Computational problems and solutions

Millan Philipose

Math 336, Spring 2021

1 Abstract

Beginning courses in mathematics pose questions such as, "what is the product of 123 and 1234?" and "what is the greatest common divisor of 10 and 70?" Though almost everyone has encountered these questions in school, it is surprisingly difficult to define what it means to "find the solution." A good definition would capture the intuitive meaning of the term "solution." For example, we would like to distinguish between the unsatisfactory answer $123 * 1234$ and the satisfactory answer 151782. A good definition must also honor the meaning of the word "find" – the answer should be obtained through a series of methodical steps rather than being produced out of thin air. The theory of computing gives us a rigorous definition, grounded in set theory, which meets both criteria.

This paper begins with a precise definition of a computational problem. We then introduce the concept of a *Turing machine* and define what it means to solve a problem. After providing a few examples of computational solutions, we show that some problems have no solution at all – first by showing that the set of problems is larger than the set of solutions, then by examining the famously unsolvable *halting problem.* We then investigate the structure of the unsolvable problems. We define a partial ordering relation known as *Turing reducibility* which indicates whether one problem is "more unsolvable" than another. Finally, we define an operator which allows us to generate "arbitrarily difficult" problems.

2 Defining computation

In mathematics, we use the term "problem" to describe results we wish to prove or examples we wish to find. In the theory of computing, however, the concept of a problem is narrow and rigorous.

2.1 What is a problem?

Definition 2.1.1. A *problem* is either a function from S to $\mathbf{N}$, where $S \subseteq \mathbf{N}$, a subset of the natural numbers, or an indicator function $f : \mathbf{N} \to \{0, 1\}$ for some set of natural numbers.

Remark 2.1.2. Every function $f : S \to \mathbf{N}, S \subseteq \mathbf{N}$ may be identified with its graph, which is a subset of $\mathbf{N} \times \mathbf{N}$. But it is known that the set $\mathbf{N} \times \mathbf{N}$ is of equal cardinality to the set of natural numbers. Thus the set of functions $f : S \to \mathbf{N}$ can be placed in a one-to-one correspondence with the power set of $\mathbf{N}$, which means that the first two forms of Definition 2.1.1 are essentially equivalent. In fact, the subsets of $\mathbf{N}$ can be put into a one-to-one correspondence with their indicator functions, so all three forms of the definition are equivalent.

We will mostly use the first definition, but at the end of the paper we will switch to the third.

This definition gives every problem two important properties. First, problems are mathematical objects. We can use any mathematical method to define a problem, as long as we construct a well-defined function. The problem does not need to contain instructions to find the image $f(n)$. Second, problems exclusively involve natural numbers. This restriction is fundamental to the theory of computing. We will revisit this fact as we define the concept of a *solution* to a problem.

Example 2.1.3 (Multiplication by two)**.** Let $f(n) = 2n$. Since $f : \mathbf{N} \to \mathbf{N}$ is a well-defined function, the function f is considered to be a problem.

Example 2.1.4 (Primality)**.** Let $f(n) = 1$ if n is prime, and let $f(n) = 0$ if n is not prime. Since every natural number is either prime or not prime (never both), $f : \mathbf{N} \to \mathbf{N}$ is a well-defined function, so the function f is considered to be a problem.

Solving a problem f means finding a way to "compute" $f(n)$ for any given number n. At first blush, this seems to be easier for the multiplication by two problem and harder for the primality problem. But what exactly does "compute" mean in this context, what does it mean for one problem to be "easier to solve" than another, and are there any problems which are impossible to solve? These questions will be answered in the upcoming sections.

2.2 Solving problems with Turing machines

To pin down what it means to solve a problem, we must first define computation. A computation is a series of small manipulations applied to an input to produce an output. This vague idea was formalized by Alan Turing in the concept of a Turing machine.

A Turing machine can be thought of as a device placed above a *tape* of characters. The machine keeps track of its internal *state* and is able to look at the single character on the tape positioned directly below. Based on this particular state-character combination, the machine overwrites the character directly beneath it, updates its internal state, and moves itself one position to the right or left along the tape [1]. Astonishingly, every computer invented since the time of Turing can be described by this rudimentary mathematical model.

Definition 2.2.1. [1] A *Turing machine* is a tuple in the form $(\Gamma, Q, q_0, q_a, q_r, \delta)$, where Γ and Q are finite sets of integers, $q_0, q_a, q_r \in Q$, and δ is a function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$. We call Γ the *alphabet* of the Turing machine, and we distinguish one element of Γ as the "blank character." We say that Q is the set of *states*, q_0 the *start state*, q_a the *accept state*, and q_r the *reject state*. We call δ the *transition function* of the Turing machine. We place the further restriction that, if $q \in \{q_a, q_r\}$, then for all $x \in \Gamma$ we must have $\delta(q, x) = (q, x, a)$ for some a.

Remark 2.2.2. It is absolutely critical that Γ and Q are finite sets. Without this condition, one could embed any problem directly into the transition function, and the Turing machine would become pointless. The purpose of a Turing machine, and of computation in general, is to decompose functions defined on infinite sets into simpler functions with finite domains.

To complete the formal definition, we now define what it means to "run" a Turing machine on a tape.

