Fast-growing Functions (Really Big Numbers)

Jacob Stavrianos

October 11, 2020

1 The Large Number Game

1.1 Game #1: Complete Information

The Large Number game is played as follows. To start, player 1 picks a positive integer. Player 2 must then pick a greater positive integer. The game continues until one player can no longer go, then the last person to go wins. An example of this game is shown below:

$$100, 100+1, 100^{100}, 100^{100}+1, n, n+1, ...$$

This game isn't very interesting. More specifically, if player 2 knows player 1's number, then they can always play that number plus 1 and the game continues forever.

1.2 Game #2: Incomplete Information

To eliminate this basic strategy, we won't let the players can't know each others' numbers. In this new game, all players will have 1 minute to write down the largest number they can on a piece of paper using only decimal digits. Whoever has the biggest number at the end wins!

Some strategies for Game #2:

1. The largest decimal digit is 9. So if I can write 1 digit per unit of time, then I should write all 9's.
2. ...But I can't write 1 digit per unit of time, I can write 1's much faster than other digits. Since the number of digits is the only really important thing in play, I should write all 1's.
3. There must be a better way to mass-manufacture 1's. For example, I could draw giant 1's and split them into smaller 1's with an eraser. By doing this in bunches of 1's, I can create n^2 1's with $2 * n$ lines, much better than just writing 1's.

Notice how each successive strategy not only beats the strategy before it, but completely crushes it. Someone using strategy 3 could beat someone using strategy 1 even with a huge time disadvantage. More generally, a better strategy

always beats a worse strategy, regardless of the amount of effort put into the worse strategy. This is the fundamental idea behind the actual Big Number Game: a smarter strategy will always win.

1.3 Game #3: The Actual Big Number Game

Now that we know how to write decimal numbers, let's add some freedom to the game. Each player has 1 sheet of paper to define a number, using words, numbers, or symbols. The only restriction is that the entry must uniquely define a positive integer. Now that we can use words and math, the numbers will get much, much bigger.

2 Recursion

2.1 Exponentiation

Let's start with a really big number, like 100. We want to generate a bigger number from our already big number by applying some formula to it. Exponents are pretty big, so we try those: we get 10^{100}! That's a really big number! If you don't believe me, try to count to it.

However, we can get even bigger numbers. Like $10^{10^{100}}$. Or even $10^{10^{10^{100}}}$, and so on. Taking 10 to the power of stuff is proving to be a very powerful generator of big numbers.

Seeing as we want to take as many powers of 10 as possible, what if we took $10^{10^{\cdots^{10}}}$ with 100 10's in total? This idea of recursive exponentiation is much, much bigger than anything so far. Formalizing this idea, we can define a function $f(a, b) = a^{a^{\cdots^{a}}}$, with b a's altogether.

2.2 Better exponentiation

This recursive system isn't new. If we try to define exponentiation in the same way, we get $a^b = a * (a * (...a))$, with b a's. Similarly, $a * b = a + (a + ...a))$, still with b a's. In words, each bigger operation is just repeatedly iterating the one below it.

With this insight, we can now define a series of operations, with each one being recursive application of the previous one. Specifically, this gives us a series of operations $H_n(a, b)$ such that

- $H_1(a, b) = a + b$
- $H_{n+1}(a, b) = H_n(a, (H_n(a, \ldots H_n(a, a))))$, with b a's in total.

This set of operations is called the Hyperoperation Series. By this definition, H_1 is addition, H_2 will be multiplication, H_3 is exponentiation, and so on. Now, consider H_{100}. Since each successive function iterates over the previous one and as such is vastly larger, $H_{100}(10, 10)$ will be much bigger than any number so far.

(Note: the wikipedia definition uses $H_0(a, b) = b + 1$, so their recursive formula is a bit messier than the one used here.)

Generating numbers with hyperoperations will get us to about Graham's number or something of similar size. Graham's Number is pretty firmly in the realm of unimaginably large numbers, but we can do much better.

