

Notes For CMSC 652 ON COMPUTABILITY  
William Gasarch

## 1 Introduction

This course will be on *Complexity Theory*. The basic question in computer science theory is

**Given a problem  $A$ , how hard is it?**

In a course in algorithms you might find an algorithm (duh) for problem  $A$  and try to prove that  $A$  is “easy”. In a course in complexity theory you want to prove that a problem  $A$  is “hard”. This is a harder task since to find an algorithm is easier than showing that none exists.

Computability theory looks at problems in terms of whether they can be solved *at all*. Many notions in Complexity Theory are modeled on similar notions from Computability Theory (which came first). In this set of notes we briefly review some basic computability theory for background.

## 2 Turing Machines

We will now try to pin down what is meant by “computable.” This definition is motivated by actual computers and resembles a machine. The definition is similar to that of a Deterministic Finite Automaton, or Push Down Automaton, but it can do much much more. Keep in mind that we want to show that this model can compute a lot of functions.

We first give the formal definition, and then explain it intuitively.

**Def 2.1** A *Turing Machine*  $M$  is a quintuple  $(Q, \Sigma, \delta, q_0, h)$  where

1.  $Q$  is a finite set of *states*
2.  $\Sigma$  is the *alphabet* (the function computed by the Turing Machine will go from  $\Sigma^*$  to  $\Sigma^*$ ), It contains a special symbol  $B$  which stands for BLANK.
3.  $\delta$  is the *next move function*. The domain and range of  $\delta$  are specified below.

$$\delta : (Q - \{h\}) \times \Sigma \rightarrow Q \times \{\Sigma \cup \{L, R\}\}.$$

4.  $q_0 \in Q$  is the *start state*
5.  $h \in Q$  is the *halting state*

The machine acts in discrete steps. At any one step it will read the symbol in the “tape square”, see what state it is in, and do one of the following:

1. write a symbol on the tape square and change state,
2. move the head one symbol to the left and change state,
3. move the head one symbol to the right and change state.

We now formally say how the machine computes a function. This will be followed by intuition.

**Def 2.2** Let  $M$  be a Turing Machine. An *Instantaneous Description* (ID) of  $M$  is a string of the form  $\alpha_1 q \alpha_2$  where  $\alpha_1, \alpha_2 \in \Sigma^*$ ,  $q \in Q$ , and the last symbol of  $\alpha_2$  is not  $B$ . Intuitively, an ID describes the current status of the TM and Tape.

**Def 2.3** Let  $M$  be a Turing Machine. Let  $\alpha_1, \alpha_2, \alpha_3, \alpha_4 \in \Sigma^*$ , and  $q, q' \in Q$ . Let  $\alpha_1 = x_1 x_2 \cdots x_k$ , and  $\alpha_2 = x_{k+1} x_{k+2} \cdots x_n$ . The symbol  $\alpha_1 q \alpha_2 \vdash_M \alpha_3 q' \alpha_4$  means that one of the following is true:

1.  $\delta(q, x_k) = (q', y)$ ,  $\alpha_3 = x_1 x_2 \cdots x_{k-1} y$  and  $\alpha_4 = \alpha_2$ .
2.  $\delta(q, x_k) = (q', L)$ ,  $\alpha_3 = x_1 x_2 \cdots x_{k-1}$  and  $\alpha_4 = x_k x_{k+1} \cdots x_n$ .
3.  $\delta(q, x_k) = (q', R)$ ,  $\alpha_3 = x_1 x_2 \cdots x_{k+1}$  and  $\alpha_4 = x_{k+2} x_{k+3} \cdots x_n$ .

Intuitively, the above definition is saying that if the Turing Machine is in ID  $C$ , and  $C \vdash_M D$ , then after *one* more move the TM will be in  $D$ . The next definition is about what will happen after *many* moves.

**Def 2.4** If  $C$  and  $D$  are IDs then  $C \vdash_M^* D$  means that either  $C = D$  or there exist a finite set of IDs  $C_1, C_2, \dots, C_k$  such that  $C = C_1$ , for all  $i$ ,  $C_i \vdash_M C_{i+1}$ , and  $C_k = D$ .