Definition 2.2.3. A *tape* is a mapping from the integers to some alphabet Γ.

Remark 2.2.4. Tapes extend infinitely in two directions and are capable of encoding an arbitrary amount of information. However, a tape being processed by a Turing machine can only include a finite number of non-blank characters. There will always be an infinite number of leading and trailing blanks.

Definition 2.2.5 (Running a Turing machine once)**.** Let $X = (\Gamma, Q, q_0, q_a, q_r, \delta)$ be a Turing machine, let T be a tape with alphabet Γ, let $n \in \mathbf{Z}$ be a position on the tape, and let $q \in Q$ be a state of the Turing machine X. Suppose q', c', n' are given by

$$(q', c', n') = \delta(q, T(n)) \in Q \times \Gamma \times \{-1, 1\}.$$

Then we create an updated tape T' defined by $T'(n) = c'$ and $T'(z) = T(z)$ for all $z \neq n$. Finally, we define the function $\mathbf{r}(X, T, n, q)$ as follows:

$$\mathbf{r}(X, T, n, q) = (X, T', n + n', q').$$

The object $\mathbf{r}(X, T, n, q)$ is the result of running the Turing machine X for one iteration on tape T at position n with state q.

Now that we know what it means to run a Turing machine once, we can define what it means to allow a Turing machine to run through multiple iterations.

Definition 2.2.6 (Running a Turing machine k times)**.** Let X be a Turing machine with $X = (\Gamma, Q, q_0, q_a, q_r, \delta)$, let T be a tape with alphabet Γ, and let $n \in \mathbf{Z}$ be a position on the tape. We define a family of functions $\mathbf{r}_k$ recursively, as follows:

$$\mathbf{r}_0(X, T, n) = (X, T, n, q_0)$$

$$\mathbf{r}_{k+1}(X, T, n) = \mathbf{r}(\mathbf{r}_k(X, T, n)).$$

We say that $\mathbf{r}_k(X, T, n)$ is the result of running X for k iterations starting at position n on tape T.

Suppose that a Turing machine is allowed to run indefinitely in this manner. A natural question is whether the machine *halts*; that is, whether the machine ever achieves one of its end-states q_a or q_r.

Definition 2.2.7 (Running a Turing machine to completion)**.** Let X be a Turing machine with accept and reject states q_a and q_r and let T be a tape with the same alphabet as X. By convention we assume that X is initially located at position 0 on the tape. If there exists a natural number k such that

$$\mathbf{r}_k(X, T, 0) = (X, T', n, q_r)$$

or

$$\mathbf{r}_k(X, T, 0) = (X, T', n, q_a)$$

for some T', n, then we say that the Turing machine X *halts on the input* T. In the first case we say that M *enters the reject state*, and in the second case we say that X *enters the accept state*. In either case, we say that this T' is the *output of* X *on* T, and we write $X(T) = T'$. On the other hand, if there exists no such k, then we say that X *fails to halt* on the input T.

Remark 2.2.8. It is easy to show that a Turing machine cannot enter both the accept and the reject states. It is also easy to show that the output $X(T)$ is well-defined. Both follow from the fact that the transition function δ, when given one of the states q_a or q_r, always returns the exactly the state and character it was given. Thus, once the Turing machine enters either the accept or reject states, its state never changes and the tape is never modified.

We can now define what it means to solve a problem. The key step is finding a way to identify natural numbers with tapes.

Definition 2.2.9 (Mapping numbers to tapes)**.** If a natural number x can be represented in binary by the digits $a_0a_1a_2...a_k$, we can take the alphabet $\{0, 1\} \cup B$ (where B is the blank character) and define the tape $T(x)$ by $T(x)(n) = a_n$ for $0 \leq n \leq k$ and $T(x)(n) = B$ for every other n. Conversely, given a tape T with alphabet $\{0, 1\} \cup B$, one can strip away all of the blanks to obtain a sequence of binary digits, which in turn corresponds to a natural number $n(T)$.

This correspondence between tapes and numbers allows us to associate Turing machines with problems.

Definition 2.2.10 (Solutions and decidability)**.** We say that the Turing machine M *computes* the function f, and that M is a *solution* to the problem f, if the Turing machine M halts and gives $n(M(T(x))) = f(x)$ for all x in the domain of f. We say that a problem f is *decidable* or *computable* if it has some solution M. We say that f is *undecidable* if it does not have any solution.

Remark 2.2.11. The domain of f must be countable, because only countable sets can be mapped into finite strings on a tape. If the domain of f were uncountable, it would be impossible to build a function $T(x)$ which maps the inputs of f to distinct tapes. This is why the domain of f must be a subset of the natural numbers.

It is usually quite laborious to construct a Turing machine. To illustrate how a Turing machine works, here is a simple example.

Example 2.2.12 (Multiplication by two)**.** Let f be the problem $f(x) = 2x$. Let our alphabet be $\Gamma = \{0, 1, B\}$, where $B = -1$ is used as the blank character. Define the set of states $Q = \{1, 2, 3\}$ where $q_0 = 1, q_a = 2, q_r = 3$ (the particular numbers do not matter), and define the transition function δ as follows:

- If $\gamma \neq B$, then $\delta(q, \gamma) = (q, \gamma, 1)$.
- If $\gamma = B$, then $\delta(q, \gamma) = (q_a, 0, 1)$.

