Math notes for August 19, 2020 session

Last revised: April 6, 2023

Some introduction will go here, explaining why the following might be worth reading, but I will have to remember what my point was when I first started to write this up. It is something about recursive definitions of (mathematical) functions.

1 Factorials

A textbook and prototypical example of a recursive algorithm is for factorial. Factorials, if you recall, are typically described[1] as in equation (1). The factorial of some number n is typically written "$n!$".

$$n! = 1 \times 2 \times 3 \times \cdots (n-1) \times n \tag{1}$$

That is something written as "5!" means $1 \times 2 \times 3 \times 4 \times 5$.

The same thing in slightly fancier notation is

$$n! = \prod_{i=1}^{n} i \tag{2}$$

but if (2) is at all confusing stick with (1).[2]

That traditional description doesn't really work well as a formal definition for a variety of reasons, so a better way to define that intent is to define it recursively, as in equation (5).

[1] When presented in high school and just over the natural numbers, which is what I'm talking about here. No one throw $\Gamma(x)$ around (yet)

[2] The only reason that I even bothered writing equation (2) is because the "$\cdots$ in equation (1) annoys me for reasons I won't get into.

Consider the factorials of 4 and of 5. If we look at how they are constructed we can see that 5! is just 5 times 4!:

$$4! = 1 \times 2 \times 3 \times 4 \tag{3}$$

$$\begin{aligned} 5! &= 1 \times 2 \times 3 \times 4 \times 5 \\ 5! &= (1 \times 2 \times 3 \times 4) \times 5 \quad \text{(added parentheses)} \\ 5! &= 4! \times 5 \end{aligned} \tag{4}$$

The idea that (5) expresses is that that the factorial of some number n is $n \times (n-1)!$.

$$n! = \begin{cases} 1 & \text{where } n = 0; \\ n(n-1)! & n > 0; \end{cases} \tag{5}$$

It may seem counter-intuitive that 0! is defined to be 1, but doing things this way makes many other definitions simpler and works with generalizations of the factorial function. So just live with it.

In Listing 1.1 we see a simple iterative (or loopy) implementation in Go factorial function, which corresponds to the definition of factorial given in equations (1) and (2). The work is done in the `for`-loop, and the result is accumulated in a variable we are calling "`result`". [3]

Listing 1.1 Iterative factorials for $n < 21$

```
func FactorialItrSmall(n int) int {
    result := 1

    for i := 2; i ≤ n; i++ {
        result = result ∗ i
    }
    return result
}
```

[3]Factorials get big really fast, and we have to do math on big numbers. Listings 1.1 and 1.2 will fail weirdly when n gets too big. And "too big" is greater than 20 on a 64 bit machine or above 12 on a 32-bit computer.

In Listing 1.2 we have an implementation that mirrors the recursive definition of the factorial in equation (5). The `switch` construction is just to find the case that n matches. In the default case, we just multiply n by the factorial of $n - 1$.

There are various reasons why one might want to implement the factorial one way or the other. To someone with a mathematical bent, the recursive way is nicer as it better expresses how we think about factorials. But typically iterative implementations are faster. Timing Listings 1.1 and 1.2 doesn't show the difference because we are limited to small values.

Listing 1.2 Recursive factorials for $n < 21$

```
func FactorialRecurseSmall(n int) int {
    switch n {
    case 0: // when n is 0
        return 1
    default: // any other value of n
        return n * FactorialRecurseSmall(n−1)
    }
}
```

To get a better sense of the timing differences with different implementations I've produced three more variants. Two of them, Listings 1.3 and 1.4, are just like the ones above except that they work on arbitrary sized integers, so we can test with larger numbers. Doing math on arbitrary sized integers and doing the memory allocation and management for storing those takes much more time than using the native data types.

In the loopy versions, 1.1 and 1.3, the number of times through the `for`-loop depends very tightly on n. Similarly, with the recursive versions, 1.2 and 1.4, the number of times the recursive function is called also depends very tightly on n.

