Primitive Rec, Ackermann's Function, Decidable, Undecidable, and Beyond
Exposition by William Gasarch

1 Primitive Recursive Functions

We would like to *formally* define some notion of "computable functions." We *attempt* to define a set of functions that contains only computable functions, and contains all of them. This attempt will fail, but the reasons for this are of interest. In a later section we will succeed.

In this attempt to define "computable functions" we try to use only basic operations (e.g. the operation "add 1"), and basic ways of putting functions together (e.g. composition). We do this to make the model as simple as possible, and to guarantee that all functions generated are computable.

Def 1.1 A function $f(x_1, \ldots, x_n)$ is *primitive recursive* if either:

1. f is the function that is always 0, i.e. $f(x_1, \ldots, x_n) = 0$; This is denoted by Z when the number of arguments is understood. This rule for deriving a primitive recursive function is called the Zero rule.

2. f is the successor function, i.e. $f(x_1, \ldots, x_n) = x_i + 1$; This rule for deriving a primitive recursive function is called the Successor rule.

3. f is the projection function, i.e. $f(x_1, \ldots, x_n) = x_i$; This is denoted by π_i when the number of arguments is understood. This rule for deriving a primitive recursive function is called the Projection rule.

4. f is defined by the composition of (previously defined) primitive recursive functions, i.e. if $g_1(x_1, \ldots, x_n)$, $g_2(x_1, \ldots, x_n)$, ..., $g_k(x_1, \ldots, x_n)$ are primitive recursive and $h(x_1, \ldots, x_k)$ is primitive recursive, then

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$$

   is primitive recursive. This rule for deriving a primitive recursive function is called the Composition rule.

5. f is defined by recursion of two primitive recursive functions, i.e. if $g(x_1, \ldots, x_{n-1})$ and $h(x_1, \ldots, x_{n+1})$ are primitive recursive then the following function is also primitive recursive

$$\begin{aligned} f(x_1, \ldots, x_{n-1}, 0) &= g(x_1, \ldots, x_{n-1}) \\ f(x_1, \ldots, x_{n-1}, m+1) &= h(x_1, \ldots, x_{n-1}, m, f(x_1, \ldots, x_{n-1}, m)) \end{aligned}$$

This rule for deriving a primitive recursive function is called the Recursion rule. It is a very powerful rule and is why these functions are called 'primitive recursive.'

To show some function is primitive recursive you build it up from these rules. Such a proof is called a derivation of that primitive recursive function.

We give some examples of primitive recursive functions. These examples will be given both rather formally (more formal than is really needed) and less formally. After these examples we shall always be less formal. The only reason to give the rather formal definition is to show that it can be done. In practice the informal one is just as informative (actually more so) and has the intuition that the formal one sorely lacks.

There are two points these examples are making: (1) LOTS of functions are primitive recursive. Most functions that you have dealt with in your lives have been primitive recursive. (2) It isn't that hard to show that functions are primitive recursive.

Example 1.2 $f(x) = x + 5$. Very formally we would say:

1. $S(x) = x + 1$ is primitive recursive by the successor rule.
2. $S(S(x))$ is primitive recursive by the composition rule, composing the function defined in 1 with the function defined in 1.
3. $S(S(S(x)))$ is primitive recursive by the composition rule, composing the function composed in 2 with the function defined in 1.
4. $S(S(S(S(x))))$ is primitive recursive by the composition rule, composing the function defined in 3 with the function defined in 1.
5. $S(S(S(S(S(x)))))$ is primitive recursive by the composition rule, composing the function defined in 4 with the function defined in 1.

Informally we would say: $f(x) = S(S(S(S(S(x)))))$ is primitive recursive by successor and several applications of composition.

Example 1.3 $f(x, y) = x + y$. Very formally we would say

1. $g(x) = \pi_1(x) = x$ is primitive recursive by the projection rule.
2. $\pi_3(x, y, z) = z$ is primitive recursive by the projection rule.
3. $S(x) = x + 1$ is primitive recursive by the successor rule.
4. $h(x, y, z) = S(\pi_3(x, y, z)) = z + 1$ is primitive recursive by the composition rule and composing S and π_3.

5. Define f using the recursion rule and g, h:

$$\begin{aligned} f(x,0) &= g(x) = \pi_1(x) = x \\ f(x,n+1) &= h(x,n,f(x,n)) = S(\pi_3(x,n,f(x,n))) = f(x,n)+1 \end{aligned}$$

Informally, just say

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

The question arises, "how does one think of these definitions?" The key point is to think recursively. The thought process for defining plus might be "Well gee, $x+0=x$, that's easy enough, and $x+(y+1)=(x+y)+1$, so I can define $x+(y+1)$ in terms of $x+y$." From that point it should be easy to see how to formalize that $f(x,y)=x+y$ is primitive recursive.