Now consider the Turing machine $M = (\Gamma, Q, q_0, q_a, q_r, \delta)$ as it runs to completion on the tape $T(x)$ (see Definition 2.2.9) for some natural number x. After starting at position 0 on the tape, the Turing machine follows the rule $\delta(q, \gamma) = (q, \gamma, 1)$ and moves to the right without altering its internal state or the data on the tape. It repeats this process until it reaches a blank character – the first of which occurs immediately after the final digit of the binary number. Once the machine reaches this blank character, it replaces the blank with a 0 and halts. Adding a zero to the end of a binary number is the same as multiplying that number by two. Hence we have $n(M(T(x))) = 2x$ for any natural number x, so we conclude that the Turing machine T is a solution to the multiplication-by-two problem. Therefore the multiplication-by-two problem is decidable.

3 Limits of computation

Not all problems have a solution. In this section we will use set theory to prove the existence of an undecidable problem, and we will develop an important example known as the halting problem.

3.1 Existence of undecidable problems

Surprisingly, the most natural way to show the existence of an undecidable problem is through a non-constructive proof. Here we will show that the cardinality of the set of problems is strictly greater than the cardinality of the set of Turing machines.

Lemma 3.1.1. *The set of problems is uncountable.*

Proof. By Remark 2.1.2, it suffices to show that the power set $2^{\mathbf{N}}$ is uncountable.

We first show that $2^{\mathbf{N}}$ is not countably infinite. Suppose for the sake of contradiction that there exists a bijection $f : \mathbf{N} \to 2^{\mathbf{N}}$. Consider the set

$$S = \{x \in \mathbf{N} : x \notin f(x)\}.$$

Since f is a bijection and $S \in 2^{\mathbf{N}}$, there must exist some natural number a such that $f(a) = S$. Now we divide into two cases. If $a \in S$, then the definition of

S tells us that $a \notin f(a) = S$, so therefore we have both $a \in S$ and $a \notin S$ which is a contradiction. On the other hand, if $a \notin S$, then we have $a \notin f(a)$, so by definition of S we must have $a \in S$, which is another contradiction. All cases lead to a contradiction, so there cannot exist any bijection between the natural numbers and the power set $2^{\mathbf{N}}$ and therefore $2^{\mathbf{N}}$ is not countable.

Now consider the set of singletons $X = \{\{0\}, \{1\}, \{2\}, ...\}$. Clearly $X \subseteq 2^{\mathbf{N}}$, and clearly the function $f(n) = \{n\}$ is a bijection between the natural numbers and X. Thus the set $2^{\mathbf{N}}$ has an infinite subset, so the set $2^{\mathbf{N}}$ itself must be infinite.

Since the set $2^{\mathbf{N}}$ is infinite and not countable, so must be the set of problems. In other words, the set of problems is uncountable. □

Lemma 3.1.2. *If X is a countable set, then the set of finite subsets of X is at most countable.*

Proof. Let S be the set of finite subsets of X. Let $Z_n = \{x \in 2^X : \#(x) = n\}$; that is, let Z_n be the set of all subsets of X which contain exactly n elements. Clearly we have $S = \cup_{n=0}^{\infty} Z_n$. Also, for every set $z \in Z_n$, there exists some ordered n-tuple $(z_k)_{k=1}^n$ which contains every element in the set z. Thus we can use the Axiom of Choice to define a function $f : Z_n \to X^n$ by $f(z) = (z_k)_{k=1}^n$. Since each tuple $(z_k)_{k=1}^n$ contains precisely the elements of z, we know that the function f is injective. But since the finite Cartesian product of countable sets is countable, the set X^n must be countable, so the existence of the injection $f : Z_n \to X^n$ tells us that the set Z_n is at most countable. Finally, since the countable union of at most countable sets is at most countable and $S = \cup_{n=0}^{\infty} Z_n$, the claim follows. □

Lemma 3.1.3. *The set of Turing machines is at most countable.*

Proof. Recall that every Turing machine is a tuple in the form $(\Gamma, Q, q_0, q_a, q_r, \delta)$, where Γ and Q are finite sets of natural numbers, q_0, q_a, q_r are natural numbers, and δ is a function from $Q \times \Gamma$ to $Q \times \Gamma \times \{-1, 1\}$. We may consider the set of Turing machines as a subset of $A_\Gamma \times A_Q \times A_{q_0} \times A_{q_a} \times A_{q_r} \times A_\delta$, where A_Γ is the set of all possible values of Γ and so on. To prove the claim, it suffices to show that each set A_Γ, A_Q... is at most countable.

