Georg Cantor: My Favourite Computer Scientist

Manasseh Ahmed

§1 Acknowledgements

Made with `evan.sty`. Special thanks to Maria Bayder and Benjamin Gurevitch for helping with general content, editing, reviewing, and LaTeX . All the diagonalization diagrams are by Benjamin Gurevitch.

§2 Prologue: A Primer for What's to Come

In mathematics the art of proposing a question must be held of higher value than solving it.

— Georg Cantor

What constitutes a good math question? Well, it should probably have a neat solution; I guess its solution should also be useful, either in other areas of mathematics or in science. While this might not be a controversial take, I think we need to understand that predicting whether a question will lead to something useful is very difficult. The journey we are about to embark upon will demonstrate this idea; we'll go all the way from set theory to computer science, and I hope you will see how, sometimes, just asking a question for its own sake can lead to remarkable ideas.

§3 Act I: A Stroll through Set Theory

Let's start with sets. Simply put, a set is a collection of some type of thing. For example, the set of my favourite numbers could be $\{7, 8, 9\}$. Or, the set of my favourite letters could be $\{a, b, c\}$. But these are boring examples, let's talk about infinite sets.

A natural question that arises in discussing infinity is: "how do we define infinity?" Well, here's an idea:

Definition 3.1 (First definition of infinity)**.** A set is "infinite" if you can procedurally list its elements in such a way that for each element in the set we can specify its place in the list.

Let's look at some examples. The set $\mathbb{N} = \{0, 1, 2, 3, ...\}$ satisfies our definition; if we were to list out the elements the list would read:

1. 0
2. 1
3. 2
4. 3, and so on...

Furthermore, we can always say where each number would go in this list: 5 would be our sixth element, 9 would be our tenth element... you get the idea...

The set $S = \{0, 2, 4, 6, ...\}$, our beloved even numbers, is also infinite in this sense. Our list would be:

1. 0
2. 2
3. 4
4. 6, and so on...

Clearly, the position of $2n$ is $n+1$. Okay, so the essay is over; we fully understand infinity right? Obviously not! As it turns out, what we've described is the "countable infinity" and there exist more complex and, in some sense, infinite types of infinities. Georg Cantor first explored this notion, and he famously showed that the real numbers $\mathbb{R}$ are not countably infinite.

Cantor employed a proof by contradiction. In essence, he assumed the opposite of what he was trying to prove and then showed that that assumption leads to an absurdity.

So, how exactly did he do it? Well, let's find out. First, we should focus our attention on $(0,1)$. After all, if the number of reals between 0 and 1 is uncountably infinite, then we're done.

To show that there are uncountably many reals between 0 and 1, we'll use a proof by contradiction (what a surprise). Let's assume that there is some list of our reals, here's how you can construct a number n that's never on the list(the numbers below are completely arbitrary):

Index	**Entry in our list**
1	$0.\textcircled{1}1011\ldots$
2	$0.1\textcircled{4}340\ldots$
3	$0.99\textcircled{9}99\ldots$
4	$0.789\textcircled{7}8\ldots$
5	$0.1234\textcircled{5}\ldots$
$\vdots$	$\ddots$

The idea is for our n to differ with entry i at its ith decimal place (the circles). In this case, the n would thus look like:

$$n = 0.\ \underset{\overset{\uparrow}{1+1}}{2}\quad \underset{\overset{\uparrow}{1+4}}{5}\quad \underset{\overset{\uparrow}{9-1}}{8}\quad \underset{\overset{\uparrow}{1+7}}{8}\quad \underset{\overset{\uparrow}{1+5}}{6}\cdots$$

Clearly n can't be on the list; it differs with each entry!

So, there you have it: multiple types of infinity! This type of argument is called a diagonalization.

Now, the pessimist inside you could be thinking that, while cool, this idea isn't practical. It seems like it's just limited to the foundations of mathematics and wouldn't find much of a home within the rest of the subject. In fact, maybe Cantor's questions about infinity weren't so useful. Well, as it turns out, Cantor was indeed very correct in posing these questions, and we will see how his idea of diagonalization isn't purely limited to set theory.

§4 Act II: A Time Skip to Computer Science

Let's fast forward: it's the 1960's and computer science is booming. In fact, not only are computers getting faster, but mathematicians are making many big discoveries in theoretical computer science. But, how can we analyse computers in a theoretical manner? Well, as usual, mathematicians have invented a myriad of terms which let us reason about computers. Let's see some:

Definition 4.1. An alphabet is some finite set of symbols.

