REALLY BIG NUMBERS

GLENN SUN
UCLA MATH CIRCLE ADVANCED 1

APRIL 9, 2023

1 Knuth's up arrow notation

Donald Knuth (1938–) is an American computer scientist and mathematician. In 1974, he received the Turing Award, the highest award in computer science equivalent to a Nobel Prize, for his contributions in pioneering the analysis of algorithms and complexity theory. He is also the inventor of the TEX typesetting system, which is being used to write this very worksheet! (Photo credit: *New York Times*)

In school, you learned about counting, then addition (repeated counting), then multiplication (repeated addition), then exponentiation (repeated multiplication). Knuth's up arrow notation generalizes this. We define $a \uparrow b = a \uparrow^1 b = a^b$, then $a \uparrow\uparrow b = a \uparrow^2 b = a \uparrow (a \uparrow (\cdots \uparrow a))$ (b times), and similarly $\uparrow^n$ as repeated $\uparrow^{n-1}$.

Problem 1. Compute the following. (You may use a calculator.)

1. $2 \uparrow\uparrow 2 =$

2. $3 \uparrow\uparrow 2 =$

3. $2 \uparrow\uparrow 3 =$

4. $2 \uparrow^3 3 =$

We can use Knuth's up arrows to quickly describe extremely large (but finite) numbers.

The OEIS (online encyclopedia of integer sequences) at oeis.org is the world's largest collection of integer sequences. Almost any sequence you can think of can be found there.

Problem 2. Even $\uparrow^2$ already grows incredibly fast. Calculate $1 \uparrow^2 1$, $2 \uparrow^2 2$, and $3 \uparrow^2 3$. Type this sequence into the OEIS search box. What is a fun fact about $4 \uparrow^2 4$?

However, certain cases aren't very large.

Problem 3. Prove that $2 \uparrow^n 2 = 4$ for all n.

1.1 The Ackermann function

Wilhelm Ackermann (1896–1962) was a German mathematician who worked in mathematical logic. Together with David Hilbert, he wrote the first ever book about first order logic. (Photo credit: *Wikipedia*, public domain)

The Ackermann function is defined

$$\begin{aligned} A(0,n) &= n+1 \\ A(m+1,0) &= A(m,1) \\ A(m+1,n+1) &= A(m, A(m+1,n)) \end{aligned}$$

Problem 4. Fill in the unshaded squares in the below table for $A(m, n)$.

$m \backslash n$	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3					■	■	■	■	■	■
4		░	■	■	■	■	■	■	■	■

(Challenge) Do $A(4, 1)$, the lightly shaded square.

Problem 5. Type the first few terms of the main diagonal $A(n, n)$ into the OEIS and read the fun facts. What is $A(4, 4)$?

Problem 6. Explain why

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0, \\ (2 \uparrow^{m-2} (n + 3)) - 3 & \text{otherwise.} \end{cases}$$

You might have read on the OEIS that the Ackermann function is *computable* but not *primitive recursive.* Last week, we talked about computability and how Alan Turing defined the Turing machine, which is widely recognized today as the "correct" definition of computability. Even if you were not here last week, $A(n, n)$ is clearly computable by humans: the previous question gives the formula. But what does primitive recursive mean?

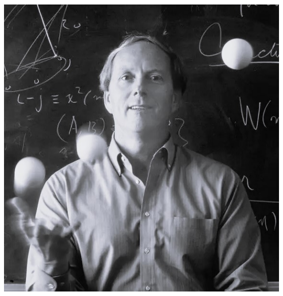
We don't often talk about what mathematicians get wrong, but in fact, mathematicians make mistakes all the time. So here is today's relevant example: originally when they tried to formalize what it means for a function to be computable, they actually got it wrong! At first, mathematicians thought that a function was computable if and only if it could be derived from

- constant functions c_n, where $c_n(x) = n$ for all x,
- the successor function s, where $s(x) = x + 1$ for all x, and
- the projection functions p_n^m, where $p_n^m(x_1, \ldots, x_m) = x_n$,

using a finite number of function compositions and for loops. (If you're unfamiliar, a for loop is an instruction that says something like "for $x = 1, 2, \ldots, n$, do some computation based on x." In particular, the number of times we do the computation, here n, is known before the start of the loop.) Today, we call such functions *primitive recursive.*

Problem 7. (Challenge) Show that the Ackermann function is not primitive recursive. You should ask an instructor for details if you want to do this problem.

1.2 Graham's number



Ronald Graham (1935–2020) was an American mathematician who worked in discrete mathematics. His work spanned many areas, including number theory, Ramsey theory, graph theory, and discrete geometry. He was also a prolific juggler, serving as the president of the International Jugglers' Association. (Photo credit: *New York Times*)

Graham was trying to solve the following problem: An n-dimensional hypercube has 2^n vertices. Draw a line between every pair of vertices, and assign each edge to be red or blue. What is the smallest n for which every possible coloring contains the following structure: four vertices on the same plane with all the lines between them the same color.

Problem 8. Prove that in this problem, $n > 2$. (In other words, draw a a square with lines between every pair of vertices that does not contain the above structure.)