Since Γ and Q are finite sets of natural numbers, Lemma 3.1.2 tells us A_Γ and A_Q are at most countable. Since q_0, q_a, q_r are all natural numbers, we have $A_{q_0}, A_{q_a}, A_{q_r} \subseteq \mathbf{N}$, meaning that each of $A_{q_0}, A_{q_a}, A_{q_r}$ is at most countable. To determine the size of A_δ, we identify each function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$ with its graph

$$\{(x, y) \in (Q \times \Gamma) \times (Q \times \Gamma \times \{-1, 1\}) : \delta(x) = y\}.$$

For any valid transition function δ, we have $\delta \subseteq (Q \times \Gamma) \times (Q \times \Gamma \times \{-1, 1\})$. Since the latter set is finite, so must be the graph of δ. Therefore, since each of Q, Γ is a subset of the natural numbers, each δ may be considered as a finite subset of the countable set $(\mathbf{N} \times \mathbf{N}) \times (\mathbf{N} \times \mathbf{N} \times \{-1, 1\})$. But this means, by

Lemma 3.1.2, that the set of all possible values of δ is at most countable. Thus A_δ is also at most countable.

Since each of $A_\Gamma, A_Q, A_{q_0}, A_{q_a}, A_{q_r}, A_\delta$ is at most countable, so must be the Cartesian product $A_\Gamma \times A_Q \times A_{q_0} \times A_{q_a} \times A_{q_r} \times A_\delta$. Therefore, by our reasoning in the first paragraph, the claim follows. □

Lemma 3.1.4. *No Turing machine solves two distinct problems.*

Proof. Suppose that the Turing machine M solves the distinct problems f and g. Then, by Definition 2.2.10, we have $n(M(T(x))) = f(x)$ and $n(M(T(x))) = g(x)$ for every natural number x. But this means that $f(x) = g(x)$ for all $x \in N$, which in turn implies that $f = g$, a contradiction. □

Theorem 3.1.5. *There exists an undecidable problem.*

Proof. Suppose for contradiction that every problem has a solution. Then there must exist a function s which maps every problem to a Turing machine which is its solution. By Lemma 3.14, this function s is injective. Since there is an injective mapping from the set of problems into the set of Turing machines, the set of Turing machines must have greater cardinality than the set of problems. But this is impossible, since the set of Turing machines is at most countable (Lemma 3.1.3) while the set of problems is uncountable (Lemma 3.1.1). Thus there must exist a problem which does not have a solution. □

3.2 The halting problem

Having verified the existence of an undecidable problem through a non-constructive proof, we now construct a famous and useful example. We will prove that no Turing machine is capable of deciding whether another Turing machine halts on a given input.

Definition 3.2.1 (Enumeration of Turing machines)**.** [1] Since the set of Turing machines is at most countable (Lemma 3.1.3), we can identify each Turing machine M with a distinct natural number $\langle M\rangle$. We say that $\langle M\rangle$ is the *natural number representation* of the Turing machine M.

Definition 3.2.2 (Halting problem)**.** [1] Let f_{halt} be defined by

$$f_{\text{halt}}(\langle M\rangle, x) = \begin{cases} 1 & \text{if } M(x) \text{ halts with output } 1 \\ 0 & \text{otherwise.} \end{cases}$$

The function f_{halt} does not strictly match the definition of a problem, as it takes pairs of natural numbers as inputs. To be fully formal, we can choose any bijection $g : \mathbf{N} \to \mathbf{N} \times \mathbf{N}$ and define a new problem function $f^+_{\text{halt}}(n) = f_{\text{halt}}(g(n))$. Here we will choose a special bijection g which works well in a subsequent proof.

Lemma 3.2.3. *There exists a bijection* $g : \mathbf{N} \to \mathbf{N} \times \mathbf{N}$ *such that* $g(2n) = (n, n)$ *for each natural number* n*.*

Proof. First we notice that the set of pairs $X = \{(a,b) \in \mathbf{N}^2 : a \neq b\}$ is an infinite subset of the countable set $\mathbf{N} \times \mathbf{N}$, so the set X is countable. Second we notice that the set of odd natural numbers O is countable. Taking these facts together, we can choose a bijection $h : O \to X$ and define the function g explicitly by $g(n) = (n/2, n/2)$ for n even and $g(n) = h(n)$ for n odd. From here it is easy to verify that $g : \mathbf{N} \to \mathbf{N} \times \mathbf{N}$ is indeed a bijection. $\square$

Theorem 3.2.4. *[1] The halting problem is undecidable.*

Proof. Suppose for contradiction that there exists a Turing machine which computes f_{halt}. That is, defining the bijection g as in the previous Lemma, suppose there exists a Turing machine M_{halt} which solves the problem $f^+_{\text{halt}}(n) := f_{\text{halt}}(g(n))$.

Using M_{halt}, we can construct a new Turing machine N^* which inputs the natural number representation of a Turing machine, applies the algorithm in Example 2.2.12 to double that number, and then applies the algorithm of the machine M_{halt}. Notice that by this procedure, the computation $N^*(\langle M \rangle)$ yields the value given by

$$N^*(\langle M \rangle) = M_{\text{halt}}(2\langle M \rangle) = f^+_{\text{halt}}(2\langle M \rangle) = f_{\text{halt}}(g(2\langle M \rangle)) = f_{\text{halt}}(\langle M \rangle, \langle M \rangle).$$

In other words, the machine N^* determines whether or not a given Turing machine accepts its own natural number representation as an input.