Now that we have plus is primitive recursive, we can use it to define other primitive recursive functions. Here we use it to define multiplication.

Example 1.4 $f(x,y)=xy$ is primitive recursive via

$$\begin{aligned} f(x,0) &= 0 \\ f(x,y+1) &= f(x,y)+x \end{aligned}$$

Now that we have multiplication is primitive recursive, we use it to define powers.

Example 1.5 $f(x,y)=x^y$ is primitive recursive via

$$\begin{aligned} f(x,0) &= 1 \\ f(x,y+1) &= xf(x,y) \end{aligned}$$

Subtraction would be a nice operation to have, but primitive recursive function only go from N to N. We do have a "truncated subtraction" called "monus"

Example 1.6 In all the above examples we took a *well known* function and showed it was primitive recursive. Here we *define* a function directly. What this function does is, if $x>0$ then it subtracts 1, otherwise it just returns 0.

$$\begin{aligned} f(0) &= 0 \\ f(x+1) &= x \end{aligned}$$

To do this formally, recall that in the recursion rule the function h can depend on x and $f(x)$. Henceforth, call this function $M(x)$.

We now use M to define a version of subtraction.

Example 1.7 $x \dot{-} y$. This function is $x - y$ if $x - y \geq 0$, otherwise it is 0.

$$\begin{aligned} f(x,0) &= x \\ f(x,y+1) &= M(f(x,y)) \end{aligned}$$

Example 1.8 We can now show that $|x-y|$ is primitive recursive. $|x-y| = (x \dot{-} y) + (y \dot{-} x)$.

Fact 1.9 *Virtually every function you can think of (primality, finding quotients and remainders, any polynomial) is primitive recursive.*

Exercise Define a notion of primitive recursive function from $(\Sigma^*)^n$ to Σ^*, where $\Sigma = \{a, b\}$. Show that concatenation is primitive recursive. (Although there are many different ways of doing this, some lead to much cleaner definitions of concatenation than others. We would like a clean definition.)

Exercise Define a notion of primitive recursive function from the integers to the integers. Show that subtraction is primitive recursive. (Although there are many different ways of doing this, some lead to much cleaner definitions of subtraction than others. We would like a clean definition.)

Exercise Show that there is a JAVA program computing the Successor functions, Projection functions, and the Zero function.

We claim that the set of primitive recursive functions are "computable." To back this up, we prove the following

Theorem 1.10 *Every primitive recursive function can be computed by a JAVA program*

Proof: We prove this by induction on the formation of a primitive recursive function. More formally, by induction on the number of steps used to derive the primitive recursive function. (WARNING: the name of the thing we will be inducting on is m, NOT n as is customary. n will be the number of arguments of the primitive recursive function being discussed.) BASE CASE: The only primitive recursive functions that can be derived with only one step are the functions $S_i(x_1, \ldots, x_n)$, $\pi_i(x_1, \ldots, x_n)$, and $Z(x_1, \ldots, x_n)$. These are all computable by a JAVA program by the Exercise above. INDUCTION STEP: Let $f(x_1, \ldots, x_n)$ be a function that is primitive recursive by a derivation of $m + 1$ steps.

If the last rule used in the derivation of $f(x_1, \ldots, x_n)$ is either the Zero rule, Projection rule, or Successor rule then $f(x_1, \ldots, x_n)$ is one of those functions, and thus by the exercise $f(x_1, \ldots, x_n)$ is primitive recursive.

If the last rule used was composition then there are functions $g_1(x_1, \ldots, x_n)$, $g_2(x_1, \ldots, x_n)$, ..., $g_k(x_1, \ldots, x_n)$, $h(x_1, \ldots, x_k)$ which are primitive recursive such that each of those function takes $\leq m$ steps to derive, and

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))).$$

Since each of the g_i's, and h, take m or less steps to derive, BY THE INDUCTION HYPOTHESIS they can be computed by a JAVA program. Using this, we write a JAVA program for f.

begin
    input$(x_1, \ldots, x_n)$;
    output$(h(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)))$;
end

This program was easy to write because we already knew we had JAVA programs for $g_1, \ldots, g_k$ and h.

We leave the last case, where the last rule was recursion, to the reader as an exercise. It is not hard. ∎

Is EVERY function that is computed by a JAVA program a primitive recursive functions? We will show the answer is NO.

Exercise Describe a mapping ρ from N onto the set of all primitive recursive functions of many variable. The function $U(x, y) = \rho(x)(y)$ should be JAVA computable. Why do you think I chose the letter U? (The actual details of the JAVA code aren't necessary, just argue that it can be done.)

Theorem 1.11 *There exists a function that is computed by a JAVA program that is not primitive recursive.*

Proof: Let ρ and U be the functions defined in the above exercise. We write a JAVA program for the sole purpose of NOT being primitive recursive. The idea is that the program's behavior on x will disagree with the primitive recursive function

$\rho(x)$ on the value x. Since the range(ρ) is the set of ALL primitive recursive functions, this will suffice.