3 Combinatorial results

3.1 Trees

The first *really big* function here has to do with trees. A tree is an undirected graph where each pair of vertices is connected by exactly 1 path. We won't really use this definition of trees, we just need a theorem about them.

Specifically, we'll use Kruskal's Tree Theorem, which states that for certain types of labeled trees, any infinite set of such labeled trees will have a pair of trees such that one tree can be embedded inside the other. Interestingly, this theorem cannot be proved in some weaker forms of logic. More on that later.

Given this theorem, we can define our first really huge function, called tree(n). We let tree(n) be the maximum size of a *sequence* of trees such that:

- The ith tree has at most $i + 1$ vertices
- No tree in the sequence is embeddable inside another tree in the sequence

By Kruskal's Tree Theorem, this function exists for all n. By being a little more clever with the sequence construction, we can get an even bigger (very famous) function called TREE(n).

3.2 A sense of scale

Now that we've gotten to the really big functions, we should take a moment to reflect on just how big the numbers we're getting are.

First of all, the number of atoms in the universe is less than 10^{1000}. *Ha.* Pathetic.

One step higher is stuff like Graham's number, as mentioned before. Graham's number is $f(f(f(\ldots f(4))))$ with 64 f's, where $f(n) = H_{n+2}(3, 3)$. Writing down Graham's number using only digits and exponents would take longer than the lifetime of the universe.

Then we get TREE. Once again, expressing TREE(10) with only hyperoperations would take a laughably long time, more time than can possibly be expressed without use of TREE itself.

4 Computability

4.1 Super-recursion

Exercise: write a computer program that computes $H_n(a, b)$.

First of all, no computer in the universe would be powerful enough to run it. But aside from that, the code is fairly simple. The H_n definition is nicely recursive, with an easy base case. This could be done in python in, say, 100 characters.

The problem is, there might be another 100 character python program that spits out even bigger numbers. So let's take the biggest one. Let S_n be the set of python programs with at most n characters that output a single natural number. We define $b(n)$ to be the maximum output of any $S \in S_n$.

A few issues here:

- One of the S's might throw an error: sure, ignore those ones.
- One of the S's might run forever: sure, ignore those ones too.

So now we have this function $b(n)$ which is supposed to be big. How big? Well, you could write a program that computes TREE(10^{1000}) in maybe 1000 characters. As such, $b(1000) \geq$ TREE(10^{1000}). As it turns out, $b(1000)$ is way bigger than anything defined so far.

4.2 Busy Beaver

This "try every possible computer program" idea is famous enough to have its own well-known function. It's defined using *Turing Machines*, which are a type of idealized computer system. As it turns out, all (good) programming languages can be represented as Turing Machines and vice versa, so using code is nice for intuition.

That said, we define the Busy Beaver function BB(n) as the largest possible number of 1's that a Turing Machine with n states can output, provided that the machine will eventually halt. If you didn't understand that sentence, ask Comp Team.

4.3 Uncomputability of $b(n)$

With all this talk about computer programs (programs for short), you can't help but wonder if there's a program that computes $b(n)$.

Assume so; call the program P that takes in n and spits out $b(n)$. Now, add a line to the end of P that adds 1 to the output. We get a program P' taking in n and outputting $b(n) + 1$.

Let l be the length of program P' with input n. Technically, l changes based on n, but this change is small (proportional to $\log n$). Now, since $b(n)$ takes the largest value over all programs, we have $b(n) \geq P'$ *output* for n much larger

than l. But P' outputs $b(n) + 1$. Wait.

In short, if there was a program P that computes $b(n)$, then b can use the program P to break itself. Thus, no computer program (or Turing Machine, or anything else) can compute $b(n)$ for all n.

4.4 Computable functions

Call a function $f : \mathbb{N} \to \mathbb{N}$ *computable* if there exists a program taking input n and outputting $f(n)$. The official definition uses Turing Machines, they're the exact same thing.

