# SE 115 Introduction to Programming I

| | |
|---|---|
| Lab No: | 05 |
| Topic: | Recursive Functions |

**Scenario 0:**

Write a function that makes factorial calculations. It will take a parameter *n*, and return the result of n factorial. Factorial means multiplying a number by every smaller positive number down to 1. By definition, `0!` equals `1`, so that is the natural stopping point for our recursion. The following is how *factorial(4)* it looks like in dry run.

*factorial(4)*

*= 4 * factorial(3)*

*= 4 * (3 * factorial(2))*

*= 4 * (3 * (2 * factorial(1)))*

*= 4 * (3 * (2 * (1 * factorial(0))))*

*= 4 * 3 * 2 * 1 * (1)*

*= 24*

In code, if `n` is `0`, we return `1`. Otherwise, we return *`n * factorial(n - 1)`*, which reduces the problem by one each time until we eventually reach `0`. For example, `factorial(4)` becomes `4 * factorial(3)`, then `4 * 3 * factorial(2)`, then `4 * 3 * 2 * factorial(1)`, and finally `4 * 3 * 2 * 1 * factorial(0)`. At that last step we return `1`, and the call stack multiplies back up to give `24`.

**Scenario 1:**

The goal in this scenario is to add up the decimal digits of a number, ignoring any minus sign. The easiest way to reason about this recursively is to take the last digit and add it to the sum of the remaining digits. First, define a function ***`sumDigits(int n)`*** that will take a number whose digits will be summed. If the number is a single digit (less than 10), we just return that digit. Otherwise, we compute (n % 10) + sumDigits(n / 10), which adds the rightmost digit to the sum of the rest of the number obtained by dropping that digit. For instance, sumDigits(305) is 5 + sumDigits(30), which becomes 5 + (0 + sumDigits(3)), and finally 5 + 0 + 3 = 8. Each step removes one digit, so the process always reaches the one-digit base case.

**Scenario 2:**

Write a function that will take the power of a given number. The function definition is **`power(long base, int exp)`**. Raising a number to a power means multiplying it by itself repeatedly. The simplest rule to stop on is that anything to the power `0` is `1`, so if `exp` is `0` we return `1`. If the exponent is larger than zero, we return `base * power(base, exp - 1)`, which makes the exponent one smaller at every step and guarantees that we will eventually hit `0`. As a short example, `power(3, 4)` becomes `3 * power(3, 3)`, then `3 * 3 * power(3, 2)`, then `3 * 3 * 3 * power(3, 1)`, and finally `3 * 3 * 3 * 3 * power(3, 0)`. The last call returns `1`, and multiplying everything together gives `81`.

*power(3,4)*

*= 3 * power(3,3)*

*= 3 * (3 * power(3,2))*

*= 3 * (3 * (3 * power(3,1)))*

*= 3 * (3 * (3 * (3 * power(3,0))))*

*= 3 * 3 * 3 * 3 * 1*

*= 81*

**Bonus:**

The Fibonacci number at position `n` is defined in terms of the two previous positions. The sequence starts with `fib(0) = 0` and `fib(1) = 1`, and every later value is the sum of the previous two, so `fib(n) = fib(n - 1) + fib(n - 2)` for `n >= 2`. The recursion stops at `0` or `1`, returning those values immediately. For example, `fib(5)` expands to `fib(4) + fib(3)`, then each of those expands again until all branches reach `fib(1)` or `fib(0)`, and the sums add up to `5`. This plain recursive version is easy to understand and fine for small `n`. If you ever need larger `n`, you can speed it up by remembering previously computed results in an array and reusing them instead of recomputing. Implement fibonacci number by creating a function with the following signature `fib(int n).`