PROGRAM DIAGONALIZE

begin

read(x);

output($U(x,x)+1$) ;

end

Let F be the function computed by the program DIAGONALIZE. We show that F is not primitive recursive. Assume, by way of contradiction, that F is primitive recursive. Then by the definition of ρ there exists an n such that F is $\rho(n)$. Hence

$$(\forall x)[F(x) = \rho(n)(x) = U(n,x)].$$

In particular, take $x = n$ to get

$$F(n) = U(n,n).$$

But by the definition of F via the JAVA program

$$F(n) = U(n,n) + 1.$$

This is a contradiction! Hence F is not computed by any primitive recursive function. ∎

How important is Theorem 1.11? The theorem itself seems to just say that PRIMITIVE RECURSIVE is not the notion we need, and perhaps some other notions will suffice. However, the proof of Theorem 1.11 was not particular to primitive recursive functions, or to JAVA. Any formalism that attempts to capture all the functions that are computable is going to run into the same kind of problem (namely, an analog of Theorem 1.11) as primitive recursive functions did. This is formalized in the following exercise.

Exercise Let $\{f_1, f_2, f_3, \ldots\}$ be a set of functions in one variable. Assume that the function $g(i,x) = f_i(x)$ is JAVA computable. Show that there is a function h that is JAVA computable, but h is not one of the f_i.

How to get around this dilemma? We take this point up in the next section.

Although the function in Theorem 1.11 is not primitive recursive, it is contrived. It was defined for the sole purpose of not being primitive recursive and would not arise in practice. Are there "natural" JAVA computable functions that are not primitive recursive? We define a JAVA-computable function which is not primitive recursive which we consider "natural." Since naturalness is in the eye of the beholder, you should feel free to disagree as to the naturalness of the function.

Def 1.12 *Ackermann's function* is the function defined by

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

It is easy to see that Ackermann's function is JAVA computable.

The following is true but we won't prove it:

Theorem 1.13 *Ackermann's function is computed by a JAVA program but is not primitive recursive.*

Though we won't prove it, here are some facts:

1. Ackermann's function grows very fast. It grows faster than any primitive recursive function.

2. Given a primitive recursive function, its derivation used the recursion rule some finite number of times.

3. Since Ackermann's function is defined using a recursion which involves applying the function to itself there is no obvious way to take the definition and make it primitive recursive. While this is not a proof, it is an intuition.

Theorem 1.11 gives an example of a computable-but-not-prim-rec function that is contrived. Theorem 1.13 gives an example of a computable-but-not-prim-rec function that is ... not contrived? To put it more clearly: Is Ackermann's function natural?

Yes!

It has actually come up on some natural places.

1. The DISJOINT-SET data structure (YES, a data structure!) supports the following operations: It keeps track of sets of elements and it can (1) make new sets (this is a way to add elements by adding a 1-element set), (2) union two sets, (3) find an element. We would like to do each operation in $O(1)$ steps. We cannot do this. Oh well. But:

   - Let $A^{-1}(n)$ be the inverse of Ackermann's function. So this is much much slower growing then, say $(\log\log\log\log n)$ or (if you know what is is $(\log^*(n))$). The DISJOINT-SET data structure does n operations in time $O(nA^{-1}(n)$ steps. Its amortized- we are not saying it does each one in $A^{-1}(n)$ steps.
   - OKAY, thats kind-of interesting, but it could be that the analysis is not as good as it could be (though Tarjan did win a Turing award) or that a better data structure is possible. If so then the appearance of ACK would not really be intrinsic. BUT see next item.

- One can show (and this requires defining a model of computation in tune to this setting) that $\Omega(nA^{-1}(n))$ is THE BEST YOU CAN DO! So we have matching upper and lower bounds that involve Ackerman!

2. For another natural appearance of Ackerman, see Wikipedia Entry on Goodstein's Theorem.

2 General Recursive Functions

The primitive recursive functions were an attempt to pin down formally what it means for a function to be computable. It failed since we could construct a program that computed a function that was JAVA-computable (hence computable) but, not primitive recursive (this was Theorem 1.11).

The proof of Theorem 1.11 was not particular to primitive recursive functions, or to JAVA. As noted after the theorem, any formalism that attempts to capture all the functions that are computable is going to run into the same kind of problem (namely, an analog of Theorem 1.11) as primitive recursive functions did.

How to get around this dilemma? One is to give up our hope of pinning down the set of functions that are computable and defined everywhere, and instead be content with looking at the *partial functions* that are computable. This also takes care of the problem that JAVA programs don't always halt.