Given the machine N^*, we can easily construct another machine M^* which gives $M^*(\langle M \rangle) = 1$ if $N^*(\langle M \rangle) = 0$ and gives $M^*(\langle M \rangle) = 0$ if $N^*(\langle M \rangle) = 1$.

Now consider the value given by $M^*(\langle M^* \rangle)$. Since M_{halt} and N^* halt on any input, so must M^*, so in particular M^* must halt on the input $\langle M^* \rangle$. From the definition of M^*, it is easy to see that $M^*(\langle M^* \rangle)$ is either zero or one.

We now divide into cases. If $M^*(\langle M^* \rangle) = 1$, then $f_{\text{halt}}(\langle M^* \rangle, \langle M^* \rangle) = 0$, so by the definition of the halting problem, the machine M^* does not halt with output 1 on the input $\langle M^* \rangle$. This directly contradicts our assumption that $M^*(\langle M^* \rangle) = 1$. On the other hand, if $M^*(\langle M^* \rangle) = 0$, then we have $f_{\text{halt}}(\langle M^* \rangle, \langle M^* \rangle) = 1$. Therefore, by the definition of the halting problem, the machine M^* halts with output 1 on the input $\langle M^* \rangle$. But this contradicts the assumption that $M^*(\langle M^* \rangle) = 0$. Therefore, a Turing machine M_{halt} solving the halting problem cannot exist. $\square$

4 Ordering problems by difficulty

Thus far we have treated decidability as a property which is either true or false. But just as certain infinities are "larger" than others, certain problems are "more undecidable" than others. In this section we will give structure to the set of problems by introducing the concept of Turing reducibility. We will show that Turing reducibility is a partial order on the set of problems, and we will consider some important examples and properties of this ordering relation.

4.1 Oracle machines and Turing reducibility

A problem f is *Turing reducible* to a problem g if there is a way to compute f while having access to an ready-made solution to g. To define what it means for one solution to have access to another, we must slightly extend the concept of a Turing machine and introduce the concept of an *oracle*.

Definition 4.1.1. [2] Fix $g(n)$ as any bijection $g : \mathbf{N} \to \mathbf{N} \times \mathbf{N}$. Viewing problems as functions $f : S \to \mathbf{N}, S \subseteq \mathbf{N}$, we say that a tape T is the *oracle tape* for a problem f if we have $g(n) \in \text{graph } f \implies T(n) = 1$ and $g(n) \notin \text{graph } f \implies T(n) = 0$. Viewing problems as indicator functions $f : \mathbf{N} \to \{0, 1\}$, we say that a tape T is the *oracle tape* for f if we have $g(n) = f(n)$ for all $n \in \mathbf{N}$.

Remark 4.1.2. Oracle tapes are like cheat sheets that allow a Turing machine to look up answers to problems. They are allowed to contain an infinite number of non-blank characters, because it sometimes takes an infinite amount of data to fully encode a problem. This distinguishes oracle tapes from ordinary tapes, which are only allowed to include a finite amount of data. We will now refer to the tapes used by ordinary Turing machine as *work tapes*.

Definition 4.1.3. [2] An *oracle machine* is a tuple in the form $(\Gamma, Q, q_0, q_a, q_r, \delta)$, where δ is a function from $Q \times \Gamma \times \{B, 0, 1\}$ to $Q \times \Gamma \times \{-1, 1\}$ and everything else conforms to the definition of a Turing machine.

Remark 4.1.4. The transition function has access to the machine's current state, the character at the current position on the work tape, and the character at the current position on the oracle tape. It outputs the new state, the new character to write onto the current position in the work tape, and the direction to move on both tapes. The key difference from is that an oracle machine's transition function can read off of an infinite oracle tape, while a Turing machine's transition function cannot.

Oracle machines run nearly identically to Turing machines. To illustrate the similarity, we will create an analogue of Definition 2.25 for oracle machines.

Definition 4.1.5 (Running an oracle machine once)**.** Let $X = (\Gamma, Q, q_0, q_a, q_r, \delta)$ be an oracle machine, let T be a tape with alphabet Γ, let T_{oracle} be an oracle tape, let $n \in \mathbf{Z}$ be a position on the tape, and let $q \in Q$ be a state of the oracle machine X. Suppose q', c', n' are given by $(q', c', n') = \delta(q, T(n), T_{\text{oracle}}(n))$. Then we create an updated tape T' defined by $T'(n) = c'$ and $T'(z) = T(z)$ for all $z \neq n$. Finally, we define the function $\mathbf{r}(X, T, T_{\text{oracle}}, n, q)$ as follows:

$$\mathbf{r}(X, T, T_{\text{oracle}}, n, q) = (X, T', T_{\text{oracle}}, n + n', q').$$

The tuple $\mathbf{r}(X, T, T_{\text{oracle}}, n, q)$ is the result of running the oracle machine X for one iteration on work tape T and oracle tape T_{oracle} at position n with state q.

There are similar analogues to the other definitions in §2.2. If X is an oracle machine and X halts on the input n when given access to the oracle tape for g, we denote the final state of the work tape by $X_g(n)$. If f and g are problems, and if X is an oracle machine run on the oracle tape for g, we say that the oracle machine X *computes* f *with recourse to an oracle for* g if $X_g(n) = f(n)$ for all $n \in \operatorname{dom} f$.