**Def 2.5** Let  $M$  be a Turing Machine. Recall that the *partial function computed by Turing Machine  $M$*  is the following partial function:  $f(x)$  is the unique  $y$  (if it exists) such that  $x q_0 \vdash_M^* y h$ . If no such  $y$  exists then  $M(y)$  is said to diverge.

Intuitively we start out with  $x$  laid out on the tape, and the head looking at the rightmost symbol of  $x$ . The machine then runs, and if it gets to the halt state with the condition that there are only blanks to the right of the head, then the string to the left of the head is the value  $f(x)$ .

Note that, just like a computer, the computation of a Turing machine is in discrete steps.

**Def 2.6** Let  $M_e$  be a Turing Machine and  $s$  be a number. The partial function computed by  $M_{e,s}$  is the function that, on input  $x$ , runs  $M_e(x)$  for  $s$  steps and if it has halted by then, outputs whatever  $M_e(x)$  output, else diverges.

Note that the function computed by  $M_{e,s}$  is intuitively computable. Although it is a partial function we can tell when it will be undefined so we can think of it as being total.

**Notation 2.7** If  $M(y)$  is defined we write  $M(y) \downarrow$ . If  $M(y)$  diverges then we write  $M(y) \uparrow$ .

## 2.1 Variations of a Turing Machine

There are many variations on Turing Machines that could be defined- allowing extra tapes, extra heads, allowing it to operate on a two dimensional grid instead of a one dimensional tape, etc. All of these models end up being equivalent. This adds to the intuition that Turing Machines are powerful.

**Theorem 2.8** *If  $f$  can be computed by a  $k$ -tape Turing Machine then  $f$  can be computed by an ordinary 1-tape Turing Machine.*

We will not prove this. With some care one can prove the following which we also won't prove.

1. If  $f$  can be computed by a  $k$ -tape Turing Machine such that, on inputs of length  $n$ ,  $\leq T(n)$  steps are used, then there is a 2-tape Turing Machine that computes  $f$  in time  $\leq O(T(n) \log T(n))$ .
2. If  $f$  can be computed by a  $k$ -tape Turing Machine such that, on inputs of length  $n$ ,  $\leq T(n)$  steps are used, then there is a 1-tape Turing Machine that computes  $f$  in time  $\leq O(T(n)^2)$ .

These results are not important for computability theory; however, they will be very important for the foundations of complexity theory. Other enhancements to a Turing Machine such as extra heads, two-dimensionality, allowing a 2-way infinite tape, do not add power and these proofs also do not increase the time that much.

## 2.2 Godelization

By using variations of Turing Machines it is not hard to show that standard functions such as addition, multiplication, exponentiation, etc. are all computable by Turing Machines. We will not give these proofs.

What else do computers do? Well, computers have to have programs that, given *a program*, run it. Is there a Turing machine that will, given a Turing Machine and an input, run it? There is! But we need to define terms carefully. For one thing, what does it mean to 'input a Turing Machine.'

We must code machines by numbers. In this subsection we give an explicit coding and its properties. The actual coding is not that interesting or important and can

be skipped, but should at least be skimmed to convince yourself that it really can be carried out. The properties of the coding are very important. A more abstract approach to this material would be to DEFINE a numbering system as having those properties. We DO NOT take this approach. If you are interested then look up ‘Acceptable Programming Systems’ on the web.

**Def 2.9** A *Godelization* is an onto mapping from  $\mathbf{N}$  to the set of all Turing Machines such that given a Turing Machine, one can actually find the number mapped to, and given a number one can actually find the Turing Machine that maps to it.

We define a Godelization. Let  $M = (Q, \Sigma, \delta, q_0, h)$  be a Turing Machine. We assume the following:

- there are  $n + 1$  states labeled  $1, 2, 3, \dots, n + 1$ ,
- state  $n + 1$  is the halting state,
- the alphabet is the numbers  $3, 4, 5, \dots, m$ .
- $L, R$  are represented by the numbers 1 and 2. (We still denote L and R by L and R. Note that L and R have numbers different from those in the alphabet.)