By the definition of $b(n)$, we know that for any computable function $f(n)$, we have $b(n) > f(n)$ for large enough n. b can just use the program for f and add one to the end result. This offers another proof that b is *uncomputable*, meaning no program exists taking in n and outputting $b(n)$ for all n.

Also, the statement $b(n) > f(n)$ is incredible on its own. That means $b(n)$ will be bigger than anything we've thought of that can be made into a program. In fact, $b(10^{100})$ another mental order of magnitude bigger than all sane computable functions.

4.5 The Halting Problem

This is one of those times in math where we get something really cool seemingly out of nowhere.

The *Halting Problem* asks whether we can build a program that determines whether its input (another program) will eventually halt. Formally, does there exist a program H taking as input an inputless program P, outputting 1 if P will run forever and 0 otherwise?

This problem was a big deal in the 20's, and part of why Hilbert gt so salty, until Alan Turing solved it in 1936 using a diagonalization argument. We will solve it here using black magic and the fact that $b(n)$ is uncomputable.

4.6 The Halting Solution

Suppose that such a program H exists. We will write a program B that takes in n and computes $b(n)$. B works as follows

Make a list of all elements of S_n, the finite set of programs with n characters or less. For each S in the list:

- If S throws an error, ignore it. A computer can check this easily.
- Run H with input S. If it returns 0, then S never halts, then ignore it.
- If S passed the above two tests, run S. If it outputs a single number, store that number in a list M

M must have finite length, so we can compute the maximum element of M, which is returned.

We made a program that computes $b(n)$! But that's provably impossible, so by contradiction, no program H exists. *Magic.*

5 Definability

That was fun, let's make even bigger numbers.

5.1 Why Math is better than CS

We should probably keep the "take the maximum value of a large set" idea. Pretty much anything else we can come up with will fall into the set (example: all possible computer programs) and so won't be big enough. But last time we only used comupter programs. Is there anything stronger that we could use?

Of course there is. We can use set theory, specifically first-order set theory.

To do this, we need to make "set theory statements with less than n characters" output numbers, that we can take the max of. This isn't too bad, but requires some formalism.

5.1.1 Set theory background

First-order logic, which is the background for set theory, deals with sentences constructed from a bunch of symbols:

- Variables ($x, y, z, \dots$ or $x_1, x_2, x_3, \dots$)
- Logical connectives (and, or, not implies, ...)
- Quantifiers (For all, there exists)

In first-order logic, each sentence is in principle either true or false. However, some sentences have unassigned variables, called *free variables*. These sentences will be either true or false once the free variables are assigned, but until then are "neutral".

We want a way to make sentences "name" a specific natural number, so we can take a maximum over all sentences. To do so, we consider sentences ϕ with exactly one free variable. We say ϕ *names* a natural number k if $\phi(n)$ (ϕ with the free variable set to value n) is true if and only if $n = k$.

Small technicality: Set theory is a specific type of logic in which all the variables are sets, along with some other things like the $\in$ operation. Because of this, set theory doesn't literally contain numbers. We get around this by defining numbers within set theory as specific sets and defining number operations like addition based on those sets. There are several examples of how to do this, like the Von Neumann Ordinals or $0 = \{\}, 1 = \{\{\}\}, 2 = \{\{\{\}\}\}, \dots$, but the details aren't important for our purposes.

For those interested, the above is a hand-wavy example of mathematical logic and model theory, so go google that.

5.2 The Rayo Function

With our set theory construction in place, we're ready to define our next really big function, which we'll call $r(n)$.

The definition should look familiar: let $r(n)$ be the maximum number named by a first-order set theory sentence ϕ with n symbols or less.

At this point, intuition for what the function is doing starts to get tricky, but we can give heuristics. We can have sentences like $\phi(x) : x = 5$, which names 5. Furthermore, we can construct arithmetic (and more generally, programs) in set theory, so we could replace 5 with pretty much anything $b(n)$ could come up with.