Bear with me here, I promise this will be useful... To make the definition less opaque, here are some examples: our every day alphabet would be the set $A = \{a, b, c, ..., z\}$, binary would be $A = \{0, 1\}$, and the decimal system would be $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Definition 4.2. A string is some sequence of symbols from a specific alphabet.

In simple terms a string is like a word. For example, a string over our standard alphabet $A = \{a, b, c, ..., z\}$ could be *complexity*. Likewise, if we use $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, 1434 could be a perfectly valid string. We can use variables to denote strings, and if we have a string s, $|s|$ is its length. For example, $|21849| = 5$.

Definition 4.3. A language is a set(not necessarily finite) of strings from a specific alphabet.

Here are some examples: If we have $A = \{a, b, c, ..., z\}$, the language L of all English words is perfectly fine. Note that in this case the language is finite.

For an infinite language, consider the set of all positive even numbers; this is a language with alphabet $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Ok, so now that we have these definitions, let's talk computers! As we all know from our everyday lives, computers solve problems. How can we rigorously define the notion of a problem? Well, here's where our earlier definitions come in handy:

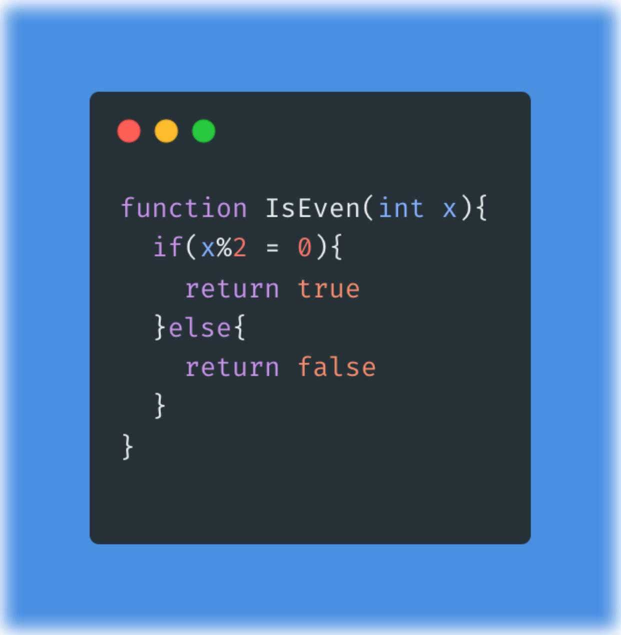
Definition 4.4. A decision problem is the problem of determining whether a string s lies in a language L, where both have a common alphabet A. We call the decision problem D_L.

You get the drill, so here's an example: If we consider $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and L as the set of all positive even numbers, the decision problem D_L would be the problem of determining if a given non negative integer is even.

Now for the punchline: we view computation as algorithms to solve decision problems. Note that the output of the algorithm would be 1(or true) if the string is in the language, and 0(or false) otherwise.

How do we specify algorithms? Well, mathematicians look at Turing machines and other models, but an intuitive way to understand it is by looking at computer code. We can mathematically analyse the run time of some pseudocode which defines a function which, in turn, solves a decision problem. When we analyse run time, we look at it in terms of input length(as defined when we explained strings). Our units of time can be seen as the basic operations undertaken when executing the code(see more in Stanford Info Lab).

Do note that what constitutes a basic operation can vary depending on who you talk to, but, as you will see, this point isn't too important. It's best to understand this notion of run time intuitively, so let's look at some pseudocode which solves the even numbers decision problem:



```
function IsEven(int x){
  if(x%2 = 0){
    return true
  }else{
    return false
  }
}
```

Figure 1: Made using Carbon

On an input x, the code above executes one test(% means remainder) on x and then executes a return statement, so we'd say its runtime is 2 regardless of x. If we were to write it as a function, we would say $R(x) = 2$. Now: the choice of considering the remainder as one operation is definitely arbitrary. This issue is why we tend to use Big O notation when talking about run time; as you will see, big O notation would give very similar results regardless of what we consider as one step(as long as our considerations are reasonable!).