We first show how to encode a rule as a number:

Let  $q_1, q_2 \in Q$  and  $\sigma_1, \sigma_2 \in \Sigma$ . (By our convention,  $q_1, q_2, \sigma_1, \sigma_2$  are numbers). The rule

$$\delta(q_1, \sigma_1) = (q_2, \sigma_2)$$

is represented by the number  $2^{q_1} 3^{\sigma_1} 5^{q_2} 7^{\sigma_2}$ . The representations for rules that have L or R in the last component are defined similarly. In any case we denote the rule that says what  $\delta(q, \sigma)$  does by  $c(q, \sigma)$ .

We now code the entire machine  $M$  as a number. Let  $p_i$  denote the  $i$ th prime. Let  $\langle -, - \rangle$  be such that the map  $(i, j) \rightarrow \langle i, j \rangle$  is a bijection from  $\mathbf{N} \times \mathbf{N}$  to  $\mathbf{N}$  which is computable by a Turing Machine. The Turing Machine  $M$  is coded by the number

$$C(M) = \prod_{i=1}^n \prod_{j=1}^m p_{\langle i, j \rangle}^{c(i, j)}$$

With our current coding, although all Turing Machines correspond to numbers, not all numbers correspond to Turing Machines. We alleviate this by convention:

**Def 2.10** Turing Machine  $M_i$  is the machine corresponding to number  $i$ , if such a machine exists, and is the machine  $(\{q\}, \{a\}, \delta, q, q)$  where  $\delta(q, a) = (q, a)$ , (i.e. the easiest machine that halts on all inputs) otherwise. The function computed by  $M_i$  is denoted  $\varphi_i$ . We may also say that  $i$  is the index of  $M_i$  or  $\varphi_i$ .

It is easy to see that a program could be written to, given a Turing Machine  $M$ , find  $x$  such that  $M = M_x$ . (We will later be assuming that a Turing Machine could carry out such a task). It is also easy to see that a program could be written to, given a number  $x$ , determine  $M_x$ .

The number  $x$  codes all the information about  $M_x$  that one might wish to know. There is nothing inherently good about the coding we used, virtually any coding one might come up with has these properties. The properties essentially say that we can treat the indices of a Turing Machines as though they were programs.

## 2.3 Other Models and the Moral of the Story

Many models of computation have been proposed. All of them have a notion of discrete time steps, as does a Turing Machine.

1. Turing Machines were proposed by Alan Turing in 1936.
2.  $\lambda$ -calculus was proposed by Alonzo Church in 1941. The  $\lambda$ -calculus enables one to speak of functions from sets of functions to sets of functions. The language LISP is based on  $\lambda$ -calculus.
3. Post Systems were proposed by Emil Post in 1943. They are a generalization of Grammars.
4. Wang machines were proposed by Hao Wang in 1957.
5. Markov Algorithms were proposed by Andrei Andreivich Markov in the 1940's.
6. Register machines were proposed by Abraham Robinson and Calvin Elgot in the 1960's, and Random Access Machines were proposed by Steven Cook and Robert Rehow in the 1970's. Both resemble an actual computer more than most models.

These models of computation had very different motivations. Now for the surprise: **THEY ALL COMPUTE THE SAME CLASS OF (PARTIAL) FUNCTIONS!** In addition, the time loss in going from one to the other is (in most cases) only a polynomial e.g. if a Markov algorithm can compute a function  $f$  and use  $T(n)$  steps on inputs of length  $n$ , then there is a Turing Machine that can compute  $f$  and takes  $T(n)^k$  steps on inputs of length  $n$ .

We have been trying to formalize what it means for a function to be “intuitively computable.” This seems like a hard concept to define rigorously. But several people who tried to formalize this notation came to the SAME class. This leads one to make a leap of faith and conclude that yes indeed, this class of functions suffices:

**Church's Thesis:** Any (partial) function that is intuitively computable (e.g. we can write down a program for it in some informal language) is computable by a Turing Machine (thus by the  $\lambda$ -calculus, etc.).

For the remainder of this course we will speak in terms of Turing Machines, but will virtually never have to worry about the formal details of a machine. To show a function is computable we will write an informal program that computes it, and show that it works.

We repeat two definitions that we made earlier, noting that in both the term “Turing Machine” can be replaced by any of the above models.