We are now ready to define what it means for one problem to be easier than another.

Definition 4.1.6 (Turing reducibility)**.** [3] Let f and g be problems. We say that f is *Turing reducible* to g if there exists an oracle machine X which computes f with recourse to an oracle for g. If f is Turing reducible to g, we write $f \leq_T g$. If f is Turing reducible to g but not vice versa, we write $f <_T g$.

Remark 4.1.7. Though we will not prove it formally, Turing reducibility is transitive. The idea is that if $f \leq_T g$ and $g \leq_T h$, then we can modify the oracle machine which computes f with recourse to g so that every time the machine attempts to consult the oracle tape for g, it instead runs the oracle machine for g with recourse to h. It is also the case that Turing reducibility is reflexive; that is, any problem f is Turing reducible to itself.

Intuitively, if a problem f is "easier" than a solvable problem g, then we would expect f to also be solvable.

Lemma 4.1.8. *If g is a computable problem and f is a problem which is Turing reducible to g, then f must also be computable.*

Proof. We use the transitive property of Turing reducibility. Since g is computable, there must exist a Turing machine $M = (\Gamma, Q, q_0, q_a, q_r, \delta)$ which computes g. Now consider the obviously computable "zero problem" $f_0(x) := 0$, which we will treat as an indicator function for the empty set. From M we can construct an oracle machine M^* by defining $\delta^*(q, \gamma, n) := \delta(q, \gamma)$ and writing

$$M^* = (\Gamma, Q, q_0, q_a, q_r, \delta^*).$$

The oracle machine M^* ignores the oracle tape and interacts with the work tape in exactly the same manner as M. Hence, when M^* is run with recourse to an oracle for the zero problem, the machine M^* computes the function g. Therefore we have $g \leq_T f_0$.

But combining this with the fact that $f \leq_T g$, we find that $f \leq_T f_0$. This means that there exists an oracle machine F^* which computes f with recourse to an oracle for the zero problem f_0. Let δ_F^* be the transition function of F^*, and let us define a Turing machine F which is the same as F^* except that it uses the transition function $\delta_F(q, \gamma) := \delta_F^*(q, \gamma, 0)$. Since the oracle tape consists entirely of zeros, the function δ_F will have exactly the same behavior as δ_F^*. Hence the Turing machine F will have the same behavior as the oracle machine F^* when it is run with recourse to the zero function, which means that F is a Turing machine that computes f. This means that f is decidable, as desired. □

Using the contrapositive of this lemma, we can employ Turing reduction to prove that certain problems are undecidable. Here is a sketch of a simple undecidability-by-reduction proof.

Example 4.1.9 (Reach problem is undecidable)**.** Let us define the *reach problem* as follows: let $f_{\text{reach}}(\langle M\rangle, x, Q) = 1$ if there exists a natural number k such that

$$\mathbf{r}_k(M, T(x), 0) = (X, T^*, n, q)$$

for some integer n, some tape T^* encoding the number 1, and some $q \in Q$. Let $f_{\text{reach}}(\langle M\rangle, x, Q) = 0$ otherwise.

There is a strong similarity between f_{reach} and the halting problem f_{halt}. In fact, given a Turing machine M and an input x, the halting problem merely checks whether M reaches one of the states $\{q_a, q_r\}$ when run on the input x. Thus the halting problem is merely a special case of the reach problem, so we have $f_{\text{halt}} \leq_T f_{\text{reach}}$. But since f_{reach} is decidable, then Lemma 4.1.8 tells us that the halting problem must also be decidable, contradicting Theorem 3.2.4. Therefore, f_{reach} must be an undecidable problem.

Turing reducibility allows us to differentiate between problems based on their relative difficulty, but it also allows us to group similar problems together.

Definition 4.1.10 (Turing equivalence)**.** [3] If f and g are each Turing reducible to the other, we say that f and g are *Turing equivalent* and write $f \equiv_T g$.

As indicated by the name, Turing equivalence is an equivalence relation. This follows directly from the fact that Turing reducibility is reflexive and transitive. Using this relation, we can partition the set of problems into equivalence classes known as *Turing degrees.*

Definition 4.1.11 (Turing degrees)**.** [3] A nonempty set of problems $\mathbf{a}$ is a *Turing degree* if every problem which is Turing equivalent to any problem in $\mathbf{a}$ is itself a member of $\mathbf{a}$. We use $\mathbf{0}$ to denote the set of computable problems.

The set of Turing degrees admits a natural ordering.

Definition 4.1.12 (Ordering Turing degrees)**.** [3] If $\mathbf{a}$ and $\mathbf{b}$ are Turing degrees, we say that $\mathbf{a} \leq \mathbf{b}$ if every problem in $\mathbf{a}$ is Turing reducible to every problem in $\mathbf{b}$.