Big O notation serves to describe the limiting behavior of a function. Essentially, it describes how a function roughly behaves as its input approaches infinity. Here's a precise definition:

Definition 4.5 (Wikipedia)**.** $f(x) = O(g(x))$ if there exists some x_0 and c such that

$$|f(x)| \leq cg(x)$$

for all $x \geq x_0$

Clearly, as long as we're reasonable, what constitutes a basic operation should only change the runtime by a constant multiple, so big O notation solves our problem of arbitrariness. Looking back to our even number program, it's clear that its run time is $O(1)$(Note: when dealing with a Turing machine things may be slower, but the slowdown doesn't affect the main result we'll see). Let's consider a more complicated example. Consider $f(x) = 3x^2 + 4x + 1$, we can say $f(x) = O(x^2)$, because after a certain point $3x^2+4x+1 \leq 4x^2$, as shown in the graph below, where blue is $4x^2$ and red is $3x^2+4x+1$:

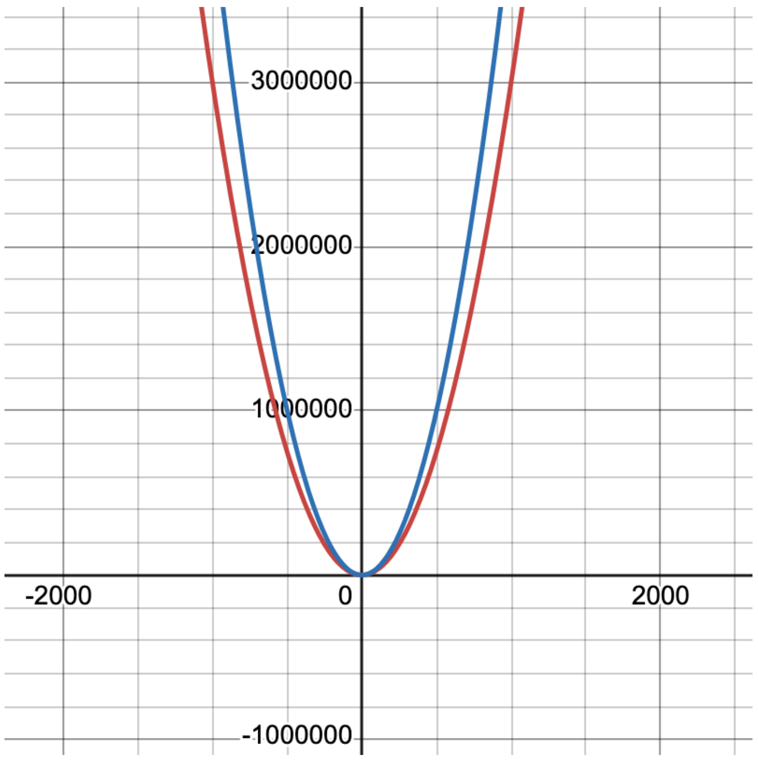


Figure 2: Made using Desmos

Now that we have everything necessary to understand run time, let's look at two complexity classes. A complexity class is a set of decision problems which satisfy a specific property.

Definition 4.6 (Polynomial Time)**.** The class P consists of all decision problems D such that there exists a computer program C where the run time of C is $R(x) = O(x^n)$ for some fixed n.

Roughly, this class includes all problems which are efficiently solvable. Its called polynomial time because the run time is bounded above by a polynomial. Of course our even numbers code is an example of polynomial time, but there are others, such as determining whether a graph is connected(think about why!).

Definition 4.7 (Exponential Time)**.** The class EXP consists of all decision problems D such that there exists a computer program C where the run time of C is $R(x) = O(2^x)$.

First think about why the 2 is unimportant and can be replaced with any positive real number. Now, obviously all problems in P are in EXP. But there are also some problems where a P solution isn't obvious but an EXP one is. Many games can be placed in EXP; intuitively this makes sense, as when you analyse a board, the number of situations increases exponentially with the number of moves made. This begs the question: "Can we rigorously show that there are some EXP problems not in P?"

§5 Act III: The Dramatic Reveal

Now that we have all the ingredients, let's finish this. We will prove that $P \neq EXP$, or in other words, there exists a problem in EXP which isn't in P.

Consider the program A which does the following on input x:

- Let P_x be the computer program defined by x, we will run $P_x(x)$ for $2^{|x|}$ steps
- If this process doesn't terminate or P_x doesn't correspond to a valid program, output 0

- If running P_x outputs 1, output 0
- If running P_x outputs 0, output 1

Clearly this program corresponds to a decision problem D. We won't prove that D is in EXP, but this should be intuitive. In a rigorous setting, you need to do some housekeeping to show this, but it's nothing mind blowing. The cool part is showing that D isn't in P.