**Def 2.11** A function computed by a Turing Machine is a *partial computable function*. If the function is total then we say it is *computable*.

### 3 Computable and Uncomputable Sets

Up to this point we have been speaking of functions. Sets are easier to study and more flexible. Most of the rest of the course will be about sets.

**Def 3.1** A set  $A$  is *computable* if there exists a Turing Machine  $M$  that behaves as follows:

$$M(x) = \begin{cases} 1 & \text{if } x \in A, \\ 0 & \text{if } x \notin A. \end{cases}$$

Computable sets are also called decidable or solvable. A machine such as  $M$  above is said to decide  $A$ .

#### 3.1 The HALTING Problem

In this subsection we exhibit a concrete example of a set that is not computable. Recall that  $M_x$  is the  $x$ th Turing Machine in the Godelization defined earlier.

**Def 3.2** The HALTING set is the set

$$K_0 = \{ \langle x, y \rangle \mid M_x(y) \text{ halts} \}.$$

Let us ponder how we would TRY to determine if a number  $\langle x, y \rangle$  is in the halting set. Well, we could try RUNNING  $M_x$  on  $y$ . If the computation halts, then GOOD, we know that  $\langle x, y \rangle \in K_0$ . And if it doesn't halt then – WHOOPS – if it never halts we won't know that!! It seems hard to determine with certainty that the machine will NOT halt EVER.

**Theorem 3.3** *The set  $K_0$  is not computable.*

**Proof:**

We show that  $K_0$  is NOT computable, by using diagonalization. Assume that  $K_0$  is computable. Let  $M$  be the Turing Machine that decides  $K_0$ . Using  $M$  we can easily create a machine  $M'$  that operates as follows:

$$M'(x) = \begin{cases} 0 & \text{if } M_x(x) \text{ does not halt,} \\ \uparrow & \text{if } M_x(x) \text{ does halt.} \end{cases}$$

Since  $M'$  is a Turing Machine, it has a Godel number, say  $e$ , so  $M_e = M'$ . We derive a contradiction by seeing what  $M_e$  does on  $e$ .

If  $M'(e) \downarrow$  then by the definition of  $M'$ , we know that  $M_e(e)$  does not halt, but since  $M' = M_e$ , we know that  $M_e(e)$  does halt. Hence the scenario that  $M'(e) \downarrow$  cannot happen. (This is not a contradiction yet)

If  $M'(e) \uparrow$  then by the definition of  $M'$ , we know that  $M_e(e)$  does halt; but since  $M' = M_e$ , we know that  $M_e(e)$  does not halt. Hence the scenario that  $M'(e) \uparrow$  cannot happen. (This alone is not a contradiction)

By combining the two above statements we get that  $M'(e)$  can neither converge, nor diverge, which is a contradiction. ■

This proof may look unmotivated— why define  $M'$  as we did? We now look at how one might have come up with the halting set if one's goal was to come up with an explicit set that is not decidable:

We want to come up with a set  $A$  that is not decidable. So we want that  $M_1$  does not decide  $A$ ,  $M_2$  does not decide  $A$ , etc. Let's make  $A$  and machine  $M_i$  differ on their value of  $i$ . So we can DEFINE  $A$  to be

$$A = \{i \mid M_i(i) \neq 1\}.$$

This set can easily be shown undecidable— for any  $i$ ,  $M_i$  fails to decide it since  $A$  and  $M_i$  will differ on  $i$ . But looking at what makes  $A$  hard intuitively, we note that the “ $\neq 1$ ” is a red herring, and the set

$$B = \{i \mid M_i(i) \downarrow\}$$

would do just as well. This is essentially the Halting problem.

**Corollary 3.4** *The set  $K = \{e \mid M_e(e) \downarrow\}$  is undecidable.*

**Proof:** In the proof of Theorem 3.3, we actually proved that  $K$  is undecidable. ■

**Note 3.5** In some texts, the set we denote as  $K$  is called the Halting set. We shall later see that these two sets are identical in computational power, so the one you care to dub THE halting problem is not important. We chose the one we did since it seems like a more natural problem. Henceforth, we will be using  $K$  as our main workhorse, as you will see in a later section.

