The Entscheidungsproblem and Alan Turing

Author: Laurel Brodkorb

Advisor: Dr. Rachel Epstein

Georgia College and State University

December 18, 2019

1 Abstract

Computability Theory is a branch of mathematics that was developed by Alonzo Church, Kurt Gödel, and Alan Turing during the 1930s. This paper explores their work to formally define what it means for something to be computable. Most importantly, this paper gives an in-depth look at Turing's 1936 paper, "On Computable Numbers, with an Application to the Entscheidungsproblem." It further explores Turing's life and impact.

2 Historic Background

Before 1930, not much was defined about computability theory because it was an unexplored field (Soare 4). From the late 1930s to the early 1940s, mathematicians worked to develop formal definitions of computable functions and sets, and applying those definitions to solve logic problems, but these problems were not new. It was not until the 1800s that mathematicians began to set an axiomatic system of math. This created problems like how to define computability. In the 1800s, Cantor defined what we call today "naive set theory" which was inconsistent. During the height of his mathematical period, Hilbert defended Cantor but suggested an axiomatic system be established, and characterized the Entscheidungsproblem as the "fundamental problem of mathematical logic" (Soare 227). The Entscheidungsproblem was proposed by David Hilbert and Wilhelm Ackerman in 1928. The Entscheidungsproblem, or Decision Problem, states that given all the axioms of math, there is an algorithm that can tell if a proposition is provable. During the 1930s, Alonzo Church, Stephen Kleene, Kurt Gödel, and Alan Turing worked to formalize how we compute anything from real numbers to sets of functions to solve the Entscheidungsproblem. The Entscheidungsproblem was shown to be unsolvable by Alonzo Church and Alan Turing separately. Alonzo Church began his work using Lambda Calculus to prove that the Entscheidungsproblem is unsolvable, in that there does not exist a procedure to tell if a proposition has a proof or not. Turing also proved the Entscheidungsproblem is unsolvable, but separately, using Turing Machines. To solve the Entscheidungsproblem, we needed a formal definition of computability, which became the Church-Turing Thesis.

3 Mathematical Background

3.1 Algorithms and Computable Functions

Let us define what an algorithm is. Informally, an algorithm is a finite deterministic process. Note that this is a very vague definition that encaptures a lot of things. The next thing to define is our intuitive definition of computability. Something is computable if an algorithm exists to accomplish the task desired. For example, a sequence is computable if we can build an algorithm that generates that sequence. We do not have any restriction on time and memory to ascertain if a process or a sequence is computable (Weber 2). Hence, we are allowed unlimited time and memory when checking to see if a process or a sequence is computable. However, in order for something to be computable, it must use a finite amount of time and memory (Weber 2). This brings up the difference between unlimited and infinity. For example, consider the set of all positive numbers. The numbers as a set do not have a bound. However, the set of all the numbers does not contain infinity since infinity is not a number. The difference here is that infinite is something that we can never properly reach while unlimited means there is no bound.

This leads to the premise of how we classify computable functions. A *computable function*, also called a *total computable function*, is any function that yields the same output no matter how many times you run the algorithm, for a given input. Furthermore, the domain of a total computable function is the natural numbers. Note that we must have an algorithm in order for the function to be deemed computable. This is a key point: we must satisfy that such an algorithm exists in order to deem that a function is total computable. A *partial computable function* is any function that is not necessarily defined for every input, provided that there exists an algorithm for that function. Furthermore, the domain of a partial computable function a subset of the natural numbers. Hence, all computable functions are partial computable functions, but not all partial computable functions are total computable functions. However, since we removed the restriction of time and memory, we could spend forever checking to see if a partial computable function is a computable function. Thus, there is never a way to tell if a function is computable or merely partial computable. We will prove this formally later.

We will now talk about the Church-Turing Thesis. The Church-Turing thesis states that lambda calculus, Turing machines, and general recursive functions are equivalent definitions for computability. Most importantly, the Church-Turing thesis says that all three are correct definitions of computability and they are what we intuitively think the definition of computability should be. In the following sections we will define and examine lambda calculus, general recursion, and Turing machines as these were the three best, and logically equivalent, definitions of computability.

3.2 Gödel's Work with Primitive Recursion and General Recursion

3.2.1 Recursive Functions

One way to define computable functions is by using recursive functions. Gödel's work to define computability using primitive recursion and general recursion was the first successful attempt made to define what it means for something to be computable. Copeland et al write, “The 1930s began with Gödel’s publication of his completeness theorem for first-order logic, and a year later, in 1931, he published his famous incompleteness results. The later concerned formal systems of arithmetic involving what we call primitive recursive axioms and inference rules. A first major step had been taken in the development of modern computability theory” (Copeland et al, viii). Gödel defined the class of recursive functions as follows:

Definition 1. [*Primitive Recursion*] The primitive recursive functions are the smallest class C of recursive functions such that the following hold:

1. The successor function $S(x) = x + 1$ is in C.
2. All Constant functions $M^n_m(x_1, x_2, ..., x_n) = m$ for $n, m \in \mathbf{N}$ are in C.
3. All projection (or identity) functions $P^n_i(x_1, x_2, ..., x_n) = x_i$ for $n \geq 1, 1 \leq i \leq n$, are in C.
4. (Composition, or substitution.) If $g_1, g_2, ..., g_m, h$ are in C, then $f(x_1, ..., x_n) = h(g_1(x_1, ..., x_n), ...g_m(x_1, ..., x_n))$ is in C, where the g_i are functions of n variables and h is a function of m variables.
5. (Recursion.) If $g, h \in C$ and $n \geq 0$, then the function f is defined below in C:

$$f(x_1, ..., x_n, 0) = g(x_1, ..., x_n)$$
$$f(x_1, ..., x_n, y + 1) = h(x_1, ..., x_n, y, f(x_1, ..., x_n, y)),$$
where g is a function of n variables and h is a function of $n + 2$ variables

Note that we used Weber’s notation for the above definition. The above are the five axioms of primitive recursion.

Let us talk about this class of functions. To start, the successor function is defined as $S(x) = x + 1$ in C. The successor function takes in x and adds one to it. For example, $S(0) = 0 + 1 = 1$. Furthermore, the constant functions are functions that output a constant value no matter what their inputs are. For example, $f(x) = 5$ is a constant function. Additionally, we have projection, or identity functions, where each function returns its ith input. We also have that the composition of functions in C are in C as well. In other words, if f and g are in C, then their composition will be in C. The following is an example of recursion. Let $n = 0$, $f(0) = 5$, $h(x_1, x_2) = x_1 + x_2$, and $f(y + 1) = h(y, f(y))$. Note that $f(1) = h(0, f(0)) = 0 + f(0) = 0 + 5 = 5$.

Note that all primitive recursive functions are total computable. Primitive recursion does not allow us to do some things that we thought to be computable. For example, let $f(x) = 1$ if there exists a twin prime greater than x. While we do not have proof that the number of twin primes are infinite, the process to find a twin prime greater than x should be partial computable. That is, we can start listing out all the primes until we find a twin prime larger than x. Hence, this process is computable. For example, we can find a twin prime greater than 7, that is, we can find 11 and 13 by checking that they are prime and twin primes. Note that we do not have that there is a twin prime greater than x for all x, since that has not been proved yet, but we can keep searching forever. However this process is not primitive recursive. An example of a function that is not primitive recursive, but is total computable is the Ackerman function. Weisstein writes, “The Ackermann function is the simplest example of a well-defined total function which is computable but not primitive recursive, providing a counterexample to the belief in the early 1900s that every computable function was also primitive recursive". Because of this function, the details of which are beyond the scope of this paper, mathematicians were convinced that something needed to be added to primitive recursion in order to capture everything we thought to be intuitively computable.

Gödel presented the concept of a general recursive function in 1934 (Copeland et all, viii). Gödel added a sixth axiom to the established five axioms above, an unbounded search axiom to establish general recursion. An unbounded search means that we can search forever for an output to any given function, and possibly never find the output so the function is undefined on that input.

