Section 3 - Pre-computation

Time versus space.

This is the essential battle that programmers face. In order to go faster, more memory is used. In order to economize on memory, more computation is needed.

This duality is demonstrated in no better way than comparing a calculated method (economizing space at the expense of time) versus an entirely precomputed method (sacrificing space to reduce time).

In this section, we will demonstrate three methods of calculating factorials from 0 to 15.

- Iteratively
- Recursively
- By pre-computation

Certainly, for the purposes of this demonstration, it is not necessary to implement both iterative and recursive methods. We do so for fun and for any lessons the reader can glean.

C Driver

Here, you will find a version written in C. We will repurpose `main()` to drive versions in assembly language.

Iterative

```
long Iterative(long n) {
    long retval = 1;
    for (long i = 1; i <= n; i++) {
        retval *= i;
    }
    return retval;
}
```

First, notice that this algorithm's work increases linearly with the parameter n. Therefore this algorithm is O(n).

We translated this function into assembly language to produce the code provided below. This code is *condensed*. To see the original code with comments, please see here.

```
ifact:
        mov         x2, x0
        mov         x0, 1        // equivalent to retval = 1
        mov         x1, 1        // equivalent to i = 1
```

```
        // This has five instructions (20 bytes) in the inner loop which
        // increases in work by O(n).

10:     cmp         x1, x2
        bgt         99f
        mul         x0, x0, x1
        add         x1, x1, 1
        b           10b

99:
        ret
```

Reminder, the above code is *condense*. You will note that the code that performs the calculation is 5 instructions (or 20 bytes) long. This isn't much but again, the algorithm runs in O(n) time.

Recursive

```
long Recursive(long n) {
    long retval;
    if (n <= 1)
        retval = 1;
    else
        retval = n * Recursive(n - 1);

    return retval;
}
```

The code below is *condensed*. The original code, with comments, can be found here.

```
rfact:
        PUSH_P      x29, x30
        mov         x29, sp

        cmp         x0, 1
        bgt         10f
        mov         x0, 1       // ensure x0 is 1 - it could be less.
        b           99f

10:     // If we get here, n must be more then 1. Recursion is needed.

        PUSH_R      x0          // save the current n
        sub         x0, x0, 1   // prepare for recursion
        bl          rfact       //
        POP_R       x1          // restore the current n
        mul         x0, x0, x1  // multiply it by recursive return
```

```
99:     POP_P       x29, x30
        ret
```