Def 2.1 A *partial function* is a function whose domain is a subset of N. A *total function* is a function whose domain is N. We denote the domain of a partial function f by domain(f).

We will add a basic way of generating one function from another which will allow functions to not always be defined, yet is basic and is something that machines can really do. Intuitively, we allow the operation of being able to search for a number without having an *a priori* bound on what the number may be (in fact, it might not exist). The resulting set of functions will be called the *General Recursive Functions*.

Keep in mind our final goal of showing that this model is very powerful- many functions will turn out to be general recursive.

Notation 2.2 The symbol μ stands for "least number such that." We illustrate its use. If $g(x)$ is a function then $\mu x[g(x) = 13]$ is the least number x such that $g(x) = 13$. Note that such a number need not exist, in which case $\mu x[g(x) = 13]$ is undefined. If $g(x_1, \ldots, x_n, y)$ is a function then $f(x_1, \ldots, x_n) = \mu y[g(x_1, \ldots, x_n, y) = 0]$ is the function that, on input $\langle a_1, \ldots, a_n \rangle$ returns the least value of y such that $g(\langle a_1, \ldots, a_n \rangle, y) = 0$. Note that such a number need not exist, in which case $f(\langle a_1, \ldots, a_n \rangle)$ is undefined. The function f is called the *unbounded minimization of* g.

Example 2.3 Let $g(x, y) = x + y$. Let

$$f(x) = \mu y[g(x + y) = 100].$$

$f(0) = 100$, $f(1) = 99$, ..., $f(99) = 1$, $f(100) = 0$, and f is undefined on all other values.

Def 2.4 A function $f(x_1, \ldots, x_n)$ is *general recursive* if either

1. f is primitive recursive
2. f is defined by unbounded minimization on a (previously defined) general recursive function $g(x_1, \ldots, x_n, y)$, i.e. there is a general recursive function g and

$$f(x_1, \ldots, x_n) = \mu y[g(x_1, \ldots, x_n, y) = 0].$$

Def 2.5 The general recursive functions are also called the *partial computable functions*. The subset of the partial computable functions that are total are called the *total computable functions*, or just the *computable functions*. The term "general recursive" will not be used again in this course, but is included for historical purposes.

We will later see that general recursive functions are very powerful.

3 Turing Machines

We will now take a different approach to pinning 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 our final goal is to show that this model can compute a lot of functions.

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

Def 3.1 A *Turing Machine* M is a quintuple $(Q, \Sigma, \delta, s, h)$ where

1. Q is a finite set of *states*
2. Σ is the *alphabet* (the function computed by the Turing Machine will go from Σ^* to Σ^*), It contains a special symbol B which stands for BLANK.
3. δ is the *next move function*, it goes from $(Q - \{h\}) \times \Sigma$ to $Q \times \{\Sigma \cup \{L, R\}\}$
4. $s \in Q$ 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 3.2 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 α_2 is not B. Intuitively, an ID describes the current status of the TM and Tape.

Def 3.3 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 3.4 If C and D are IDs then $C \vdash_M^* D$ means that either $C = D$ or there exist a finite set of IDs $C_1, C_2, \ldots, C_k$ such that $C = C_1$, for all i, $C_i \vdash_M C_{i+1}$, and $C_k = D$.

Def 3.5 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 $xs \vdash_M^* yh$. 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 $f(x)$.

For examples of Turing machines, and exercises on building them to do things, see Lewis and Papadimitriou's text "Elements of the Theory of Computation.", or the Hopcroft-Ullman White book. Other books also contain this material.

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

Def 3.6 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 3.7 If $M(y)$ is defined we write $M(y)\downarrow$. If $M(y)$ diverges then we write $M(y)\uparrow$.

4 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.

Def 4.1 A k-tape Turing Machine is a quintuple $(Q, \Sigma, \delta, s, h)$ such that Q, Σ, s, and h are as in a normal Turing Machine, but δ is a function from $(Q - \{h\}) \times \Sigma^k$ into $Q \times \{\Sigma \cup \{L, R\}\}^k$. A *configuration* is a k-tuple of strings of the form $\alpha q \beta$ where $q \in Q$, $\alpha, \beta \in \Sigma^*$ the last symbol of β is not B (where B is the special blank symbol), and all the q in all the tuples are the same. If the input is x then the standard initial configuration is formed by assuming that x is on the first tape, the head on the first tape is pointing to the rightmost symbol of x, and on all other tapes the head is at the leftmost symbol of the tape.

It turns out that extra tapes do not increase power.