(Challenge) Prove that in this problem, $n > 3$.

Graham proved that $n \leq g(64)$, where we define $g(1) = 3 \uparrow^4 3$, and $g(n) = 3 \uparrow^{g(n-1)} 3$. Some people refer to $g(64)$ as *Graham's number*. Because $g(n)$ is an interesting integer sequence, I originally wanted to ask you to also look it up in the OEIS, but unfortunately I realized the following problem:

Problem 9. If we divide the observable universe into cubes of side length 1 Planck length, there are about 10^{185} such cubes. Show that $g(1) = 3 \uparrow^4 3$ has more than 10^{185} digits. (In other words, show that $3 \uparrow^4 3 > 10^{10^{185}}$.)

Problem 10. For large values of n, which is bigger: $g(n)$ or $A(n, n)$? Give a quick argument for why.

2 The busy beaver function

Now we describe even bigger numbers. Recall last week's definition of a Turing machine. This is a model of computation mimicking how humans actually compute. It consists of:

- A *tape* consisting of *cells*, which we think of as an infinitely long row of pieces of paper.
- A *finite set of symbols* (last week we used 1 and blank) that can be written on the cells, akin to writing English or math on paper.
- A *finite set of states* $q_0, q_1, \ldots$ that tell you what "stage" of the algorithm you are in, akin to knowing in your mind what you're trying to do.
- A *set of instructions* that based on your current state and what you read on the paper, tells you to update your state, then either overwrite the symbol on the tape or move

to an adjacent sheet of paper. For example, one instruction could be "if you are in q_0 and you read 1, then move right and stay in q_0." Another could be "if you are in q_1 and you read blank, then write 1 and change to state q_3." Last week, we abbreviated these as $q_0 1 R q_0$ and $q_1 B 1 q_3$.

Furthermore, in the definition we gave last week, a Turing machine halts (stops its computation) if there is no instruction for what to do in the current state reading the current cell. The output of the computation is the number of 1's on the tape when/if it halts.

We define the busy beaver number BB(n) is the largest number of steps a Turing machine can take before halting, given an all-blank input, and using n states. If a machine does not halt on the all-blank input, we don't consider it.

Problem 11. Suppose we have a Turing machine with only one state q.

1. List the only 8 possible instructions.

2. A Turing machine must contain at most one instruction starting with $q1$ and at most one instruction starting with qB. How many Turing machines are there with one state?

3. Of these Turing machines, which ones halt on the empty input?

4. What is BB(1)?

Problem 12. This problem will show how to compute large numbers with few states. Note that you need at least n steps to output n, so this also shows how to make programs run for many steps with few states.

1. Last week, you wrote a set of instructions that would output n on an all-blank input. How many states did you use? (If you were not here last week or forgot your answer, first write a set of instructions that does this!)

2. Describe a set of instructions that on input x, outputs $2x$. (The input is given as a string of x 1's surrounded by infinitely many blanks in both directions. It might be too tedious to write down the instructions explicitly, so you can just informally describe what you want to do.)

3. Using the previous part, describe a way to output 2^n from an all-blank input using just (approximately) cn states, where c is some constant.

4. (Challenge) Describe a way to output $2 \uparrow^n 2$ using just approximately cn states, where c is some constant.

Despite BB(1) being quite small, BB(n) actually grows way faster than both $A(n, n)$ and $g(n)$ after a while, as well any function given by a formula that you can think of! Now we will work towards showing this surprising fact.

Problem 13. This problem will develop the Halting problem, which will be a key tool in our argument in the next problem. You may skip this problem if you did these things last week.

1. Let $p_1, p_2, p_3, \ldots$ be an enumeration of all Turing machines. (For example, you can list all the machines with 1 state, then 2 states, then 3 states, etc.) Show that the function

$$f(x) = \begin{cases} p_x(x) + 1 & \text{if the computation of } p_x(x) \text{ halts} \\ 0 & \text{if the computation of } p_x(x) \text{ doesn't halt} \end{cases}$$

is not computable by any Turing machine.

2. Using the previous part, show that the function

$$H(x, y) = \begin{cases} 1 & \text{if the computation of } p_x(y) \text{ halts} \\ 0 & \text{if the computation of } p_x(y) \text{ doesn't halt} \end{cases}$$

is not computable by any Turing machine. (This is called the halting problem. We can interpret x as the code you are trying to run and y as the input to that code, so there is no algorithm to test if your code will get stuck in an infinite loop.)

Problem 14. This problem will show that BB(n) grows faster than the Ackermann function and Graham's function.

1. Explain why the Ackermann function $A(n, n)$ and Graham's numbers $g(n)$ are both Turing computable. (Again, no need to explicitly write instructions, just a rough idea will do.)

2. Show that if BB(n) could be computed by a Turing machine, then we would have an algorithm to decide the halting problem.

3. Conclude that BB(n) grows faster than any computable function.

3 More big numbers

Problem 15. Read about TREE(3). How does TREE(n) compare to BB(n)?

Problem 16. Read about Rayo's number. How does Rayo(n) compare to BB(n)?