Definition 2 (*General Recursion*)**.** The general recursive functions are the smallest class C of functions such that the previous five axioms, given in Definition 1, along with the following axiom hold.

6th Axiom

Assume $f(x_1, \cdots x_n, y)$ is a general recursive function in C. We define a function g such that $g(x_1, \cdots x_n, y)$ =least y such that $f(x_1, \cdots x_n, y) = 0$, and for all $z \leq y$, $f(x_1, \cdots x_n, y)$ is defined. Then g is in C.

The following is an example of unbounded search, using our formal definition.

Let $f(x, y) = \begin{cases} 0, & \text{if } y > x \text{ and } y \text{ is an even prime;} \\ 1, & \text{otherwise.} \end{cases}$

Note that $f(2, 5) = 1$ and $f(0, 2) = 0$. Note that f is a primitive recursive function.

Define $g(x) = \Big\{$ least y such that $f(x, y) = 0$ and for all $z \leq y$, f(x,z) is defined.

Definition 3. A function *diverges* on an input if there is no defined output.

It follows that $g(0) = 2$, $g(1) = 2$, and $g(2)$ diverges. Note that $g(2)$ diverges because 2 is the only even prime, and 2 is not less than itself. Note that g diverges for all inputs greater or equal to 2. However, g will still search to find an even prime greater than its input forever. Hence, it is a general recursive function. However, it is not a primitive recursive function, since it is not total computable.

General recursive functions are an example of intuitively computable functions. The work in using recursive functions to try to define computable functions was started by Gödel, and continued by Church and Stephen Kleene. To summarize, Gödel defined primitive recursion in 1931, and then recursion in 1934, in order to formalize what it means for a function to be computable. Soare writes, "by the beginning of 1936, Gödel was not convinced that his own recursive functions captured the [intuitively] calculable functions. However, when Gödel saw the following analysis by Turing later in 1936, he was immediately convinced" (Soare 5). We will see Turing's definition of computable in a later section.

3.3 λ Calculus

3.3.1 Introduction

Church suggested that lambda definability be taken as a precise definition of computability. Additionally, lambda definable functions are exactly the general recursive functions. His work with lambda calculus was done by himself and his graduate student Stephen Kleene. They were working to formally define what it means for something to be computable and what it means for something to be incomputable. Church's suggestions are mathematically equivalent to a Turing machine, and are now known collectively as the Church-Turing Thesis (Weber 65). The Church-Turing thesis states that lambda calculus, Turing machines, and general recursive are equivalent definitions for computability, and more importantly that they are the correct definitions of computability.

Lambda calculus is the most basic programming language, which can be written and solved using only pencil and paper. Remember, at this time computing machines were over a decade away. Lambda calculus runs on substitution. For example, $(\lambda x.xxz)y = yyz$. Here we are told to replace every instance of x in xxy with y, so we get yyz. If we are not told what we are substituting in with, it is a function. An example of this is $\lambda x.x$. This means "for every instance of x in x". Note that we are not told what our input is. Let us examine the identity function $(\lambda x.x)$ for more explanation. To start, $(\lambda x.x)y$ is equal to y since it replaces every x in x with y. In other words, the function returns what one puts in, in this case y. Note that we do not define a domain or a codomain. This is consistent for all lambda definable functions.

We could also have the expression $(\lambda x.y)z$, which equals y. This happens because there is not an x in y with which to replace z and therefore returns what was already there. The x is the variable, y is what the function does to the input, and z is the input. This function, $(\lambda x.y)$ is called the constant function.

Let us look at an example with two variables. To start the rule is the following: $(\lambda xy.E)AB = ((\lambda x.(\lambda y.E))A)B$, where E, A,and B are expressions. In other words we are expanding out the expression. For example, let us look at $(\lambda zx.tzx)sk$:

$((\lambda z.(\lambda x.tzx))s)k$ `Expanded the expression out`

`Our instructions say to replace` z `in` $(\lambda x.tzx)$ `with` s. `This yields the following:`

$(\lambda x.tsx)k$ `Here we are told to replace` x `in` tsx `with` k. `This yields the following:`

tsk.

The last line is our final answer. So the first variable, z was replaced with the first input, s, and the second variable, x was replaced with the second input, k.

3.3.2 Bound and Free Variables

Another important piece of lambda calculus is the difference between a free variable and a bound variable. The x in $\lambda x.y$ is bound since it appears next to λ, and the y is free. Bound variables are those that appear next to a λ and they are also the variables in the expression that are going to get replaced. For example, $(\lambda x.xxy)zx$. The first three x's are bound, while the last x is free. Note that the y is free because it is not going to get replaced. This concept of free and bound variables is important because you cannot input a free variable and turn it into a bound variable. In order to preform some substitutions we need to create a dummy variable. For example, let us look at $(\lambda x.(\lambda y.xy))y$. Here we can see that the first and second y's are bound, because y is in a position to get replaced in xy. Note that the third y is free. If we were to make the substitution, we would have the following:

$$\lambda y.yy.$$

Because lambda calculus doesn't depend on the name of variables, we should get equivalent answers no matter how the variables are named.

So, we rename the bound variable as t and make the substitution as follows:

$$\begin{aligned}&(\lambda x(\lambda t.xt))y\\&= (\lambda t.yt).\end{aligned}$$

(Rojas 4).

Note that the two answers are not the same. The $(\lambda t.yt)$ is the correct result because we have no confused bound and free variables. Furthermore, we would get this answer no matter how the variables are named so long as we do not confuse bound and free variables.

3.3.3 More Difficult Example

Let us look at a more complicated example, $(\lambda xy.yxx)(\lambda z.yz)(\lambda rs.rs)$. Note that we can expand the double variable to get the following: $(\lambda x.(\lambda y.yxx))(\lambda z.yz)(\lambda rs.rs)$.

Note that if we followed the instructions given,we would confuse bound and free variables. However, let us carry out those instructions in order to see what goes wrong:

$[\lambda y.y(\lambda z.yz)(\lambda z.yz)](\lambda rs.rs)$ `Here we replaced each` x `in` $(\lambda y.yxx)$ `with` $(\lambda z.yz)$`.`

Note that all the y's are bound now. Before the third y was free, but now it is bound because it has turned into two bound y's. We should have created a dummy variable and then carried out the instructions. Creating a dummy variable w, we have the following:

$$\begin{aligned}
&\Rightarrow [(\lambda x.(\lambda w.wxx))(\lambda z.yz)](\lambda rs.rs) \texttt{ Here we have created a dummy variable } w\\
&\Rightarrow [\lambda w.w(\lambda z.yz)(\lambda z.yz)](\lambda rs.rs) \texttt{ Here we have replaced } x \texttt{ in } (\lambda w.wxx) \texttt{ with } (\lambda z.yz)\\
&\Rightarrow (\lambda rs.rs)(\lambda z.yz)(\lambda z.yz) \texttt{ Replaced } w \texttt{ in } w(\lambda z.yz)(\lambda z.yz) \texttt{ with } (\lambda rs.rs).\\
&\Rightarrow (\lambda r.(\lambda s.rs))(\lambda z.yz)(\lambda z.yz) \texttt{ Here we have expanded out } (\lambda rs.rs)(\lambda z.yz)(\lambda z.yz)\\
&\Rightarrow (\lambda s.(\lambda z.yz)s)(\lambda z.yz) \texttt{ Substituting } (\lambda z.yz) \texttt{ in for } r\\
&\Rightarrow (\lambda z.yz)(\lambda z.yz) \texttt{ Substituting } (\lambda z.yz) \texttt{ in for } s\\
&\Rightarrow y(\lambda z.yz) \texttt{ Replacing } z \texttt{ in } yz \texttt{ with } (\lambda z.yz).
\end{aligned}$$

The above example was taken from *Computability Theory* by Rebecca Weber. However, it was solved differently, including correcting a small mistake in naming variables. Weber mistakenly made the dummy variable replace the unbound y, and while she got the same answer, this would not be true if we had more unbound ys in the problem. Note that the last line is our final step since we are not given anything with which to replace z in yz with. Observe that this example highlights the importance of not confusing bound and free variables.