Theorem 4.2 *If f can be computed by a k-tape Turing Machine then f can be computed by an ordinary Turing Machine.*

A careful analysis of the proof of the above theorem reveals that the 1-tape machine is not that much more inefficient then the equivalent 2-tape machine. In particular, we have actually shown that if the 2-tape machine halts on inputs of length n in $T(n)$ steps, then the 1-tape machine will halt, on inputs of length n, in $T(n)^2$ steps. While this is not important for recursion theory, it will be a significant fact in complexity theory. The best known simulation of a multitape Turing Machine by a fixed number of tape machine is that any function f that can be computed by k-tape Turing Machine in $T(n)$ steps on inputs of length n can be computed by a 2-tape machine in $T(n) \log T(n)$. (See Hopcroft-Ullman, the White book.)

Other enhancements to a Turing Machine such as extra heads, two-dimensional, allowing a 2-way infinite tape, do not add power. Note that a Turing Machine with many added features resembles an actual computer.

Exercise Discuss informally how to convert various variants of a Turing Machine to a 1-tape 1-head 1-dim Turing Machine. Comment on how runtime and number of states are affected.

5 Godelization

By using variations of Turing Machines it would not be hard to show that standard functions such as addition, multiplication, exponentiation, etc. are all computable by Turing Machines. We wish to examine functions that, in some sense, take Turing Machines as their input. In order to do this, 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, but will discuss it at the end of this section.

Def 5.1 A *Godelization* is an onto mapping from 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 $M = (Q, \Sigma, \delta, s, h)$ be a Turing Machine. We assume the following:

- there are $n+1$ states labeled 1,2,3,$\ldots, n+1$,
- state $n+1$ is the halting state,
- the alphabet is the numbers 3,4,5$\ldots, m$.
- 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 ith prime. Let $\langle -, - \rangle$ be such that the map $(i, j) \to \langle i, j \rangle$ is a bijection from $\mathsf{N} \times \mathsf{N}$ to 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 5.2 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 φ_i We may also say that i is the index of M_i or φ_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.
Exercise Informally show that the following functions are computable:

- Given x, determine the number of states in M_x.
- Given x, determine the number of symbols in the alphabet of M_x.
- Given numbers x, y, s, determine if M_x halts on y in less than s steps.
- Given numbers x and y, produce the code for the Turing Machine that computes the composition of the functions computed by M_x and M_y.

Our coding has some very nice properties that we now state as theorems. 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. We will be using these theorems informally, without explicit reference to them, for most of this course.

Theorem 5.3 *(s-1-1 Theorem) There exists a primitive recursive function $s_{1\text{-}1}$ such that for all $x, y,$ and z*

$$M_x(y, z) = M_{s_{1\text{-}1}(x,y)}(z)$$

.

Intuitively this is saying that parameters and code are interchangeable. If JAVA was being used instead of Turing Machines, the s-1-1 function would merely be replacing a READ statement with a CONST statement.

The above theorem is better known in its generalized form:

Theorem 5.4 *(s-m-n Theorem) For every m and n there exists a primitive recursive function $s_{m\text{-}n}$ such that for all x, $\langle y_1, \ldots, y_{n+1} \rangle$, and $\langle z_1, \ldots, z_m \rangle$*

$$M_x(\langle y_1, \ldots, y_{n+1} \rangle, \langle z_1, \ldots, z_m \rangle) = M_{s_{m-n}(x, \langle y_1, \ldots, y_{n+1} \rangle)}(\langle z_1, \ldots, z_m \rangle)$$

.

Both the s-1-1 theorem and the s-m-n theorem are proven by actually constructing such functions. These constructions were of more interest when they were proven than they are now, since now the notion of treating data and parameters the same has been absorbed into our culture.

The next theorem says that there is one Turing Machine that can simulate all others. It is similar to a mainframe: you feed it programs and inputs, and it executes them.

Theorem 5.5 *(Universal Turing Machine Theorem, or Enumeration Theorem) There is a Turing Machine M such that $M(x, y)$ is the result of running M_x on y. (Note that this might diverge.)*

Convention 5.6 We will always denote the Universal Turing Machine by U.

We will be using the s-m-n Theorem and the existence of a Universal Turing Machine, throughout this course (usually implicitly). A more abstract approach would have been to build these two properties into definitions:

Def 5.7 An *acceptable programming system* (henceforth APS) is a list of all the Turing computable functions $\varphi_1, \varphi_2, \ldots$ such that the s-m-n Theorem, and the enumeration theorem are true relative to that numbering.

One concern might be that if we prove theorems for our particular APS will it be true for all APS's. The following theorem says YES, as it says that all APS's are essentially the same.

Theorem 5.8 *(Rogers isomorphism theorem) Let $\varphi_1, \varphi_2, \ldots$ and $\phi_1, \phi_2, \ldots$ be two APS's. There exists a bijection f, computable by a Turing Machine, such that $\varphi_i \equiv \phi_{f(i)}$.*

6 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. λ-calculus was proposed by Alonzo Church in 1941. The λ-calculus enables one to speak of functions from sets of functions to sets of functions. The language LISP is based on λ-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 Rechow 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 computes 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 λ-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 6.1 A function computed by a Turing Machine is a *partial computable function*. If the function is total then we say it is *computable*.