History note: $r(n)$ is based off of the function Rayo(n), originally defined by MIT professor Agustin Rayo in a "large number duel" against Princeton's Adam Elga. No really, this happened. See http://tech.mit.edu/V126/N64/64largenumber.html for proof.

5.3 Provability

Analogy time. We took a maximum over computable functions (programs) and got a function that itself wasn't computable. We just took a maximum over ZFC (the standard set theory; others can be used as well). So is the resulting function un-ZFC-able?

The answer is yes, by similar logic to the $b(n)$ proof. Specifically, we can prove that no 2-free-variable sentence $\theta(m, n)$ exists such that, for each m, the 1-free-variable sentence $[\theta(m)](n)$ names $r(m)$.

Philosophically, this means ZFC has no idea what $r(n)$ is. Maybe you can use this to prove Godel's incompleteness theorem. Go ask a logician.

You might now be convinced that this pattern is general. We generate large numbers by taking a maximum over all "statements" in some "theory", coming up with a function that the theory doesn't know how to describe. Thus, the more powerful our theory, the bigger our function, and the less our mathematics understands it. This leads to funny consequences like big enough functions having valued dependent on the Continuum Hypothesis.

6 The Great Beyond

We've arrived at the state-of-the-art for large numbers. Sure, you can always add 1, but I've never heard of functions meaningfully bigger than $r(n)$ in the same way that TREE and $b(n)$ were the biggest so far.

However, there are some cool tricks that can be done, which are at least interesting to think about.

6.1 Oracles

When we were defining $b(n)$, we motivated it by saying it could write a program for Hyperoperations. *Hyperoperations.* Surely we can do better than that. So what happens when we give $b(n)$ something more powerful to play with?

Let's define a new function $b_2(n)$. It's the same maximum over all programs, with one difference: programs for $b_2(n)$ can call the function $b(n)$. Computability-wise this is impossible, but who cares? There are still finitely many programs and $b(n)$ is well-defined, so $b_2(n)$ also is.

This idea of letting a program magically call a function is called an *oracle*. Some of you (comp team?) might have heard of these before; it's the same concept here, but used for the purpose of making huge numbers.

We can write this as $b_2(n) = b^b(n)$, as in the function $b(n)$ but with an oracle to (the ability to call) the function b.

We can keep going with this: let $b_3(n) = b^{b_2}(n)$. Exercise: build hyperoperations into this. But don't get too carried away, working smarter always beats working harder.

6.2 More set theory

In defining $r(n)$, we pretty arbitrarily decided to use "first-order set theory" for our sentences, specifically ZFC. We could use any set theory we wanted. Specifically, we could build a stronger set theory, which hopefully will be able to name larger numbers.

This actually does give some sizeable improvements, in the same way that $r(n)$ crushes $b(n)$, which is like a really weak set theory only describing computable things.

This creates a problem of constructing stronger set theories, which is difficult and pretty advanced but definitely possible. One niche-famous example is called First Order Oodle Theory, the result of appending an "is provable" operator to Set Theory to make fun of Godel's incompleteness theorem.

6.3 Diagonalization

Diagonalization, like a James Bond villain, is the shadowy figure behind all of our large functions. It's also a hard term to define.

The term originated with Cantor's Diagonal Argument, which proves that no bijection (one-to-one matching) exists between a set and its powerset. The proof is by construction, defining an element of the powerset in such a way that it

can't be matched with anything.

In general, mathematicians use the term diagonal to describe proofs that go like this: assume something is true, having to do with all statements in a certain language. We construct something, usually either self-referentially or by combining every statement of the language, that creates a contradiction.

Diagonalization is a super common proof technique in many fields of logic, leading to many famous results that philosophers like. The standard proof of the halting problem (Turing's proof) uses diagonalization, as well as the key lemma in Godel's incompleteness theorem (fittingly called the Diagonal Lemma).