## 3.2 Undecidable sets and $m$ -reductions

Now that we have  $K$  undecidable, we can show that other sets are undecidable as well. Our proofs will be along the lines of “to show that  $A$  is undecidable we show that if  $A$  were decidable, then so would be  $K$ , thus  $A$  cannot be undecidable.”

We start with an easy one:

**Example 3.6** The set

$$A = \{x \mid x - 17 \in K\}$$

is undecidable. Assume that  $A$  is decidable via Turing Machine  $M$ . Using this, we show that  $K$  is decidable.

1. Input( $x$ )
2. Run  $M$  on the input  $x + 17$ . (Output whatever it outputs.)

Note that  $x \in K$  iff  $x+17 \in A$ , so the algorithm decides  $K$ . This is a contradiction, so  $A$  is undecidable.

**Exercise 1** Show that the set

$$B = \{x \mid x + 17 \in K\}$$

is undecidable.

The key part of the proof that  $A$  is undecidable is to find a very nice question to ask  $M$ , in this case the question  $x + 17 \in A$ ? In most proofs, the hard part is finding the right question to ask, a question whose answer is informative in terms of determining if  $x \in K$ .

We now give a harder example.

**Example 3.7** Show that the set

$$A = \{x \mid \varphi_x(17) \downarrow\}$$

is undecidable. Assume  $A$  is decidable via Turing Machine  $M$ . We use  $M$  in an algorithm to decide  $K$ .

1. Input( $x$ )
2. Create a machine  $M'$  that does the following (DO NOT RUN THIS MACHINE!!!!!!!! ONLY CREATE IT.)
  - (a) Input( $z$ )
  - (b) Run  $M_x(x)$  (note that we are NOT using the input  $z$  here).

Find the Godel number  $y$  of  $M'$  (so  $\varphi_y$  is the function computed by  $M'$ )

3. Run  $M(y)$ . (Output whatever it outputs.)

To see that this algorithm decides  $K$  note that

$x \in K \Rightarrow M_x(x) \downarrow \Rightarrow M'$  will halt on all inputs  $\Rightarrow M'$  will halt on 17  $\Rightarrow$  the Godel number of  $M'$  is in  $A \Rightarrow M(y)$  will say YES.

$x \notin K \Rightarrow M_x(x) \uparrow \Rightarrow M'$  will not halt on any input  $\Rightarrow M'$  will not halt on 17  $\Rightarrow$  the Godel number of  $M'$  is not in  $A \Rightarrow M(y)$  will say NO.

**Exercise 2** In the above proof, the number 17 was not relevant. (It was a ‘red herring’ or a ‘paper tiger’) Name some other sets for which the proof that  $A$  is undecidable applies equally well to.

All the proofs that sets are undecidable, in this subsection, have had a very similar flair. We now codify these proofs—that is, define some terms and prove some theorems that will be used in all future proofs without a need for explicit mention.

As mentioned before, the key point is finding the appropriate question (or questions) to run  $M$  on to find out things about  $K$ . The following definitions and theorems make this notion rigorous:

**Def 3.8** Let  $A$  and  $B$  be any two sets.  $A$  is *m-reducible to B*, denoted  $A \leq_m B$  if there is a computable function  $f$  such that  $x \in A$  iff  $f(x) \in B$ .

The ‘ $m$ ’ in the above definition is because  $f$  can be many-to-one, as opposed to the following definition:

**Def 3.9** Let  $A$  and  $B$  be any two sets.  $A$  is *one-reducible to B*, denoted  $A \leq_1 B$  if there is a computable one-to-one function  $f$  such that  $x \in A$  iff  $f(x) \in B$ .

We will not be concerned with whether our reductions are 1-1 except in an occasional exercise.

**Exercise 3** Show that if  $A \leq_m B$  then  $\bar{A} \leq_m \bar{B}$ . Show that if  $A \leq_1 B$  then  $\bar{A} \leq_1 \bar{B}$ .