We now give examples of computable functions.

1. $f(x)$ is the xth prime, computable.

2. $f(x) = x^7 + 12x^5$, computable.

3. Ackermann's function is computable.

4. Any JAVA program that halts on all inputs you can think of is computing a computable function.

We give an interesting example of a partial computable function. We want a function that will, on input e, output some PRIME that M_e halts on. If M_e does not halt on any prime, then the function will be undefined.

First attempt (which will fail): run $M_e(2)$. If it halts then output 2, else run $M_e(3)$. If it halts then output 3, else run $M_e(5)$. This will not work since you cannot tell if $M_e(2)$ halts.

So what to do?

Well, we can try to run $M_e(2)$ for a few steps, then try $M_e(3)$ for a few steps, then go back to $M_e(2)$ and try out various other primes as we go. We try $M_e(p)$ for s steps for many primes p and numbers s. This process is known as DOVETAILING. Before presenting the formal algorithm we'll need pairing functions.

Def 6.2 Let π_1 and π_2 be computable function such that the set $\{(\pi_1(x), \pi_2(x)) : x \in \mathsf{N}\}$ is all of $\mathsf{N} \times \mathsf{N}$.

Algorithm for f:

1. Input(e)

2. $i := 1$

   FOUND := FALSE

   While NOT FOUND

   $x := \pi_1(i)$

   $s := \pi_2(i)$

   Run $M_e(x)$ for s steps.

   If x is prime and $M_e(x)$ halts within s steps then

   output(x)

   FOUND := TRUE

   else $i := i + 1$

The algorithm looks at ALL possible pairs (x, s) and if we find that $M_e(x)$ halts in s steps, and x is prime, then we halt. Note that if M_e halts on SOME prime then $f(x)$ will be such a prime; however, if M_e does not halt on any prime, then the algorithm will diverge (as it should).

7 Some strange examples of computable functions

Functions that are almost always 0 are very easy to compute: just store a table.

Example 7.1 Let f be the function that is $f(0) = 12$, $f(10) = 20$, $f(14) = 7$, f is zero elsewhere. The function f is easily seen to be computable. Just write a program with a lot of 'if' statements in it. It will output 0 on values that are not 0,10, or 14.

In the above example, the function f was given EXPLICITLY so it was easy to write the program. Even if a function is not given to us explicitly, we may be able to show that it is computable.

Example 7.2 Let f be the function that is nonzero on values less than 10, and on those values always outputs the input squared. From the description we can deduce that $f(1) = 1$, $f(2) = 4$, $f(3) = 9$, $f(4) = 16$, $f(5) = 25$, $f(6) = 36$, $f(7) = 49$, $f(8) = 64$, $f(9) = 81$, and f is zero elsewhere.

In the above example, even though we were not given the function explicitly, we could derive an explicit description from what was given. In the next example this is no longer the case, but the function is still computable.

INTERESTING EXAMPLE

One needs to know what the Goldbach Conjecture to appreciate this example: Goldbach's conjecture is still unknown. It is: every even is the sum of two primes.

Example 7.3 Let f be the function such that if Goldbach's conjecture is true then f is 74 on all numbers less than 4 and zero elsewhere, and if Goldbach's conjecture is false then f is 17 on all less than 3 and zero elsewhere. Since we don't know whether or not Goldback's Conjecture is true, WE DO NOT KNOW what f is. But we DO know that EITHER

1. $f(1) = 74$, $f(2) = 74$, $f(3) = 74$, $f(x) = 0$ elsewhere, OR
2. $f(1) = 17$, $f(2) = 17$, $f(x) = 0$ elsewhere.

So THERE EXISTS a JAVA program for f. In fact, we can write down two programs, and know that one of them computes f, but we don't know which one. But to show that f is computable WE DO NOT CARE WHICH ONE! The definition of computability only said THERE EXISTS a JAVA program, it didn't say we could find it.

An even more interesting example:

Example 7.4 Let f be the following function: if Goldbach's conjecture is false then f is 888 on the the smallest even n such that n cannot be written as the sum of two primes, and 0 elsewhere. if Goldbach's conjecture is true then f is always 0. If Goldbach's conjecture is false then f is one of the following.

1. $f(2) = 888$, f is zero elsewhere
2. $f(4) = 888$, f is zero elsewhere
3. $f(6) = 888$, f is zero elsewhere
4. etc. $\vdots$

The fact that this list is infinite should not bother us. It is still the case that f is computable since one of the functions on this list is f, or f is always 0.

These functions are computable EVEN THOUGH WE CAN"T FIND CODE FOR THEM.

If I asked you what a computable function was you might say

f is computable if there exists a TURING MACHINE to compute it.