And while these have been used to illustrate the differences between iterative and recursive algorithms, none of these are the way that you would actually implement these unless you have some very specific needs. And that is because the number of computations needed is directly proportional to n.

Wouldn't it be nice if there were a way to compute factorials, even if just approximately, where the number of operations doesn't grow so directly with n? A function that works for this whose amount of computation mostly just

Listing 1.3 Loopy big integer factorials

```
// FactorialIter computes n! iteratively.
func FactorialIter(n int) *big.Int {
    // we will accumulate the result in 'result'
    result := big.NewInt(1)

    for i := 2; i ≤ n; i++ {
        bi := big.NewInt(int64(i))
        result.Mul(result, bi) // result = result * i
    }
    return result
}
```

Listing 1.4 Recursive big integer factorials

```
// FactorialRecursive computes n! recursively
func FactorialRecursive(n int) *big.Int {
    switch n {
    case 0:
        return big.NewInt(1)
    default:
        bn := big.NewInt(int64(n))
        z := new(big.Int)
        return z.Mul(bn, FactorialRecursive(n−1))
    }
}
```

depends on the level of precision that we want (things like the square root work that way too) would be dandy. This is where the Gamma function, $\Gamma(z)$, comes in. The Gamma function, $\Gamma(z)$, is something that you really don't want to think about and is really quite complex.[4] What is relevant to us here is that factorials can be defined in terms of the Γ function as in (6).

$$n! = \Gamma(n+1) \tag{6}$$

The good news is that almost every computing language math library will have an efficient implementation of the (real) Gamma function, as it is very useful not only for computing factorial, but also in statistical analysis.

So Listing 1.5 gives a very efficient way to compute factorials for when we really need to compute factorials if you don't need the factorial of anything larger than 170. If you do need something larger than 170! you are doing it wrong. You probably only need the logarithms of the factorials or there is a far more efficient approximation for the problem you are trying to solve than using factorials directly.

Listing 1.5 Approximate factorials using Γ function (for $n < 171$)

```
func FactorialGamma(n int) float64 {
    return math.Gamma(float64(n) + 1.0)
}
```

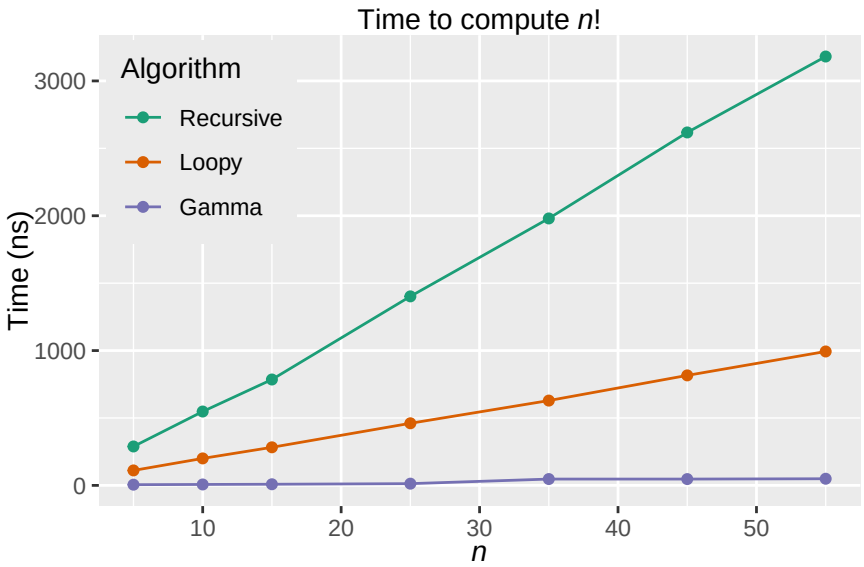
1.1 Timings

Largely because I wanted to teach myself how to use Go's benchmarking tools, I actually looked at the times of these, which you can see in Figure 1.