We hearken back to the proof by contradiction: assume that there is a computer program P_B represented by string B which has a runtime of cx^n, thereby placing D in P. Clearly, if the length of P_B's code is large enough, then $c|B|^n \leq 2^{|B|}$, meaning that A can fully simulate P_B.

Thus, we pad B's length until this is the case. Then, we feed B to A. clearly $A(B)$ will be the opposite of $P_B(B)$, by definition. This is paradoxical, since B and A are meant to have the same output; the only solution is that B can't exist, meaning that D isn't in P!(This is a modified version of Boaz Barak's and Sanjeev Arora's proof of the time hierarchy theorem)

Now, it may not be clear, but this proof uses the exact same diagonalization argument as Cantor, look:

Consider an enumeration of polynomial time programs P_1, P_2, P_3, and so on. We will construct a similar table as we did for our proof about real numbers:

Inputs	P_1	P_2	P_3	P_4	P_5	$\cdots$	P_n	$\cdots$
Programs:P_1	$\mathbf{P_1(P_1) = 1}$	$P_1(P_2) = 1$	$P_1(P_3) = 1$	$P_1(P_4) = 1$	$P_1(P_5) = 1$	$\cdots$	$P_1(P_n) = 1$	$\cdots$
P_2	$P_2(P_1) = 0$	$\mathbf{P_2(P_2) = 0}$	$P_2(P_3) = 0$	$P_2(P_4) = 0$	$P_2(P_5) = 0$	$\cdots$	$P_2(P_n) = 0$	$\cdots$
P_3	$P_3(P_1) = 1$	$P_3(P_2) = 1$	$\mathbf{P_3(P_3) = 0}$	$P_3(P_4) = 0$	$P_3(P_5) = 0$	$\cdots$	$P_3(P_n) = 0$	$\cdots$
P_4	$P_4(P_1) = 0$	$P_4(P_2) = 1$	$P_4(P_3) = 1$	$\mathbf{P_4(P_4) = 1}$	$P_4(P_5) = 1$	$\cdots$	$P_4(P_n) = 1$	$\cdots$
P_5	$P_5(P_1) = 0$	$P_5(P_2) = 1$	$P_5(P_3) = 0$	$P_5(P_4) = 1$	$\mathbf{P_5(P_5) = 0}$	$\cdots$	$P_5(P_n) = 0$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	
P_n	$P_n(P_1) = 1$	$P_n(P_2) = 0$	$P_n(P_3) = 1$	$P_n(P_4) = 1$	$P_n(P_5) = 1$	$\cdots$	$\mathbf{P_n(P_n)} = 1$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\ddots$

Our program A is constructed so that:

$$\begin{aligned}
A(P_1) = 0 \neq P_1(P_1) \\
A(P_2) = 1 \neq P_2(P_2) \\
A(P_3) = 1 \neq P_3(P_3) \\
A(P_4) = 0 \neq P_4(P_4)
\end{aligned}$$

$$A(P_5) = 1 \neq P_5(P_5)$$
and so on...

As you can see, we're using the exact same motif Cantor used in his proof! It's diagonalization with some extra spice!

§6 Epilogue: Why Math?

It's clear to me that Cantor did not intend for his diagonalization argument to be used in theoretical computer science decades after his death. However, a question which he posed lead to exactly that.

Cantor's investigations into set theory were the right questions because he dove into something unknown; he didn't care about applications, he just wanted to understand the mathematical world. In my opinion, the applications don't need to drive mathematical discover; you should study something for its intrinsic elegance. So, dive into any math problem which you find interesting; your ideas might be useful 100 years later!

§7 Further Reading

I highly recommend reading Introduction to the Theory of Computation by Michael Sipser. It will introduce all the rigour necessary to be able to understand the full versions of the ideas presented in this essay. Once you've read this, you should look at Computational Complexity: A Modern Approach by Boaz Barak and Sanjeev Arora for a challenging, but rewarding read.

§8 References

`https://en.wikipedia.org/wiki/Big_O_notation`

`http://infolab.stanford.edu/~ullman/focs/ch03.pdf`

`Computational Complexity: A Modern Approach by Boaz Barak and Sanjeev Arora`