I might say

f is computable if THERE EXISTS a Turing machine to compute it.

The key thing is that THERE EXISTS a Turing machine, even if I can't find it.

AN EXAMPLE OF 'I DO NOT KNOW AND I DO NOT CARE', that is not related to computer science:

Example 7.5 Do there exists two irrational numbers x and y such that x^y is rational? I will show you pairs (a, b), and (c, d) such that either

1. a and b are irrational and a^b is rational, OR
2. c and d are irrational and c^d is rational.

Even at the end of the proof I won't know which pair works.

Let $a = \sqrt{2}$, $b = \sqrt{2}$, $c = (\sqrt{2})^{\sqrt{2}}$, $d = \sqrt{2}$. We already know that $\sqrt{2}$ is irrational. If $(\sqrt{2})^{\sqrt{2}}$ is rational, then (a, b) works. If $(\sqrt{2})^{\sqrt{2}}$ is irrational then c is irrational, d is irrational, and

$$c^d = ((\sqrt{2})^{(\sqrt{2})})^{\sqrt{2}} = (\sqrt{2})^2 = 2$$

so the pair (c, d) works. I DO NOT KNOW whether or not $(\sqrt{2})^{\sqrt{2}}$ is irrational, but in either case, I get what I want, so for now I DO NOT CARE.

8 Computable and Computably Enumerable 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 8.1 A set A is *computable* if there exists a Turing Machine M that behaves as follows:

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

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

Some examples of computable sets.

1. The primes.
2. The Fibonacci numbers (any number in the set $1, 2, 3, 5, 8, 13, ...$ where every number is the sum of the previous number). If you want to know if a number x is a Fib number, just calculate the Fib numbers until you either spot x or surpass it. If you spot it then its a Fib number, if you surpass it, its not.
3. (x, y, s) such that $M_x(y)$ halts within s steps.
4. Most sets you can think of are computable.

Are there any noncomputable sets? Cheap answer: The number of SETS is uncountable, the number of COMPUTABLE SETS is countable, hence there must be some noncomputable sets. In fact, there are an uncountable number of them. I find this answer rather un enlightening.

9 The HALTING Problem

In this subsection we exhibit a concrete example of a set that is r.e. but not computable. Recall that M_x is the xth Turing Machine in the Godelization defined earlier.

Def 9.1 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 9.2 *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:

$$M'(x) = \begin{cases} 0 & \text{if } M_x(x) \uparrow \\ \uparrow & \text{if } M_x(x) \downarrow \end{cases} \tag{2}$$

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

Proof: In the proof of Theorem 9.2, we actually proved that K is undecidable. ∎

Note 9.4 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.

10 Computably Enumerable Sets

K_0 and K are not decidable. Well, what CAN we say about K_0 that is positive. Lets look back at our feeble attempt to solve K_0. The algorithm was: on input x, y, run $M_x(y)$ until it halts. The problem was that if $\langle x, y\rangle \notin K_0$ then the algorithm diverges. But note that if $(x, y) \in K_0$ then this algorithm converges. SO, this algorithm DOES distinguish K_0 from $\overline{K}_0$. But not quite in the way we'd like. The following definition pins this down

Def 10.1 A set A is *computably enumerable* (henceforth "r.e.") if there exists a Turing Machine M that behaves as follows:

$$M(x) \begin{cases} \downarrow & \text{if } x \in A \\ \uparrow & \text{if } x \notin A \end{cases} \tag{3}$$

Exercise Show that K and K_0 are r.e.
Exercise Show that if A and B are computable then $A \cap B$, $A \cup B$, and $\overline{A}$ are computable. Which of these are true for r.e. sets?

There is a definition of r.e. that is equivalent to the one given, and is more in the spirit of the words "compatibly enumerable."

Theorem 10.2 *Let A be any set. The following are equivalent:*

1. *A is the domain of a partial computable function (i.e. A is r.e.)*
2. *A is the range of a total computable function or $A = \emptyset$ (this definition is more like enumerating a set).*

Proof: We show 1) $\rightarrow$ 2) $\rightarrow$ 1).

1) $\rightarrow$ 2): Let A be the domain of a partial computable function f. Let M be a Turing Machine whose domain is A. If A is empty, then 2) is established. Assume that A is nonempty and let $a \in A$. Let g be the (total) computable function computed by the following algorithm:

1. Input(n).
2. If $n = 0$ then output a.
3. Compute $X = \{g(0), g(1), g(2), \ldots, g(n-1)\}$.
4. Let $Y = \{0, 1, 2, \ldots, n\}$. If $Y - X$ is empty then output a. If $Y - X$ is not empty then run M on every element of $Y - X$ for n steps. If there is some $y \in Y - X$ such that $M(y)$ halts within n steps then output the least such y. Else output a.