**Theorem 3.10** If  $A$  and  $B$  are sets,  $B$  is computable, and  $A \leq_m B$ , then  $A$  is computable.

**Proof:** Let  $f$  be computed by  $M$  and  $B$  be decided by  $N$ . The following algorithm decides  $A$

1. Input( $x$ )
2. Run  $M$  on  $x$  and call the result  $y$  (note that  $y$  is  $f(x)$ ).

3. Run  $N$  on  $y$ . (Output whatever it outputs.)

$x \in A \Rightarrow f(x) \in B \Rightarrow N(y)$  will say YES.

$x \notin A \Rightarrow f(x) \notin B \Rightarrow N(y)$  will say NO. ■

**Corollary 3.11** *If  $K \leq_m A$  then  $A$  is undecidable.*

**Proof:** By the above theorem, if  $A$  is decidable then  $K$  is decidable. This is a contradiction. ■

**Corollary 3.12** *If  $\bar{K} \leq_m A$  then  $A$  is undecidable.*

**Proof:** Similar. ■

We will use Corollary 3.11 implicitly for the rest of these notes: one way to show a set is undecidable will be to exhibit an algorithm for a reduction  $f$  that reduces  $K$  to that set. All sets shown undecidable in this subsection can be recast in that form. We do that:

**Example 3.13** We show that the set

$$A = \{e \mid \varphi_e(17) \downarrow\}$$

is undecidable. We show  $K \leq_m A$ .

1. Input( $x$ )

2. Create a machine  $M'$  that does the following (DO NOT RUN THIS MACHINE!!!!!! ONLY CREATE IT.)

(a) Input( $z$ )

(b) Run  $M_x(x)$

3. Find the Godel number  $y$  of  $M'$  and output it.

$x \in K \Rightarrow M_x(x) \downarrow \Rightarrow M'$  will halt on all inputs, including 17  $\Rightarrow$  the Godel number of  $M'$  is in  $A$

$x \notin K \Rightarrow M_x(x) \uparrow \Rightarrow M'$  will not halt on any inputs, including 17  $\Rightarrow$  the Godel number of  $M'$  is not in  $A$

We now give MANY examples of sets that are undecidable.

**Example 3.14** The set

$$A = \{e \mid \varphi_e \text{ is total} \}$$

is undecidable. The following algorithm computes an  $m$ -reduction  $f$  of  $K$  to  $A$ .

1. Input( $x$ )
2. Create a machine  $M'$  that does the following:
  - (a) Input( $z$ )
  - (b) Run  $M_x(x)$
3. Output the index of  $M'$ .

The reasoning is left as an exercise.

**Example 3.15** The set

$$A = \{e \mid \varphi_e \text{ computes the square function} \}$$

is undecidable. The following algorithm computes an  $m$ -reduction  $f$  of  $K$  to  $A$ .

1. Input( $x$ )
2. Create a machine  $M'$  that does the following:
  - (a) Input( $z$ )
  - (b) Run  $M_x(x)$
  - (c) (Note that this step will not be reached unless  $M_x(x)$  halts) Compute  $z^2$  and output it.
3. Output the index of  $M'$ .

$x \in K \Rightarrow$  for all  $z$  the computation of  $M'(z)$  will complete the computation of  $M_x(x)$  and then compute  $z^2$ , so it will always output  $z^2 \Rightarrow$  the index of  $M'$  is in  $A$ .

$x \notin K \Rightarrow$  for all  $z$  the computation of  $M'(z)$  will diverge while it is trying to compute  $M_x(x) \Rightarrow M'$  diverges on all inputs  $\Rightarrow$  the index of  $M'$  is NOT in  $A$ .

**Example 3.16** The set

$$A = \{e \mid \text{the domain of } \varphi_e \text{ is a noncomputable set} \}$$

is undecidable. The following algorithm computes an  $m$ -reduction  $f$  of  $K$  to  $A$ .

1. Input( $x$ )
2. Create a machine  $M'$  that does the following:
  - a) Input( $z$ )

b) Run  $M_x(x)$

c) (Note that this step will not be reached unless  $M_x(x)$  halts) Run  $M_z(z)$

3. Output the Godel number of  $M'$

$x \in K \Rightarrow$  for all  $z$ , the computation of  $M'(z)$  will finish step  $b$ , and then proceed to run  $M_z(z)$ , hence the domain of  $M'$  is  $K \Rightarrow \text{domain}(M')$  is noncomputable  $\Rightarrow$  Godel number of  $M'$  is in  $A$ .