3.3.4 Importance of Lambda Calculus

Lambda calculus is more than notation. The integers can be represented in lambda calculus. Let us look at how this representation works. First, $\lambda sz.z$ is the function on inputs s and z that outputs z. The natural numbers are defined as follows:

$$\begin{aligned}
0 &\equiv \lambda sz.z\\
1 &\equiv \lambda sz.s(z)\\
2 &\equiv \lambda sz.s(s(z))\\
3 &\equiv \lambda sz.s(s(s(z)))\\
&\vdots
\end{aligned}$$

Note that 0 is defined as a function that takes in two variables and outputs the second input. We have defined 0 as this absurd function; however, it makes everything else work which is why we have done so. The successor function is defined to be $S = (\lambda wyx.y(wyx))$. Let us look at $S(0)$, which is the successor of zero. We know that the successor of zero is one, so this is an easy example to check. Note the following:

$$\begin{aligned}
S(0) &\equiv (\lambda wyx.y(wyx))(\lambda sz.z) \texttt{ Here we are inputting } (\lambda sz.z) \texttt{ into } (\lambda wyx.y(wyx)) \\
&\equiv \lambda yx.y((\lambda sz.z)yx) \texttt{ Replaced } w \texttt{ in } y(wyx) \texttt{ with } (\lambda sz.z) \\
&\equiv \lambda yx.y((\lambda z.z)x) \texttt{ We are replacing all the } s\texttt{'s in } z \texttt{ with } y \texttt{ so the } s \texttt{ and } y \texttt{ go away.} \\
&\equiv \lambda yx.y(x) \texttt{ Note that the } z \texttt{ in } z \texttt{ is replaced with } x. \\
&\equiv \lambda sz.s(z) \texttt{ Note } \lambda yx.y(x) \texttt{ is equivalent to } \lambda sz.s(z) \texttt{ by changing the names of the variables.} \\
&\equiv 1. \quad \texttt{Note that } \lambda sz.s(z) \texttt{ is equivalent to } 1.
\end{aligned}$$

What we have demonstrated here is the use of the successor function, showing $S(0) = 1$, as desired. We do not need to define the successor function in order to have the equivalent of the natural numbers in lambda calculus. However, we do need the successor function so that we can do operations with the natural numbers as we would expect.

Additionally, we can define addition, subtraction, and multiplication in this system. It is important to note that lambda calculus is also capable of doing recursion, and it is capable of creating any function that Gödel's general recursion is capable of creating, and lambda calculus is equivalent to a Turing machine. Additionally, lambda calculus can be used to encode logic operators as follows:

$$\begin{aligned}
\texttt{True:} &\quad T = (\lambda xy.x) \\
\texttt{False:} &\quad F = (\lambda xy.y) \\
\texttt{And:} &\quad (\lambda zw.zwF) \\
\texttt{Or:} &\quad (\lambda zw.zTw) \\
\texttt{Not:} &\quad (\lambda z.zFT).
\end{aligned}$$

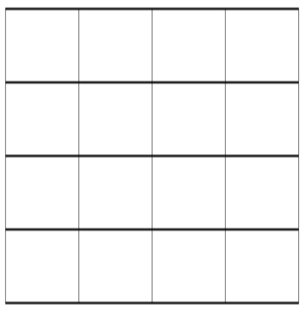
In conclusion, lambda calculus established a way of formally solving computing problems. It allowed Church and Kleene to say that something is computable if and only if it is definable through lambda calculus. Essentially, lambda calculus is notation, arithmetic, and is capable of coming up with the same functions as a Turing machine, although that was not demonstrated until Turing's paper on computable numbers. Lambda calculus was used to show that the Entscheidungsproblem has no solution by Church and Kleene.

4 Introduction to Turing's 1936 Paper

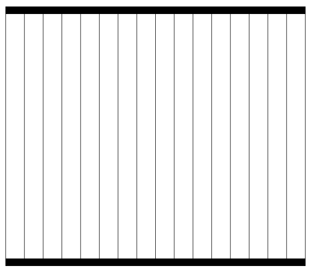
In Alan Turing's 1936 Paper "On Computable Numbers, with an Application to the Entscheidungsproblem", Turing sought to define computable using an ideal computing machine, a Turing Machine. This was an entirely new concept that drastically changed how we solve problems today. For a large portion of his paper, Turing works to develop a system of writing logical statements into notation that is comparable to Turing machines. In other words, he is the first to take formal logic and apply it to an ideal computing machine. Soare writes, "Turing's 1936 paper is probably the single most important paper in computability. It introduces the Turing machine, the universal machine, and demonstrates the existence of undecidable problems. It is most often used in mathematics and computer science to define computable functions. It is perhaps Turing's best known and most influential paper" (xix).

4.1 Description of a Turing Machine

Recall your early mathematics classes when you did arithmetic in grids, performing only one operation at a time. Turing uses this to build an intuitive understanding of his machines. Turing writes, "computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character is sometimes used. But such a use is always avoidable, and I think that it will be agreed upon the the two-dimensional character of paper is no essential of computation" (249). Essentially, Turing is reminding the reader of their experience with sums and charts, like the one below:



And saying that it is equivalent to the picture below:



We are taking a two dimensional grid and making it into a one dimensional tape, and saying that these can be used in the same way. We will call this figure a tape. Turing argues that there is only a need for finitely many symbols. He argues this because, "it is always possible to use sequences of symbols in place of a single symbol" (249). For example, the number 17 is comprised of two symbols, 1 and 7, but we read it as one symbol in its own right. Hence, we could have a long strand of symbols be taken to be one symbol, like the number 4110126212611113. He further writes, "the behavior of the computer at any moment is determined by the symbols he is observing, and his 'state of mind' at that moment" (250). We can liken this to us reading one symbol at a time and only accomplishing one task based on what we are reading. He also says, "let us imagine the operations performed by the computer to be split up into 'simple operations' which are so elementary that it not easy to imagine them further divided" (250). In other words, we will only be performing one step at a time. He adds to this with, "we may suppose that in a simple operation not more than one symbol is altered". This is not to say that we cannot do several things in one action, but that we may only alter one cell on the tape at a time. He writes:

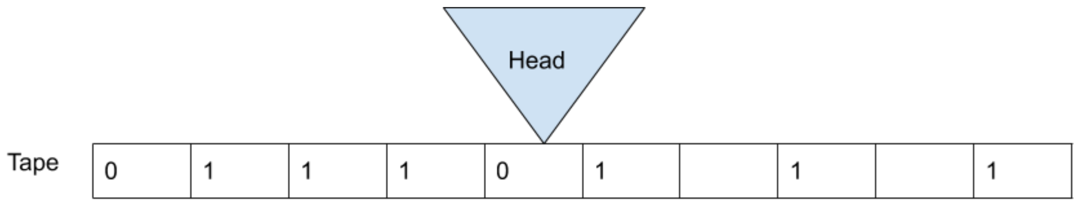
> The simple operations must therefore include:
>
> 1. Changes of the symbol on one of the observed squares.
>
> 2. Changes of one of the squares observed to another square within L squares of one of the previously observed squares.
>
>   It may be that some of these changes necessarily involve a change of state of mind. The most general single operation must therefore be taken to be one of the following:
>
> 3. A possible change of (*1*) of symbol together with a possible change of state of mind.
>
> 4. A possible change (2) of observed squares, together with a possible change of state of mind

(250). In summary, we can make a change of one of the observed cells, or move. We can change a symbol and change the state of mind, or change what cell the machine is observing and change the state of mind.