Similarly, we can see in Figure 2 that the memory demands on the recursive algorithm rise with the size of n while the memory requirements using the $\Gamma(x)$ remain constant. Contrasting figures 1 and 2 we do see that the memory demands of the loopy (iterative) algorithm does not grow as fast as the time. I'll put explaining why recursive algorithms can consume more memory onto the stack of things that I'd like to talk about someday.

[4] All puns intended.

Figure 1: Timing of factorial computation from listings 1.3, 1.4, and 1.5



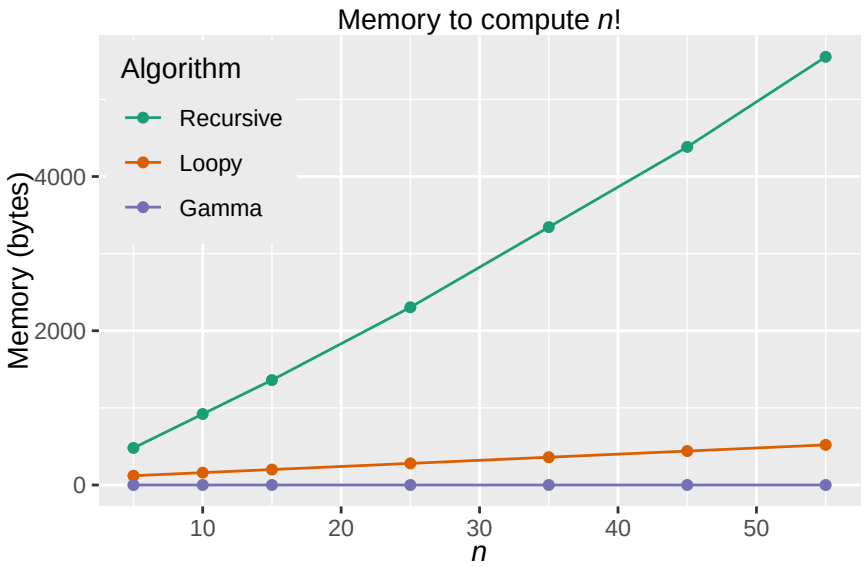
2 Handshakes

Recall the problem of how many handshakes we have in a group of n people with each person shaking everyone else's hand exactly once. Or let's shift this to something more socially distant and actually relevant. If we have n people and we need a (symmetric) encryption key for each pair, then how many of these keys do we need.

So if we have five people: Alice, Bob, Charlie, Dana[5], and Eve. One (very inefficient) way of doing computing the number of keys needed is to work through each pair and count them. So Alice, will have a separate key with each of the other four. That gives us four so far. Bob already has a key with Alice, so he just needs one for the other three. So that gives us another three, making a total (so far) of seven. We've already counted Charlie's keys with Alice and Bob, but he also needs a key with each of Zul and Eve. So that brings us to a total of nine. Finally, we need a key for Zul and Eve, bringing our grand total to 10.

[5]There is no Dana. Only Zul.

Figure 2: Memory requirements of factorial computation from listings 1.3, 1.4, and 1.5



That approach of actually running through the set of $\{1, 2, \ldots, n\}$ items and counting up the possible pairs is illustrated in Listing 2.1.

Listing 2.1 Counting the handshakes, one by one

```
func HandshakeCounting(n int) int {
    count := 0 // we will accumulate our count here

    for i := 1; i < n; i++ {
        for j := i + 1; j ≤ n; j++ {
            count = count + 1
        }
    }
    return count
}
```

Now moving on to Daniel's recursive algorithm for the handshaking algorithm, consider the n-th person added to the group as adding $n - 1$ new handshakes. And so we can define the function as in equation (7), which can be implemented as shown in Listing 2.2.

$$f(n) = \begin{cases} 0 & \text{where } n = 0; \\ (n-1) + f(n-1) & \text{where } > 0. \end{cases} \tag{7}$$

Listing 2.2 Counting the handshakes recursively

```
func HandshakeRecursive(n int) int {
    switch n {
    case 0:
        return 0
    default:
        return (n − 1) + HandshakeRecursive(n−1)
    }
}
```

Fortunately, there is a much simpler way of performing the computation. Looking at and thinking about equation 7, we can see (after a while of thinking about it) that for each of the n people we are adding $n - 1$ keys. But

because each pair of people share a key, we really just need half of those. So the whole thing can be summed up as

$$f(n) = \frac{n(n-1)}{2} \tag{8}$$

$$= \frac{n^2 - n}{2} \tag{9}$$

That naturally leads to the simplest algorithm for this problem as shown in Listing 2.3.