We show that range(g) =domain(f). If y is in the range of g then it must be the case that $M(y)$ halted, so y is in the domain of f. If y is in the domain of f then let n be the least number such that $M(y)$ halts in n steps and $y \leq n$. If there is some $m < n$ such that $g(m) = y$ then we are done. Otherwise consider the computation of $g(n)$. In that computation $y \in Y$ but might not be output if there is some smaller element of Y. The same applies to $g(n+1), g(n+2), \ldots$. If there are z elements smaller than y in A then one of $g(n), g(n+1), \ldots, g(n+z)$ must be y.

2) $\rightarrow$ 1). Assume that A is either empty or the range of a total computable function. If A is empty then A is the domain of the partial computable function that always diverges, and we are done. Assume A is the range of a total computable function f. Let g be the partial computable function computed by the following algorithm:

1. Input(n).
2. Compute $f(0), f(1), \ldots$ until (if it happens) you discover that there is an i such that $f(i) = n$. If this happens then halt. (if it does not, then the function will end up diverging, which is okay by us).

We show that an element n is in the range of f iff $g(n)$ halts. If n is in the range of f then there exists an i such that $f(i) = n$; this i will be discovered in the computation of g on n, so $g(n)$ will be 1. If $g(n)$ halts then an i was discovered such that $f(i) = n$, so n is in the range of f.

∎

Several questions arise at this point:

- Are there any sets that are r.e. but not computable?
- Are there any sets that are NOT r.e.?
- If a set is r.e., then is its complement r.e. ?

The second question can be answered in a cheap way: since there are an uncountable number of sets and a countable number of r.e. sets (since there are only a countable number of Turing Machines), there are an uncountable number non-r.e. sets. While this is true, it is not a satisfying answer. We will give more concrete answers to all these questions.

First we relate r.e. and computable sets.

Theorem 10.3 *A set A is computable iff both A and $\overline{A}$ are r.e.*

Proof: If A is computable then $\overline{A}$ is computable. Since any computable set is r.e. both are r.e.

Assume A and $\overline{A}$ are r.e. Let M_a be a Turing Machine that has domain A and M_b be a Turing Machine that has domain $\overline{A}$. The set A is computable via the following algorithm: on input x run both $M_a(x)$ and $M_b(x)$ simultaneously; if $M_a(x)$ halts then output YES, if $M_b(x)$ halts then output NO. Since either $x \in A$ or $x \in \overline{A}$, one of these two events must happen. ∎

This theorem links two of our questions: there exists an r.e. set that is not computable iff r.e. sets are not closed under complementation.

10.1 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 Σ be a finite alphabet.

   INPUT: $A = \{\alpha_1, \ldots, \alpha_n\}$, $B = \{\beta_1, \ldots, \beta_n\}$ 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 $m, i_1, \ldots, i_m, j_1, \ldots j_m$ such that

   $$\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_m} = \beta_{i_1}\beta_{i_2}\cdots\beta_{i_m}$$

2. HILBERT'S TENTH PROBLEM. Given a polynomial in many variables $p(x_1, \ldots, x_n)$ with integer coefficients, does there exist integers $a_1, \ldots, a_n$ such that $p(a_1, \ldots, a_n) = 0$.

3. HILBERT'S TENTH PROBLEM (improvements) Given a polynomial in just 13 variables $p(x_1, \ldots, x_{13})$ with integer coefficients, does there exist integers $a_1, \ldots, a_{13}$ such that $p(a_1, \ldots, a_{13}) = 0$.

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-EMPTYNESS. 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?

11 Sets that are even harder than HALT

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

Def 11.1 If $A \leq_{\mathrm{T}} B$, but $B \not\leq_{\mathrm{T}} A$, 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.

How can TOT be written:

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

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 11.2 Σ_n is the class of all sets A that can be written as

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

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

Def 11.3 Π_n is the class of all sets A that can be written as

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

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

Def 11.4 A set is Σ_n-complete if $A \in \Sigma_n$ and for all sets $B \in \Sigma_n$, $B \leq_{\mathrm{m}} A$.

We now state a theorem without proof.

Theorem 11.5 *For every i there are sets in $\Sigma_i - \Pi_i$, there are sets in $\Sigma_{i+1} - \Sigma_i$, there are Σ_i-complete sets, and there are Π_i-complete sets.*

Exercise (You may use the above Theorem.) Show that a Σ_i-complete set cannot be in Π_i.
Exercise Show that K is Σ_1-complete. Show that $\overline{K}$ is Π_1-complete.
Exercise Show that if A is Π_i-complete then $\overline{A}$ is Σ_i-complete.

We show that FIN (the set of indices of Turing machines with finite domain) is in Σ_2 and that COF (the set of Turing machines with cofinite domains) is in Σ_3. It turns out that FIN is Σ_2-complete, and COF is Σ_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.

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

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