$x \notin K \Rightarrow$  for all  $z$ , the computation of  $M'(z)$  will not finish step  $b$  hence the domain of  $M'$  is  $\emptyset \Rightarrow \text{domain}(M')$  is computable  $\Rightarrow$  Godel number of  $M'$  is not in  $A$ .

### 3.3 Turing Reductions

In the last section we showed that sets  $A$  were undecidable by showing that if  $A$  was decidable, then by using a program for  $A$  we could decide  $K$ . But we only used that program in a limited way. We only called it once.

We want a more powerful type of reduction. For example, informally it is clear that  $A \times \bar{A} \leq A$  under some type of reduction.

**Def 3.17** (Informal) Let  $A$  be any set. A set  $B$  is computable in  $A$  if there is a Turing Machine that, together with a “subroutine” for  $A$  can decide  $B$ . The set  $A$  is called an oracle. We denote the fact that  $B$  is computable in  $A$  by  $B \leq_T A$ , and say that “ $B$  is Turing-less than  $A$ ”

It is easy to see that for all sets  $A$ ,  $A \times \bar{A} \leq_T A$ .

We will not deal with Turing reductions in this course except in an occasional exercise. Professional Computability theorists deal mostly with Turing reductions.

### 3.4 Undecidable problems that are NOT based on Turing Machines

All the undecidable problems encountered so far have been sets or functions that deal with Turing machines. The question arises, are there any “NATURAL” undecidable problems. One could argue that HALT actually is natural, but we seek problems that do not mention Turing machines.

There are some such problems. The proofs that they are unsolvable usually entail showing that a Turing machine computation can be coded into them. However they are, on the face of it, natural. We list them but do not give proofs.

1. POST’S CORRESPONDENCE PROBLEM. Let  $\Sigma$  be a finite alphabet.

INPUT:  $A = \{\alpha_1, \dots, \alpha_n\}$ ,  $B = \{\beta_1, \dots, \beta_n\}$  where  $\alpha_i, \beta_j \in \Sigma^*$ .

OUTPUT: YES if there is a word  $w$  and an integer  $m$  such that  $w$  can be formed out of  $m$  symbols of  $A$  (repeats allowed), and also out of  $m$  symbols

in  $B$  (repeats allowed). NO otherwise. (Formally we say YES if there exists  $m, i_1, \dots, i_m, j_1, \dots, j_m$  such that

$$\alpha_{i_1} \alpha_{i_2} \cdots \alpha_{i_m} = \beta_{j_1} \beta_{j_2} \cdots \beta_{j_m}$$

2. HILBERT'S TENTH PROBLEM. Given a polynomial in many variables  $p(x_1, \dots, x_n)$  with integer coefficients, do there exist integers  $a_1, \dots, a_n$  such that  $p(a_1, \dots, a_n) = 0$ .
3. HILBERT'S TENTH PROBLEM (improvements) Given a polynomial in just 13 variables  $p(x_1, \dots, x_{13})$  with integer coefficients, do there exist integers  $a_1, \dots, a_{13}$  such that  $p(a_1, \dots, a_{13}) = 0$ .
4. CFG UNIV. Given a Context Free Grammar, does it generate EVERYTHING.
5. CFG EQUIV. Given two Context Free Grammars, do they generate the same set?
6. CFG NON-EMPTINESS. Given a Context Free Grammar, does it generate any strings?
7. OPTIMAL DPDA PROBLEM Given a Push Down Automata for a language, and promised that the language is actually recognizable by a deterministic Push Down Automata, find the size of the smallest Deterministic Push Down Automaton that will recognize it.
8. WORD PROBLEM FOR GROUPS (If you do not understand this problem do not worry, the point is that there are unsolvable problems in a branch of math called Group theory, which is NOT a branch of Logic or Recursion Theory.) Given a group by generators and relations, and then given a word, is it the identity?
9. TRIANGULATION PROBLEM FOR MANIFOLDS (Even if you do not understand this problem the point is that there are undecidable problems in Geometry.) Given two triangulations of four-dimensional manifolds, are those manifolds homeomorphic?