Listing 2.3 Calculating handshakes from a formula

```
func HandshakeClosed(n int) int {
    return (n * (n - 1)) / 2
}
```

3 Exponentiation

We were all taught at some point that a^b means multiplying a to itself b times. Something like

$$b^n = \underbrace{b \times b \times \cdots \times b}_{n \text{ occurrences of } b} \tag{10}$$

Consider, for example, that 3^7 is three times bigger than 3^6. So we can recognize that $3^7 = 3 \times 3^{7-1}$. More generally

$$b^{n+1} = b \times b^n \tag{11}$$

We also know that we want something like 3^1 to equal three, and so there is one other things that we need to make the definition in (11) to make it work.

$$b^{n+1} = \begin{cases} b & \text{when } n = 0, \\ b \times b^n & \text{otherwise.} \end{cases} \tag{12}$$

From either definition of b^n would get the important fact that

$$b^n \times b^m = b^{n+m} \tag{13}$$

but I will skip how we get there. Either accept that or play with it for a bit to see that this at least seems to make sense.

But there are some things we can conclude from (12) that we cannot from (10). Suppose we want to know whether we can define exponentiation for when n is not a positive whole number. The definition in (10) just can't even be expressed when n isn't some positive whole number. For the definition in (12) we may not know how it would work, but at the outset we don't know that it can't work.

If we want to keep $b^{n+1} = b \times b^n$ true for all n, then we can ask what happens when $n = 0$. Let's look at this when $n = 0$ and $b = 3$.

$$3^{0+1} = 3^1 = 3 \times 3^0 = 3 \tag{14}$$

and so if $3 \times 3^0 = 3$ then 3^0 must be 1. This remains true for any base, b (expect for when b zero): $b^0 = 1$. In fact, many people prefer a slight variant of (12)

$$b^n = \begin{cases} 1 & \text{when } n = 0, \\ b \times b^{n-1} & \text{otherwise.} \end{cases} \tag{15}$$

Now let's ask about $3^{\frac{1}{2}}$. What could that mean? Well, making use of the fact in (13) that $b^n \times b^m = b^{n+m}$, we know that

$$3^{\frac{1}{2}} \times 3^{\frac{1}{2}} = 3^{\frac{1}{2}+\frac{1}{2}} = 3^1 = 3 \tag{16}$$

So we have a thing, $3^{\frac{1}{2}}$, that when multiplied to itself is 3. This makes $3^{\frac{1}{2}} = \sqrt{3}$.

My first point here is that the recursive *definition* of exponentiation is much more general than the loopy one. It can be extended to n being negative or complex as well in ways that make sense and preserve all of the properties of exponentiation that are found useful. This is one of the reasons why math teaching might use the iterative definition (10) at first to give people an idea of it, but will actually use a more abstract definition as in (15).

Even though (15) works as a definition when n is negative or fractional, it does no better than (10) at giving us a way to compute b^n when n is not a positive whole number. But rest assured that just as $\Gamma(x)$ came to

the rescue of computing the factorial, the actual algorithms computers use for exponentiation (for floating point numbers) is based on an even more abstract definition and can be computed more efficiently.

4 Algorithms

There is a beauty of expressiveness in defining things recursively. But often times recursive algorithms are slower and consume more memory than their iterative counter-parts. As Daniel said that when trying to figure out the complexity of an algorithm it is easier to break it down into different forms as well.