Remark 4.1.13. The relation $\leq$ is a partial order on the set of Turing degrees. Reflexivity follows from the fact that every element of $\mathbf{a}$ is equivalent to every other element of $\mathbf{a}$. Transitivity follows from the fact that Turing reducibility is transitive. For antisymmetry, we observe that if Turing classes $\mathbf{a}, \mathbf{b}$ satisfy $\mathbf{a} \leq \mathbf{b}$ and $\mathbf{b} \leq \mathbf{a}$, then every problem in $\mathbf{a}$ is reducible to every problem in $\mathbf{b}$ and vice versa. Hence every problem in $\mathbf{a}$ is Turing equivalent to every problem in $\mathbf{b}$. Since $\mathbf{a}, \mathbf{b}$ are both Turing degrees, this means that $\mathbf{a} = \mathbf{b}$.

4.2 The jump operator

So far our universe of Turing degrees is quite small. We know of the class of computable problems, and we know of the class of problems which are Turing equivalent to the halting problem, but we do not know of any problems harder than the halting problem. A natural question is whether there is a maximal Turing degree; that is, whether there is a set containing the "least computable" problems in existence. It turns out that the answer is no. The jump operator allows us to take any problem and generate another which is strictly more difficult to solve.

To generate these difficult problems, we consider the halting problem on an oracle machine. We will construct a function which checks whether a given oracle machine halts on a given input when it is run with recourse to an oracle for a particular problem. For the sake of convenience, we will invoke Remark 2.1.2 and limit our discussion to problems $f : \mathbf{N} \to \{0, 1\}$ which are indicator functions of sets $S \subseteq \mathbf{N}$.

Definition 4.2.1 (Enumeration of oracle machines)**.** By a simple modification of Lemma 3.1.3 (left to the reader), there are at most countably many oracle machines. Hence, we can assign a unique natural number representation $\langle M \rangle$ to every oracle machine M.

Definition 4.2.2 (Jump operator)**.** [4] Let $f : S \to \mathbf{N}$ be a problem. We define the *Turing jump* of f to be the following problem:

$$f'(\langle M \rangle, x) = \begin{cases} 1 & \text{if } M_f(x) = 1 \\ 0 & \text{if } M_f(x) \neq 1 \text{ or if } M_f(x) \text{ fails to halt.} \end{cases}$$

We say that the transformation $f \mapsto f'$ is the *jump operator*.

Remark 4.2.3. As with the halting problem, we can turn f' into a fully formal problem f'_{new} by choosing a bijection $g : \mathbf{N} \to \mathbf{N} \times \mathbf{N}$ and defining $f'_{\text{new}} := f'(g(n))$.

Example 4.2.4. Consider the jump f'_0 of the zero problem $f_0(x) = 0$. Roughly speaking, f'_0 will check whether a given oracle machine halts on a given input when run with recourse to an oracle for the zero problem. But an oracle for the zero problem gives no information – every character on the oracle tape is 0 and therefore the oracle tape does not affect the behavior of the transition function at all. Hence, an oracle machine with recourse to an oracle for the zero problem is essentially the same as a standard Turing machine, which means that the problem f'_0 is equivalent to the traditional halting problem.

Our aim is to prove that for any problem f, we have $f <_T f'$. To do so, we must first introduce a special kind of oracle machine which directly consults the oracle tape without performing any computation.

Definition 4.2.5 (Lookup machine)**.** Let M be an oracle machine. We say that M is a *lookup machine for f* if $M_f(x) = f(x)$ for all x in the domain of f.

Remark 4.2.6. Because Turing reducibility is reflexive, every problem has a lookup machine. That is, for any problem f, there must exist an oracle machine which computes f with recourse to an oracle for f. In particular, if f is an indicator function $f : \mathbf{N} \to \{0, 1\}$, then the lookup machine for f halts on all natural number inputs.

We are now ready to prove the existence of arbitrarily difficult problems.

Lemma 4.2.7. *For any natural number m, there exists a bijection $g_m : \mathbf{N} \to \mathbf{N} \times \mathbf{N}$ such that $g_m(2n) = (m, n)$.*

Proof. First we notice that the set of pairs $X = \{(a, b) \in \mathbf{N}^2 : a \neq m\}$ is an infinite subset of the countable set $\mathbf{N} \times \mathbf{N}$, so the set X is countable. Second we notice that the set of odd natural numbers O is countable. Taking these facts together, we can choose a bijection $h : O \to X$ and define the function g explicitly by $g_m(n) = (m, n/2)$ for n even and $g_m(n) = h(n)$ for n odd. $\square$

Lemma 4.2.8. *For any problem f, we have $f \leq_T f'$.*

Proof. To show that $f \leq_T f'$, it suffices to construct an oracle machine which computes f with recourse to an oracle for f'. That is, choosing any lookup function M for the problem f and defining the bijection $g_{\langle M\rangle}$ as in the previous Lemma, it is enough to construct an oracle machine which computes f with recourse to an oracle for the problem $f'_{\text{new}}(n) := f(g_{\langle M\rangle}(n))$.