A Turing Machine is an ideal way of computing functions. A Turing machine has a tape, like the one pictured above, that runs through a reader, or a head, and prints symbols on the right side of the tape with the left side of the tape being the memory or scratch paper. This means that the right side of tape is the output of the machine. Furthermore, the difference between the right side of the tape and the left side of the tape is dictated by where the head starts. To the left of the cell where the head starts is the left side of the tape. The cell in which the head starts is the beginning of the right side of the tape.

In addition to its physical properties, a Turing machine has a program which consists of a finite list of states, a finite list of symbols, and a finite list of instructions. Note that the Turing machine can go to the left, and make changes to the tape as well, as we discussed above. We can liken this to a human reading one symbol at a time, and then writing down an instruction on the right side of the tape if instructed. We specify Turing machines via quadruples $< a, b, c, d >$ of instructions. First, a is the state the Turing machine is currently in. Second, b is the symbol the Turing machine's head, or reader, is currently reading. Third, c is the instruction to the head to write or move. Note that there may be multiple instructions for c that are preformed sequentially. In our definition above, c contains multiple instances of 1 or 2. Fourth, d is the state, also called the $m-$configuration, the Turing machine is in at the end of the instruction's execution. Turing uses $m-$configuration instead of state, so we will too. The $m-$configuration specifies what the machine is to do during various times of its actions. The machine's operations are dependent on what $m-$configuration it is in at that time. It is not necessary that there be any instruction at all, which may prevent the computation from continuing. In keeping with our definition of computable, we must allow the Turing machine unlimited time and memory, but not infinite, which is a very technical distinction, but a very important one.

Below is a simple diagram of a Turing machine.



An automatic machine is a machine that is completely determined by pre-arranged instructions. It needs no further input to run the desired program. A function machine has an input.

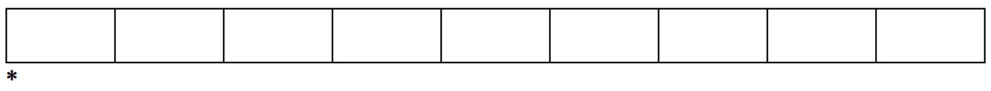
Let us look at a concrete example of a Turing Machine:

Table 1:

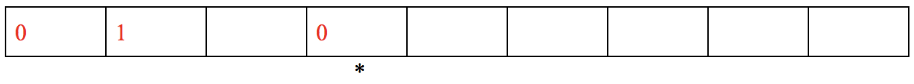
Configuration		Behavior	
m-configuration	symbol	operations	final m-configuration
b		$P0$, R, $P1$, R, R, $P0$	q
q	0	L, $P1$	m
m	1	R, R, $P0$, R, $P1$, R, R, R	b

Note that in the table we use "m-configuration" instead of state. These are the same thing.

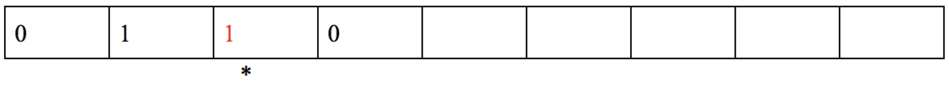
For our Turing machine, we have $P0$ to mean print 0, R to mean move right, L to mean move left, and $P1$ to mean print 1. So, the starting state is b. Below is a blank tape with the * showing where we start.



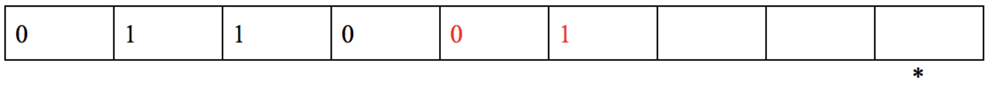
Below is a tape that shows the first step of instructions carried out:



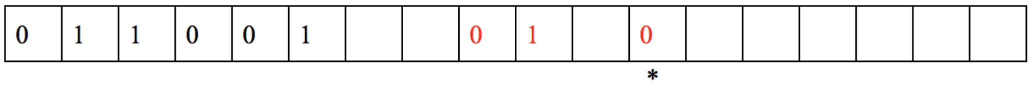
Note that because we have finished the instructions for $m-$ configuration b, we are in $m-$ configuration q. Hence, carrying out the instructions for $m-$ configuration q, we have the following. Note that the red symbols are the ones printed in this stage, where the black symbols are the ones printed in previous stages.



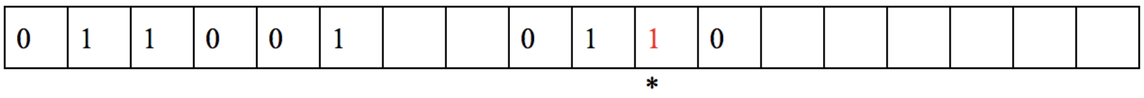
Note that because we have carried out the instructions for q, we are now in $m-$ configuration m. Hence we have the following:



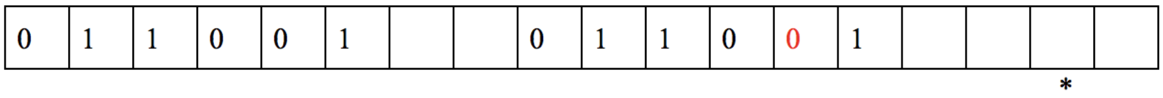
Note that we end up in the spot where the $*$ is underneath. Because we have finished the instructions for m, we are now in $m-$ configuration b. Hence we have the following:



Now we are in $m-$ configuration q. Note the following:



Note that we end where the last red zero is, and we are now in $m-$ configuration m. Note the following:



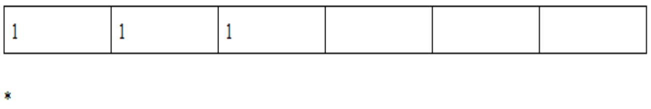
Note we are now in $m-$ configuration b. Hence, this machine is a loop that will run forever. This machine is an automatic machine since it did not need any input to start running. All machines in Turing's paper are assumed to be automatic. The sequence printed by the machine is $011001BB011001\cdots$, where the B's stand for blanks. If we remove the blanks and place a zero and a decimal before the first number in the sequence, we have the number computed by the machine. Hence, we have $0.011001011001\cdots$. Note that this number never ends.

We will now look at a function machine. A function machine must have an input and can only do something for one input at a time. For a function machine to stop, it must enter a halting state. For some machines, their halting states are never reached. Let us look at $f(x) = x + 3$. If we have input n for the function, we write that as $n+1$ ones. Hence if we have 2 as an input we write that as three ones. Hence, if we were inputting 0, we input 1 on the tape. The machine is described below: Let us look at the example $f(x) = x + 3$ for input 2. Note that

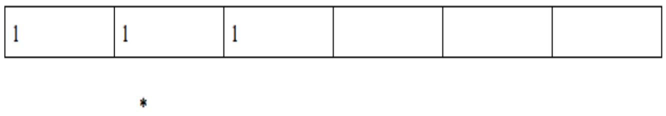
Table 2:

Configuration		Behavior	
$m-$configuration	symbol	operation	$m-$ configuration
m	1	R	k
k	1	R	k
	B	$P1$	g
g	1	$R, P1$	q
q	1	halt	

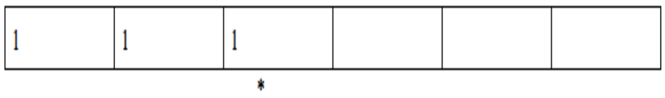
$f(2) = 2 + 3 = 5$. Using the Turing machine we have the following. Note that the *'s are used to show where the head is at. The first image below is the input written out on the tape.



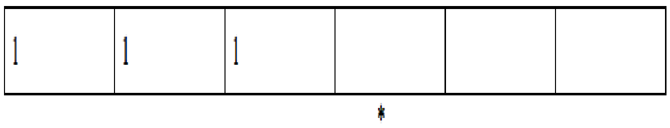
This is our input tape. We are starting in $m-$ configuration m.



