Warm-Up Exercise

Here's a recursive function definition:

- f(x) = f(x-1)+f(x-1)
- f(0) = 1

Implement this in Scala.

1. What big-O complexity is your solution?
2. Can you do better? Why / why not / how?

Two naive implementations but one using an if-else expression and the other using pattern matching:

```
def f(x : Int): Int =
  if (x == 0) 1
  else f(x - 1) + f(x - 1)

def f: Int => Int = {
  case 0 => 1
  case x => f(x-1) + f(x-1)
}
```

1. The big-O complexity is $O(2^n)$, where n is the argument `f` is called by the user.

   One can imagine a tree with the root being `f(x)` where `x` is the initial value that the user calls `f` with. The root has two children `f(x - 1)`, each of which in turn have two children. Thus, at depth `i`, we have 2^i nodes. Summing up nodes at all levels, we get 1 + 2 + 4 ..

   $$1 + 2 + 4 + ... + 2^x = 2^{x+1} - 1 = O(2^x)$$

2. Alternative implementations:

   a. One can note that `f(x)` is simply the sum of 2^x ones that come from the `f(0)` leaf nodes of the tree we mentioned earlier. In other words, $f(x) = 2^x$. This can be implemented as easily as `def f(x) = 1 << x`, making it a single instruction function.

b. We could make it linear to the argument `x`, replacing two sub-calls to `f(x - 1)` with one as in `2 * f(x - 1)`

c. Implement `f` iteratively with mutation.

d. Use dynamic programming to memoize results of sub-calls, so that already computed values aren't recomputed.

Of course, options b, c, d are more of flight of thought, given option a is a single instruction implementation.

Other interesting notes:

- Since we're dealing with recursive functions, we can as well discuss tail-recursion. If we implement `f` (where we make one sub-call, ie `f(x) = 2 * f(x-1)`) with tail calls in mind, the compiler can optimize the sub-calls to occupy the same stack rather than taking `O(x)` stack size.

```
def f(x: Int): Int = {
  @scala.annotation.tailrec
  def loop(x: Int, acc: Int): Int =
    if (x == 0) acc
    else loop(x - 1, acc * 2)

  loop(x, 1)
}
```

- Interestingly, a compiler can have tail-call optimization even if the function is not tail-recursive. Implementing a certain function tail-recursively can be cumbersome and, more importantly, less efficient, e.g., `List.map`, which if tail-recursive usually needs to construct a new list and reverse it, so having such an optimization can be fortunate. OCaml recently got this optimization based on the paper "Tail Modulo Cons"