## 4 Sets that are even harder than HALT

Are there sets that are even "harder to decide" than HALT? We first say what this means formally:

**Def 4.1** If  $A \leq_T B$ , but  $B \not\leq_T A$ , then  $B$  is *harder than*  $A$ .

In this section we exhibit sets that are harder than  $K$  but do not prove this. Recall that  $K$  can be written as

$$K = \{e \mid (\exists s)M_e(e) \text{ halts in } s \text{ steps}\}.$$

Note that we have one quantifier followed by a COMPUTABLE statement.

**Def 4.2** A set  $A$  is *Computably Enumerable (c.e.)* if any of the following equivalent statements hold.

1. There is a computable set (often called a predicate)  $B$  such that

$$A = \{e \mid (\exists s)[(e, s) \in B]\}.$$

2. There is a Turing Machine  $M$  such that  $x \in A \iff M(x) \downarrow$
3. There is a Turing Machine  $M$  such that

$$A = \{x \mid (\exists y)[M(y) = x]\}.$$

4. There is a Total Turing Machine  $M$  such that

$$A = \{x \mid (\exists y)[M(y) = x]\}.$$

We will later see that c.e. is analogous to NP. But for now we are concerned with finding sets that are “harder” than this. We consider quantifiers.

How can  $TOT$  be written in terms of quantifiers?

$$TOT = \{e \mid (\forall x)(\exists s)M_e(x) \text{ halts in } s \text{ steps}\}.$$

This is two quantifiers followed by a computable statements.

It turns out that  $TOT$  cannot be written with only one quantifier and is harder than  $K$ . We can classify sets in terms of how many quantifiers it takes to describe them. Adjacent quantifiers of the same type can always be collapsed into one quantifier.

**Def 4.3**  $\Sigma_n$  is the class of all sets  $A$  that can be written as

$$A = \{x \mid (\exists y_1)(\forall y_2) \cdots (Qy_n)R(x, y_1, y_2, \dots, y_n)\},$$

where  $R$  is a computable relation and  $Q$  is  $\exists$  if  $i$  is odd, and  $\forall$  if  $i$  is even.

**Def 4.4**  $\Pi_n$  is the class of all sets  $A$  that can be written as

$$A = \{x \mid (\forall y_1)(\exists y_2) \cdots (Qy_n)R(x, y_1, y_2, \dots, y_n)\},$$

where  $R$  is a computable relation and  $Q$  is  $\forall$  if  $i$  is odd, and  $\exists$  if  $i$  is even.

**Def 4.5** A set is  $\Sigma_n$ -complete if  $A \in \Sigma_n$  and for all sets  $B \in \Sigma_n$ ,  $B \leq_m A$ .

We now state a theorem without proof.

**Theorem 4.6** For every  $i$  there are sets in  $\Sigma_i - \Pi_i$ , there are sets in  $\Sigma_{i+1} - \Sigma_i$ , there are  $\Sigma_i$ -complete sets, and there are  $\Pi_i$ -complete sets.

**Exercise 4** (You may use the above Theorem.) Show that a  $\Sigma_i$ -complete set cannot be in  $\Pi_i$ .

**Exercise 5** Show that  $K$  is  $\Sigma_1$ -complete. Show that  $\overline{K}$  is  $\Pi_1$ -complete.

**Exercise 6** Show that if  $A$  is  $\Pi_i$ -complete then  $\overline{A}$  is  $\Sigma_i$ -complete.

We show that  $FIN$  (the set of indices of Turing machines with finite domain) is in  $\Sigma_2$  and that  $COF$  (the set of Turing machines with cofinite domains) is in  $\Sigma_3$ . It turns out that  $FIN$  is  $\Sigma_2$ -complete, and  $COF$  is  $\Sigma_3$ -complete, though we will not prove this. As a general heuristic, whatever you can get a set to be, it will probably be complete there.

$$FIN = \{e \mid (\exists x)(\forall y, s)[ \text{If } y > x \text{ then } M_{e,s}(y) \uparrow ]\}$$

$$COF = \{e \mid (\exists x)(\forall y)(\exists s)[ \text{If } y > x \text{ then } M_{e,s}(y) \downarrow ]\}$$