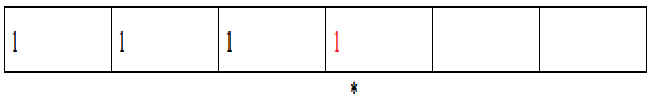
We end in $m-$ configuration k.



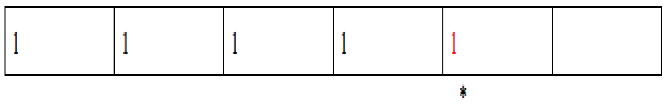
We end in $m-$ configuration k.



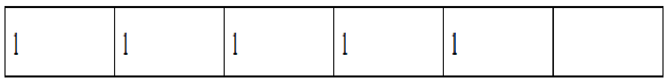
We now have a blank cell, so carry out the following instructions. We are still in $m-$configuration k.



We have printed 1. We end in $m-$ configuration g.



We have printed 1. We end in $m-$ configuration q.



This is the machine's output.

The red one is the character the machine writes. Since there are five 1's, the answer is five, which is what we expected.

To summarize, function machines differ from automatic machines in that they must have an input. If the machine halts, and does not enter into another $m-$ configuration, the machine converges on its input. If a machine diverges, it may loop through or stall at some states. The complete configuration of a stage of a machine is everything the machine is doing and has done (what has been printed out so far) at that stage of processing.

5 Advanced Logic with Turing Machines

A set is computable if its characteristic function is computable by a function machine.

5.1 Uniformity

Definition 4. A *characteristic function* checks if elements are in a set. Notationally, this means,

$$\chi_A(n) = \begin{cases} 1, & \text{if } n \in A \\ 0, & \text{otherwise,} \end{cases}$$

where A is the set in question, and n is some input for χ_A. For example, if we wanted to see which of the natural numbers are in the set of all even numbers, A, we would input them into the χ function. We would get $\chi_A(1) = 0$; $\chi_A(2) = 1$; $\chi_A(3) = 0$, and so on.

Definition 5. A computable set is a set that its characteristic function is computable by a function machine.

Another important definition is uniformity: "A uniform proof of computability explicitly builds a program to compute a function" (Weber 3). That is, for a function to be uniformly computable, there must be explicit instructions to compute the function. A nonuniform proof only proves that there exists a program that can compute the function. Let us look at a coin flip. Each flip can either be in set $\{heads\}$ or it can be in set $\{tails\}$, but not both sets at once. The sets are computable because we can easily write a program that outputs the sets because the sets are finite. However, there is no uniform proof to see in what set a coin flip will be, since the only way to prove which set a coin flip will be is to flip the coins. It is important to note that if the sets were infinite, as in we have an infinite number of coin flips, they are then not computable since the program that prints each set will not end, and we could not tell the outcome for any given flip.

5.2 Gödel Numbering and Enumeration of Computable Sequences

The goal of this section is to assign natural numbers to Turing machines, called Description Numbers. The way we do this is through Gödel numbering. Gödel numbering is a way to assign numbers to finite strings of symbols, in our case, these strings are the instructions of Turing machines. For example, let 1 represent 0, 2 represent 1, and 3 represent q_1, where 0, 1, and q_1 appear in some Turing machine. If we wanted to express the string $01q_1$ as a number, we would write $2^1 3^2 5^3$. If we wanted to do 011, we would write $2^1 3^2 5^2$. If we wanted to do $q_1 01 q_1 q_1$, we have $2^3 3^1 5^2 7^3 11^2$, which is some really big number. We are indicating the order of the sequence by using the n^{th} prime to indicate the n^{th} position. We can do this with logic symbols as well. Recall that Turing machines can be written as $< a, b, c, d >$, which then can written as a number, using Gödel numbering. Hence, every machine's instructions can be written as a number. Then we can write up entire sets of instructions as a number, which is the Description Number of the machine. We can do this for both function and automatic machines. The machine whose Description Number is n may be described as P_n.

Then we can make a universal function machine. This is denoted $U(n, m) = P_n(m)$, where n is the Description Number of an individual machine, and m is the input. Some of the time, the universal function machine will come up with nonsense and the machine will diverge.

Definition 6. Uniformly Effective Listing
A list of functions is called a *uniformly effective listing* if there exists a computable function that gives the outputs for any function in the list, say $g(n, x)$, that finds f_n and outputs the output of f_n for input x.

To each computable sequence, there corresponds at least one Description number. This means that the Description Numbers are not necessarily unique. The non-uniqueness comes from a computable sequence being the output of multiple different machines, each with its own Description Number. Because the numbers themselves are unique, natural numbers, we can list them out. All of the computable sequences can be listed uniformly since we can list all of the Description Numbers of automatic machines. Therefore, we can list all the partial computable functions because we can list the corresponding Description Numbers of the Turing machines. This list includes all the machines that are not circle free, which includes machines that do not print anything. The computable sequences and Description Numbers are therefore uniformly computable.

If a machine only prints 0's and 1's, it is called a computing machine. If we place a 0 and a decimal in front of the computed sequence, we have the number computed by the machine written in binary. Turing also adds, "A number is computable if it differs by an integer from the number computed by a circle-free machine" (233). Because there are countably infinite Turing machines, but uncountably many real numbers, many real numbers are not computable.

5.3 Unsolvable Problems

A computably enumerable set is any set that we can list, as in there are instructions for a program that can list out the set. A computably enumerable set is the domain of a partial recursive function. We will show that there exists sets that are computably enumerable, but not computable. The following example is adapted from Soare.

Let P_e be the partial computable function given by the Turing machine with description number e. Let W_e be the domain such that $W_e = dom(P_e) = \{x : P_e(x) \text{ converges } \}$. Note that when we say "converges", we mean the function is defined for that input. This means that the Turing machine halts for that input. Let $K = \{x | x \in W_x\} = \{x | P_x(x) \text{ converges } \}$. Note that K is all of the x's such that the x^{th} partial recursive function or the x^{th} Turing machine converges on x. This means that the x^{th} Turing machine halts on the x^{th} input.

Let K^c be the complement of K. We will show that K^c is not a computably enumerable set. Suppose $K^c = W_e$. Is $e \in W_e$? If yes, $e \in K^c$, so $e \notin W_e$, which is a contradiction. If no, $e \notin W_e$, which means that $e \notin K^c$, which means that $e \in W_e$, which is a contradiction. These two contradictions contradict that $K^c = W_e$, meaning that $K^c \neq W_e$ for every e so K^c is not a computably enumerable set.

We will also look at why K is not computable. Suppose the opposite that K is computable, then K^c is computable, since K^c is the complement of K. This is a contradiction because K^c is not computably enumerable because it is not equal to the domain of a partial computable function, which is what we showed earlier. Because K^c is not computably enumerable, it is not computable. Therefore, K is not computable. Recall that $K = \{x | x \in W_x\} = \{x | P(x) \text{ converges } \}$. We could define a machine to have K as its domain, as in create a machine with $K = \{x | x \in W_x\} = \{x | P_x(x) \text{ converges } \}$. That is, we want a machine, M, that only outputs when P_x converges for the x inputs. We can build so that the machine outputs only when $P_x(x)$ outputs. In other words, M is copying the action of P_x on input x. This how we build the domain of K. This means that K is exactly the domain of a Turing machine and therefore is computably enumerable.

6 Turing's Circle Free Machines and Application of Diagonalization Process

In this section we will prove that there is no general way to tell if a machine is circle free.

