithm...

- has a sufficiently precise definition of what the legal inputs are.
- computes something (the algorithm itself).
- has a well-defined output.

An algorithm is said to be...

- partially correct if it sometimes stops and has the expected output when it does. It is not guaranteed to stop.
- totally correct if it always stops and produces the expected output for any legal input.

Partial correctness can be proven. Total correctness, except in trivial cases, cannot be proven.

Euclid's algorithm

Computes the greatest common divisor of two numbers. GCD(A, B). (Without modulo.)

An example of a totally correct algorithm.

```
while a != b
   c = a - b
   if (c < b)
      a = b
      b = c
   else if (c > b)
      a = c
   if (a == b)
      return a
```

Collatz conjecture/Hailstone numbers

Collatz states that all numbers are "hailstone" numbers, which satisfy the following algorithm. Will the following algorithm always stop?

It is definitely partially correct, but there is no way to prove that it is totally correct.

```
int id(int n)
   int m = n
   while (m > 1)
      if (m % 2 === 0)
         m = m / 2
      else
```

```
        m = 3 * m + 1
    return n
```

RISC - Reduced Instruction Set Computer

A reduced instruction set computer (RISC) is a microprocessor that uses a small set of simple instructions to perform tasks quickly. In Tony's example we look at a RISC architecture that uses only SUB and BRZ.

These won't be tested according to Tony.

```
SUB A, B     // Subtraction: A = A - B

BRZ A, adr   // Branch if zero: if A == 0, go to instruction at address "adr"

CLR A ?      // Clear: A = 0
becomes
SUB A, A

ADD X, Y     // Addition: How do we add?
becomes
CLR Z
SUB Z, Y
SUB Z, X
CLR X
SUB X, Z

MOV A, B     // Move the value from B into A
CLR A
ADD A, B

MUL A, B     // Multiply A and B and store the result in A
CLR C
```

```
loop:       // Loop, adding A into C/decrementing B each time.
   BRZ B, continue
   ADD C, A
   SUB B, 1
   BRZ 0, loop

continue:
   MOV A, C    // Store the final result in A
```

Recursion to iteration:

The following are some examples as to how to change from recursive functions to iterative functions.

Factorial example:

```
fact(n) =
{  undefined      n < 0
 1            n < 2
 n * fact(n-1)   otherwise }

// Recursive function to calculate the factorial of a certain number. In C-like langu
int fact(int n) {
   if (n < 0) error();

   if (n < 2) return 1;

   else return n * fact(n - 1);
}

// Can we remove the recursion to reduce the overhead?
int fact(int n) {
```

```
    if (n < 0) error();

    int result = 1;

    while (n > 1) {
        result *= n;
        n--;
    }

    return result;
}
```

Fibonacci example:

```
arr index: 1, 2, 3, 4, 5, 6, 7,  8,  ...
fib num:   1, 1, 2, 3, 5, 8, 13, 21, ...

// Method for the i'th fibonacci number:
fib(i) =
{  i < 3      1
   else       fib(i - 2) + fib(i - 1)   }

// Exponentially recursive method to return the i'th fibonacci number.
int fib(int i) {
    if (i < 3) return 1;
    return fib(i-2) + fib(i-1);
}

// Can we remove the recursion to reduce the overhead?
int fib(int i) {
    int f1 = 1;
    int f2 = 1;  // f2 will always hold the answer

    while (i > 2) {
```

```
        int t = f2;
        f2 = f1 + f2;
        f1 = t;
        i--;
    }

    return f2;
}

// The two methods below are still recursive, but no longer exponentially so.
int fib(i) {
    return fib3(0, 1, i);
}

int fib3(int f1, int f2, int i){
    if (i < 3) return f2;
    else return fib3(f2, f1+f2, i--);
}
```

Types of recursion:

Primitive recursive function:

This is a recursive function where the recursion can be removed.

Tail recursive:

A programming technique where a function returns the result of a recursive call without any further processing.

All tail recursive functions are primitive recursive.

Ackermann's function:

The first example of a recursive function where the recursion could not be removed:

```
f(m, n) = { n + 1            m == 0
            f(m - 1, 1)      n == 0
            f(m, f(m, n - 1)  otherwise }
```

“Not all computable functions are primitive recursive”.