If M' is any lookup machine for f'_{new}, we can define an oracle machine X that inputs a number n, applies the algorithm in Example 2.2.12 to double that number, and applies the algorithm of M' to consult the oracle tape and find the value of $f'_{\text{new}}(2n)$. By this procedure, we have

$$X_{f'_{\text{new}}}(n) = M'_{f'_{\text{new}}}(2n) = f'_{\text{new}}(2n) = f'(g_{\langle M\rangle}(2n)) = f'(\langle M\rangle, n).$$

But the lookup machine for f halts and outputs 1 precisely for inputs n such that $f(n) = 1$. On the other hand, since the lookup machine always halts, the lookup machine fails to output 1 precisely for inputs n such that $f(n) = 0$. Therefore, by definition of the Turing jump, we have $f'(\langle M\rangle, n) = 1$ if and only if $f(n) = 1$, and we have $f'(\langle M\rangle, n) = 0$ if and only if $f(n) = 0$. Therefore, by the equation displayed above, we have

$$X_{f'_{\text{new}}}(n) = f'(\langle M\rangle, n) = f(n)$$

for all natural numbers n. Hence X is an oracle machine which computes f with recourse to an oracle for f'_{new}, so f must be Turing reducible to f'. $\square$

Lemma 4.2.9. *For any problem f, we have $f' \not\leq_T f$.*

Proof. The proof is essentially identical to the proof that the halting problem is undecidable. We will reproduce it here for the reader's convenience.

Suppose for contradiction that there exists an oracle machine which computes f' with recourse to an oracle for f. That is, defining the bijection g as in the Lemma 3.2.3, suppose there exists an oracle machine M' which solves the problem $f'_{\text{new}}(n) := f'_{\text{new}}(g(n))$ with recourse to an oracle for f.

Using M', we can construct a new oracle machine N^* which inputs the natural number representation of an oracle machine, applies the algorithm in Example 2.2.12 to double that number, and then applies the algorithm of the machine M'. Notice that by this procedure, the computation $N^*(\langle M\rangle)$ yields the value given by

$$N^*_f(\langle M\rangle) = M'_f(2\langle M\rangle) = f'_{\text{new}}(2\langle M\rangle) = f'(g(2\langle M\rangle)) = f'(\langle M\rangle, \langle M\rangle).$$

By definition of the jump operator, the machine N^* determines whether or not a given oracle machine accepts its own natural number representation as an input when run with recourse to an oracle for f.

Given the machine N^*, we can easily construct another machine M^* which gives $M^*_f(\langle M\rangle) = 1$ if $N^*_f(\langle M\rangle) = 0$ and gives $M^*_f(\langle M\rangle) = 0$ if $N^*_f(\langle M\rangle) = 1$.

Now consider the value given by $M^*_f(\langle M^*\rangle)$. Since M' and N^* halt on any input, so must M^*, so in particular M^* must halt on the input $\langle M^*\rangle$. From the definition of M^*, it is easy to see that $M^*_f(\langle M^*\rangle)$ is either zero or one.

We now divide into cases. If $M^*_f(\langle M^*\rangle) = 1$, then $f'(\langle M^*\rangle, \langle M^*\rangle) = 0$, so by the definition of the jump operator, we do not have $M^*_f(\langle M^*\rangle) = 1$. This directly contradicts our assumption that $M^*_f(\langle M^*\rangle) = 1$. On the other hand, if $M^*_f(\langle M^*\rangle) = 0$, then we have $f'(\langle M^*\rangle, \langle M^*\rangle) = 1$. Therefore, by the definition of the jump operator, the machine M^* halts with output 1 on the input $\langle M^*\rangle$. But this contradicts the assumption that $M^*_f(\langle M^*\rangle) = 0$. Therefore, an oracle machine M' which computes f' with recourse to an oracle for f cannot exist, so $f' \not\leq_T f$. $\square$

Theorem 4.2.10. *For any problem f, we have $f <_T f'$.*

Proof. By Lemma 4.2.8, we know that $f \leq_T f'$. By Lemma 4.2.9, we have $f' \not\leq_T f$. Therefore, by definition of Turing equivalence, we have $f \leq_T f'$ and $f \not\equiv_T f'$, and the claim follows. $\square$

Corollary 4.2.11. *No problem is maximal with respect to Turing reducibility. That is, there exists no problem f such that $f \not<_T g$ for every other problem g.*

This result can be extended to Turing degrees. Indeed, if $\mathbf{a}$ is a Turing degree, then we can choose a problem $f \in \mathbf{a}$ and consider the uniquely determined Turing degree $\mathbf{b}$ which contains the problem f'. We have $f \not\equiv_T f'$, so $\mathbf{a} \not\equiv \mathbf{b}$. Also for any problems $a \in \mathbf{a}$ and $b \in \mathbf{b}$, we have

$$a \equiv_T f \leq_T f' \equiv_T b$$

and therefore $a \leq_T b$. Since a, b are arbitrary, this means that $\mathbf{a} \leq \mathbf{b}$. But since $\mathbf{a} \not\equiv \mathbf{b}$ and $\mathbf{a} \leq \mathbf{b}$, we can conclude that $\mathbf{a} < \mathbf{b}$. Therefore, there is no maximal Turing degree.

References

[1] D. Gordon, “Turing machines, diagonalization, the halting problem, reducibility,” September 2015.

[2] R. I. Soare, “Turing oracle machines, online computing, and three displacements in computability theory,” pp. 21–22, January 2009.

[3] M. Davis, “What is Turing reducibility?,” *Notices of the American Mathematical Society*, vol. 53, p. 1219, November 2006.

[4] T. A. Slaman, “Aspects of the Turing jump,” p. 1, 2000.