6.1 Diagonalization and Partial Functions

Theorem 1. *We cannot make a uniformly effective listing of all total computable functions.*

Proof. Suppose we could have a set c with these properties:

1. There is a uniformly effective listing of c.

2. Every function in c is a total computable function.

And that we have $c = f_1, f_2, f_3, \cdots$. Because set c has a uniformly effective listing, there exists a computable function $g(n, x)$ that finds f_n and outputs the output of f_n for input x. Note that we have not said that set c must contain all total computable functions, but that all functions in set c are total computable. In other words, not necessarily every total computable function is in c, but all functions in c are total computable. Below is a mockup of this list of functions:

Function	Outputs					
1	0	1	2	3	4	3 ⋯
2	1	4	6	3	4	1 ⋯
3	9	8	0	7	6	2 ⋯
4	0	0	1	0	1	0 ⋯
5	1	9	17	2	8	5⋯
⋮						

For example, take $f_5(4)$, which is the fifth function on the list and looks at the fourth output. Say that $f_5(4) = 2$, then $g(5, 4) = 2$. Looking at our example, if we look at $g(x, x)$ for inputs 1 through 5 are given by $g(1, 1) = 0$; $g(2, 2) = 4$; $g(3, 3) = 0$, et cetera. Hence, $g(x, x)$ =0, 4, 0, 0, 8 $\cdots$. Now we will define the function $h(x) = g(x, x) + 1$.

Therefore, $g(x, x)+1$ gives us 1, 5, 1, 1, 9$\cdots$, for our example. In other words, we are going across the diagonal and adding 1 to whatever output we have there. The new numbers, formed along the diagonal, are our function $h(x) = g(x, x) + 1$. Note that $h(x)$ is a total computable function since we know $g(n, x)$ is total computable.

If $h(x) \in c$, then $h = f_n$ for some n. Then we have $h(n) = f_n(n)$. We then have $f_n(n) = g(n, n)$. However, $h(n) = g(n, n) + 1$, which is a contradiction. Therefore, $h(n)$ is not in c. So, we know that c cannot be the total set of all computable functions, since we found a computable function not in c. In other words, we can never list out all the total computable functions. However, we can have a uniformly computable listing of all partial computable functions because we have a uniformly computable list of all Turing machines. Recall that we talked about this in the section on Gödel numbering. □

6.2 Turing's Proof that there is no complete list of all computable sequences

Turing proved a very important theorem, that there is no complete list of all computable sequences. This result will be used in later parts as part of a multiple-step process to show that the Entscheidungsproblem has no solution.

Theorem 2. *There is no uniformly computable list of all computable, infinite sequences.*

Proof. Let us assume that there is a list of some infinite computable sequences, called s. Let a_n be the nth computable sequence. Let $a_n(m)$ be the mth term of the sequence a_n. Note that $a_n(n)$ is the diagonal term. Below is a mockup of set s.

a_n	1	2	3	4	$\cdots$
a_1	0	1	1	0	$\cdots$
a_2	1	0	0	1	$\cdots$
a_3	0	0	1	0	$\cdots$
$\vdots$					

Note that in our example of set s, we used sequences of just zeros and ones, but it is similar for other sequences as well. In our example above, the sequence of diagonal terms is $001\cdots$. Let d be the sequence with nth term $1 - a_n(n)$. Observe that d is going along the diagonal and subtracting each value there from 1. In other words, d will change a 0 to a 1, and a 1 to a 0. In our example d=$110\cdots$. Note that d cannot be in set s because if $d = a_n$, then d=1-$a_n(n)$ and $d(n) = a_n(n)$. This is a contradiction. We cannot have d being equal to two different values. Therefore, because we have found a computable sequence not in s, we cannot have a complete, effective listing of all computable sequences. Thus, there is no way for us to generate a complete listing of all computable sequences. $\square$

It is helpful to note that this proof is very similar to the proof given by Cantor for there is no complete listing of all real numbers. Furthermore, this proof is very nearly the same proof given in the previous section. Only here we dealt with computable sequences, whereas in the previous section we dealt with computable functions.

6.3 Proving that we cannot tell if a machine will ever print a given number

In this section we will define what it means for a machine to be circle free, prove that there is no general process to tell if a machine will ever print a given number, and prove that there is no way to tell if a machine is circle free. Let us start with the definition of circle-free.

Definition 7. Circle-free.
If a machine only a finite number of 0's and 1's, it is *circular* (Turing 233). Otherwise, it is said to be *circle-free.* A sequence is said to be *computable* if it can be computed by a circle-free machine.

A number which is a Description Number of a circle-free machine will be called a *satisfactory number.* Later, we will show that there is no general process for determining whether a number is satisfactory or not.

Theorem 3. *There is no general process to tell if a machine will ever print a given number.*

Lemma 1. *There is no computable process to tell if a machine is circle-free.*

Proof to Lemma 1:

Proof. Assume that we can tell if a machine is circle-free. However, if we can tell if a machine is circle-free, then we can generate a list of all the circle-free machines. Because each machine is circle free, each machine gives a sequence of infinite zeros and ones, simply by outputting its sequence of infinite zeros and ones, and ignoring any other symbol it outputs. This contradicts that we can not make a list of all computable, infinite sequences (Theorem 2). Thus, there is no way to tell if a machine is circle free. $\square$

Proof of Theorem 3:

Proof. Let us assume that there is a way to tell if a machine ever prints a given number, when supplied with the Description Number of that machine. So, let us be given the Standard Description of an arbitrary machine M. Let M_1 "be a machine which prints the same sequence as M, except that in the position where the first 0 printed by M stands, M_1 prints $\bar{0}$. M_2 is to have the first two symbols 0 replaced by $\bar{0}$, and so on" (Turing 248). In other words, $M_1, M_2 \cdots$ are changing 0's to $\bar{0}$. For example, let M print $010100111\cdots$. Then M_1 would print $\bar{0}10100111\cdots$. Additionally, M_2 would print $\bar{0}1\bar{0}100111\cdots$. Let G be a machine that tests if there are any zeros in $M_1, M_2, \cdots$. More specifically, G computes the description number of M_1 and asks if M_1 prints a zero. If it does, G does not print anything. If M_1 does not print a zero, G prints a zero. Then, G moves on to test the next machine, M_2, in the same way. Machine G continues testing M_3, M_4, M_5 and so on. If "G never prints a zero, then M prints a zero infinitely often; if G prints a zero sometimes, then M does not print zero infinitely many often" (Turing 248). Note that G only prints a zero when some M_j does not print a zero; that is, when M_j either changed all the original 0 to $\bar{0}$, or never had any zeros to begin with. If G prints a zero, then M_j has finitely many zeros. If G never prints a zero, then all M_j do print a zero. Hence, if G never prints a zero, M prints a zero infinitely often. Therefore, if we have a general process to tell if G prints a zero, we have a general process to tell if M prints an infinite number of zeros. Similarly, we can test if M prints infinitely many ones. By these two processes, we can test if M prints an infinite amount of zeros and ones; that is, we can test if M is circle-free.

By Lemma 1, we know there is no way to tell if a machine is circle-free. Thus, we cannot tell if a machine ever prints a given number, when supplied with the Description Number of that machine.

□

7 Turing's Approach to the Entscheidungsproblem

The Entscheidungsproblem, or Decision Problem, says there is a computable process to tell us if there's a proof to a proposition. We can translate the Entscheidungsproblem into Turing machines. Turing created his machines with the goal that he could test to see if something is provable using his machines. In other words, a machine could print out a 1 if the statement was true, and a 0 if the statement was false. Turing says, "corresponding to each computing machine, M, we construct a formula $Un(M)$ and show that, if there is a general method for determining whether $Un(M)$ is provable, then there is a general method for determining whether M ever prints a zero" (Turing 259). In essence, we have the Entscheidungsproblem described using Turing machines. This will be broken into two lemmas, which is what Turing does in his 1936 paper.

The machine M is the input into a function Un. The output of $Un(M)$ is a formula in formal logic that is interpreted as "M prints a zero". The formula Un is complex, and we will not represent it here. Note that Un is not a machine. It only says that something prints a zero. Note that $Un(M)$ is not always true. Turing showed that we won't always know if $Un(M)$ is false.

We will start with the following two lemmas.

Lemma 1: If a zero “appears on the tape in some complete configuration of M, then $Un(M)$ is provable” (Turing 260). The proof is to use formal logic to model the configurations the machine, M, after assuming that zero appears on the tape. Since zero appears on the tape, and we would go through modeling the configurations of the machine in formal logic to show that, we then have a formal proof that the machine prints a 0. Hence, $Un(M)$ is provable.

Lemma 2: “If $Un(M)$ is provable, then [a zero] appears on the tape in some complete configuration of M” (Turing 260).

The proof goes as follows: assume $Un(M)$ to be provable. Note, $Un(M)$ written out in formal logic shows that there is a zero on the tape somewhere, provided that $Un(M)$ is true. Because we assumed $Un(M)$ to be provable, we have that it is true.

Proof. Entscheidungsproblem is Unsolveable in that there is no process to do what the Entscheidungsproblem says.

Assume that the Entscheidungsproblem is true. Let M be an arbitrary Turing machine. Let’s put $Un(M)$ into the Entscheidungsproblem machine. We have the following two situations. If $Un(M)$ is provable, M will print a zero (Lemma 2). The other situation is that $Un(M)$ is not provable. If it is not provable, then a zero will not appear on the tape of M, which is the contrapositive of Lemma 1. We therefore have a general process to see if a random machine prints a zero. We know this is impossible by Theorem 3. Therefore, there is no algorithm that can do what the Entscheidungsproblem proposed. Hence, the Entscheidungsproblem is unsolvable. $\square$

8 Impact of Turing’s 1936 Paper

8.1 The nature of proofs

One important thing to note about Turing’s paper is that he attempts to relate the intuitive notion of computability to the formal definition. This is especially evident on page 250, where he relates Turing machines to the human mind. He worked to build Turing machines from our intuitive understanding of computability. More specifically, Turing compared the tape to mathematics done by children in elementary schools. He also talked about how we naturally do one operation at a time, and that that is what his machines do. He writes, “The behavior of the computer at any moment is determined by the symbols which he is observing . . . if he wishes to observe more, he must use successive states” (250). Here is a fluid comparison between computers and the human mind. Again, this idea of forming formal definitions from intuitive ideas was very new and different. The concept of defining formal definitions using intuition as it pertains to Turing’s paper will be explored more in the following sections.

8.2 Gödel’s Reaction to Turing’s Paper

While Gödel supported Turing’s definition of computable, as well as his solution to the Decision Problem, he did not believe in one thing that Turing wrote. This was Turing’s description of the states of a machine versus the states of a human mind. Turing writes, "we will also suppose that the number of states of mind which need be taken into account is finite" (250). Turing is saying that the number of configurations that a machine has needs to be finite. Gödel takes issue with the idea that a human mind does not have infinitely many states since Turing has compared his machines to the brain throughout his paper. However, this is not Turing's intent. Yet it did start a philosophical discussion about the differences between machines and humans, which continues today.

Copeland writes, "Gödel's 1931 incompleteness results triggered, on one hand, precise definitions of effective computability and its allied notions, and on the other, some much-criticized arguments for the conclusion that the mathematical power of the mind exceeds the limitations of Turing machine" (Copeland 1). In other words, the early work done in computability theory did not just have impact on the mathematical community, but on the philosophical community as well. Gödel's issue with Turing's ideas about configurations does call into question why Gödel was so supportive of Turing's definition of computability. Copeland writes, "So how could Gödel embrace Turing's analysis of computability despite finding it to involve a fallacy, namely the imposition of a boundedness restriction on the number of states of mind? Unfortunately, Gödel says very little about his reasons for endorsing Turing's analysis, and any answer to this question is necessarily speculative. Gödel . . . was thinking of a computation procedure as a finite procedure, and at no point did he imply that this reflects, or is justified in terms of, limitations in human cognition" (Copeland 17). Additionally, Gödel says that no single machine can be equivalent to a human mind; this is known as his sharp result.

Turing, however, disagrees. Turing says that a machine is very much like a human mind. Copeland goes on to support Turing's thoughts by saying, "If one allows that, from time to time, the mind is identical to different Turing machines, allowing different sets of proofs, then there is no inconsistency between the sharp result and the claim that every stage of a dynamically changing mind (or of a dynamically changing collection of a number of minds) is a Turing machine" (Copeland 24). In other words, while Turing insists that there must be a finite number of $m-$ configurations for one machine, there are an infinite number of Turing machines. Hence, we can change between machines. Therefore, we are saying that the mind is finite but that it is infinite in that it can preform infinitely many tasks. More specifically, we could assign a Turing machine to preform one specific task that has a finite number of instructions. Then, because there are infinitely many Turing machines, we can preform infinitely many tasks by switching to different machines. Thus, there is no inconsistency between the sharp result and Turing's ideas on the human mind.

Furthermore, "in a lecture given circa 1951, Turing made it clear that his then-radical idea that machines can learn is the crux of his reply to the Mathematical Objection; and he stressed the importance of the ideal of learning new methods of proof in his 1947 discussion of the objection, describing the mathematician as searching around and finding new methods of proof" (Copeland 25). In other words, Turing thought that machines could be taught to think like humans, to use different approaches to different problems, and move between states of mind. The premise that machines can be taught to think on their own is the core to artificial intelligence. In essence, Turing designed theoretical, and then practical computers, and started the field of artificial intelligence. Remember that these mathematicians had yet to see a computer in real life, and the designs were still years away.

In summary:

> Gödel's critical note on Turing is of interest, not because Turing committed a philosophical error about the mind-machine issue, for he did not, but because of the light that the critical note helps to shed on the similarities, and the differences, in the views of these two great founders of the study of computability. Gödel praised Turing's analysis of computability but objected to the thought that Turing's restrictive conditions are grounded in facts about human cognition, and he misinterpreted some of Turing's general statements about the analysis of computability as claims about the mind in general. In fact, Turing agreed with Gödel that the mind is more powerful than any given Turing machine. Unlike Gödel, however, Turing did not think that the mind is something different from machinery.

(Copeland 27-28)

8.3 The nature of his proof that the Entscheidungsproblem is Unsolvable

For a very long time, many people did not take Turing's paper as a formal proof that the Entscheidungsproblem is unsolvable. This is interesting because we are forever trying to justify if something counts as a formal proof or not. Shapiro writes, "At the time, and for at least the next half-century, there was a near consensus that [the Church-Turing Thesis] is not subject to mathematical proof or refutation. That is, almost everyone held that mathematicians did not and could not attempt to derive [the Church-Turing Thesis] or its negation from accepted or previously established principles of mathematical domains" (Shapiro 153). In other words, the Church-Turing thesis was developed from our intuitive definition of computability. It is very hard, and often impossible, to design a formal proof for something that we built from intuition. Turing's proof of the Church-Turing Thesis, while it makes sense, is by no means formal or rigorous. Shapiro writes, "Church's and Kleene's point may be that computability is an intuitive notion. Sometimes the word 'pre-theoretic' is used. How can one prove that an intuitive, pre-theoretic notion is coextensive with one that enjoys a rigorous or otherwise formal definition? In general, how do intuitive, mathematical notions relate to their rigorously defined successors?" (Shapiro 154). This echoes how one may feel about computability theory. Some things that we consider to be right have not been formally proven, or were proven after we used informal definitions for hundreds of years. The most famous example is Fermat's Theorem. Further examples are illustrated in geometry. We have things like undefined terms, because they are held to be so basic and intuitive to not need a formal definition. For example, we do not define "point." We also have Euclid's Five Axioms which are the basis we build geometry from, because they are reasonably intuitive and understandable. We use these axioms even though they are impossible to prove, because they appear to be the correct axioms to describe our universe.

So is Turing's paper a proof? It is certainly intended that we take it as such. Shapiro writes, "If Turing's (1936) paper does constitute a proof, or even a sketch or a gem of a proof, then why did so many people fail to see it that way? Aren't proofs supposed to be compelling for any rational agent that understands the language?" (Shapiro 157). In math, we are forever trying to set up laws to govern our system of logic. Deciding what constitutes a proof is a big part of this process. While Turing's proof is less than formal, it is important that the attempt appeals to our intuition and logic.

9 Alan Turing's Life and Impact

9.1 Life

Turing's family motto was Fortuna audentes Juvat: fortune favors the brave (Sara Turing 3). His mother met his father, Julius Turing, in 1907. Alan Mathison Turing was their second son and was born in 1908. His mother writes, "a very clever child, I should say, with a wonderful memory for new words" (Sara Turing, 9) when Alan was three years old. She further writes, "even at age five he had an answer to one's arguments which could not be gainsaid - a foretaste to his powers in argument in later years" (Sara Turing 11). In short, Alan Turing was a clever and happy child who was very curious. He became known for his shortcuts in his work, such as simply knowing the answer while others had to do a lot of work to get it (Sara Turing, 12-13). In 1938, he writes, "Mathematical reasoning may be regarded rather schematically as the exercise of a combination of two faculties, which we may call *intuition* and *ingenuity*" in a paper to the London Mathematical Society (Sara Turing 13). This was clearly in evidence to his 1936 paper where he made the leap in designing Turing machines, and then showing all of their capabilities. He was forever experimenting with clay and plants, making up mixtures when he was a child (Sara Turing 13-15). His love of the outdoors stayed with him and he took up running. When he went off to college, boarding school for Americans, he biked the sixty miles to the school by himself in 1926 over two days. He was fourteen years old.

In 1927, when he was fifteen, he discovered the series for $tan^{-1}x$, not knowing that it was previously known (Sara Turing 26). When asked how he got his answer, he said "Well, it's right, isn't it?" (Sara Turing 26). This was very common, that he knew the answer simply because he knew it, and that he did not have well thought out work for it, much to the annoyance of his teachers. He wrote explanations on complicated topics for his mother, distilling them so that she could understand the very complicated, doing so ever since he was a young child (30). While Turing's school did not necessarily approve of his overwhelming amount of curiosity, it is clear that his mother remained a strident supporter and confidant to him. Sara writes, "the school reports remind me of the incoming tide" (30). She goes on to say, "in 1928 the time came round for the taking of the School Leaving Certificate. There was considerable tension in the common room between the literary and the scientific members of staff" (31) since Alan had weakness in the written word and his methods for solving problems were under question. So while he was considerably brilliant, his teachers did not always think so because they could not follow his logic. Sara also writes, "at home we regarded Alan as the family encyclopedia; he seemed to have the answers to all our scientific queries. He was always most amusing; the most ordinary acts, such as going into town, somehow, in his case, were fraught with adventure or unusual occurrences, so that he came in for a good deal of chaff, from me especially, but it was always taken in good part. For all his brilliance and his achievements he never posed as 'high-brow'" (39). He never became boastful, especially with others. Specifically, when "he collaborated with others he was most insistent on the credit due to them, while minimising his own share. He always lamented his own slowness, as he deemed it, yet many others were amazed at the rapidity which he saw the solutions to their problems" (Sara Turing 45).

During his childhood, his best friend, Christopher Morcom died in 1930. This forever changed Alan Turing.

He writes in 1930, “I feel that I shall meet Morcom again somewhere and that there will be some work for us to do together as I believed there was for us to do here. Now that I am left to do it alone I must not let him down, but put much energy into it, if not as much interest, as if he were still here. If I succeed I shall be more fit to enjoy his company than I am now” (35). He went on to win the Christopher Morcom Prize for Natural Science in 1931.

He went to King’s College in Cambridge to study mathematics in 1931. He regularly impressed his professors to the extent that they remembered him some thirty years later. He was heavily involved in peace movements and did not tolerant Nazi sympathies. One former classmate writes, “so far as I can remember he had no particular political affiliations, but he recognized Nazi doctrine as an evil and was not prepared to compromise with it by the smallest gesture . . . there was nothing boorish about him, but he had no use for any sort of pretence” (Sara Turing 42). It is no great surprise then that he would become involved in the war effort. He graduated in 1934 and was elected as a Fellow to John’s College at 22 years old.

During World War II, Turing put his ideas into action to break nazis codes. During the height of the war, “the cryptanalysts at Bletchley Park were decoding about 39,000 intercepted messages each month, a figure that rose subsequently to more than 84,000 per month—two messages every minute, day and night” (Copeland 2019). This accomplishment was made possible by Turing’s earlier work and his work in cracking Enigma. Their work ended the war two years early and saved roughly over 14 million people (Göke).

After the war was over, Turing was recruited to create one of the first electronic computer for the National Physical Laboratory (Copeland 2019). If it had been built, “it would have had vastly more memory than any of the other early computers, as well as being faster. However, his colleagues at NPL thought the engineering too difficult to attempt, and a much smaller machine was built, the Pilot Model ACE (1950)” (Copeland 2019). While his team did not build the first electronic computer, his work with Turing machines greatly influenced the first electronic computer. Turing went on to write the first programming guide (Copeland 2019).

In March 1951, he was convicted of gross indecency because he was homosexual. This occurred because his ex-boyfriend had broken into his home. When he went to the police about the crime, they convicted him for being gay. He was sentenced to a year of estrogen injections. Because of his crime he lost his security clearance and effectively was unable to continue his work (Göke). Suffering from depression he committed suicide by ingesting a cyanide laced apple.

9.2 Impact

Let us begin with Turing’s accomplishments during his 1936 paper. Copeland writes, “It was in the course of his work on the Entscheidungsproblem that Turing invented the universal Turing machine, an abstract computing machine that encapsulates the fundamental logical principles of the digital computer” (Copeland 2019). The universal Turing machine, which we talked about earlier, laid the foundation for the computers that followed. Copeland further writes, “An important step in Turing’s argument about the Entscheidungsproblem was the claim, now called the Church-Turing thesis, that everything humanly computable can also be computed by the universal Turing machine. The claim is important because it marks out the limits of human computation”

(Copeland 2019). We have still not found anything that a human or mind could compute that a Turing machine could not. It is important to note that for all we have done with computers in the some eighty years since Turing designed his Turing machines, we have not found anything that cannot be simulated by a Turing machine. This includes quantum and super computers. It remains today an impressive feat.

His work directly led to the creation of the first electronic computer. He further improved that accomplishment by designing ways for the computer to hold more memory and be faster. Turing's ideas on artificial intelligence have led to some of the greatest philosophical discussions of our time. Additionally, he profoundly changed the outcome of World War II. He saved fourteen million lives and ended the war two years early (Göke).

For all of this, it is important that we remember Alan Turing.

10 Bibliography:

Copeland, B.J. “Alan Turing.” Encyclopædia Britannica, Encyclopædia Britannica, Inc., 19 June 2019, www.britannica.com/b
Turing.

Copeland, B. Jack, et al. Computability: Turing, Gödel, Church, and Beyond. The MIT Press, 2015, ProQuest Ebook Central, ebookcentral-proquest-com.gcsu.idm.oclc.org/lib/gcsu/home.action.

Göke, Niklas. “The Man Who Changed History Twice in a Single Moment.” *Medium*, Personal Growth, 31 Oct. 2019, medium.com/personal-growth/alan-turing-how-to-change-history-twice-in-a-single-moment-5cfdb47b8e8d.

Rojas, Raúl. “A Tutorial Introduction to the Lambda Calculus.” ArXiv abs/1503.09060 (2015): n. pag.

Soare, Robert Irving. *Turing Computability: Theory and Applications.* Springer, 2016.

Turing, A.M. *On Computable Numbers with an Application to the Entscheidungsproblem*, Proc. London Math. Soc. (2) **42** (1936), 230-265, reprinted in [Davis 1965], pp. 115-153.
Turing, Sara, et al. *Alan M. Turing.* Cambridge University Press, 2015.

Weber, Rebecca. *Computability Theory.* American Mathematical Society, 2012.

Weisstein, Eric W. “Ackermann Function.” From MathWorld–A Wolfram Web Resource.
http://mathworld.wolfram.com/AckermannFunction.html