Notes for CS 321

Paul Cull

Department of Computer Science
Oregon State University

January 4, 2016

Chapter 1

Turing's Legacy

1.1 What Is An Algorithm?

In his address to the International Congress of Mathematicians in 1900, David Hilbert set out a list of problems that he thought would be the leading mathematical problems in the 20th century. One of these, now known as Hilbert's Tenth Problem, asked:

> Is there an algorithm to determine if a Diophantine equation has a solution?

Diophantine refers to Diophantus of Alexandria who lived from about 200 to 284. A Diophantine equation is a polynomial equation with integral coefficients to which only integral solutions are sought. For example,

$$x + y = 5$$

has lots of solutions, but

$$x^4 + y^2 = 6$$

has no solutions.

The amazing thing about Hilbert's question was not that it asked how to solve a mathematical problem, but that it really focused on an *algorithm*, and brought to the attention of mathematicians the question:

What is an algorithm?

This question was worked on for the next for the next 40 years. A variety of answers were proposed by such people as Church, Gödel, Post, Kleene, and Markov. But in 1936, Alan Turing came up with the most intuitive proposal — the Turing machine. (Of course, Turing didn't call it a Turing machine, but that is the name we use now.) We will not describe the Turing machine here. There are many good descriptions, for example: Davis [], Minsky [], Hopcroft and Ullman [], Linz [], Herken []

The wonderful thing about Turing's idea is that it also leads to the idea of the stored program digital computer, or as we usually say a *computer*. Since we are familar with computers we can discuss many of Turing's results without worrying about the details of his model. In fact, we don't even have to worry about the details of the computer, we can think instead of a high level programming language, and we don't have to worry about the details of the language. Think in C or think in Lisp or even think in Fortran. As long as the language is strong enough to represent the programs we need, we don't need the details.

We can state a definition of *algorithm* which is sufficient for our purposes:

> An **algorithm** is a computer program which correctly solves **ALL** instances of a problem.

We all know algorithms for a variety of problems like sorting and addition. Of course, we are talking about idealized computer programs that do not have to contend with "real-world" limitations like too little time or too little space. So, we speak of an algorithm for addition even though we know that any "real" implementation could not add arbitrarily large numbers.

Turing's amazing insight was to realize that not all problems can have algorithms and to display a problem, the Halting Problem, which cannot have an algorithm.

1.2 Acceptors, Rejectors, and Recognizers

As part of his analysis of computation Turing realized that he could restrict the input to be $\mathbb{N}$ the set of natural numbers. This is not hard for us to believe because when we represent anything in a computer, it is ultimately mapped to a sequence of bits which we can interpret to be a natural number.

In Turing's analysis a problem was a mapping from natural numbers to natural numbers. But, he also realized that he could restrict his attention to **YES, NO** problems which take a natural number input and output either **YES** or **NO**.

In line with standard mathematical practice, a set is defined by a characteristic function, that is a function which takes an element x of the universe and outputs **YES** or **NO** depending on whether or not x is in the set. For a set to be computable, Turing said that its characteristic function had to be computable. But this meant that there had to be a computer program for each computable set. By a counting argument due to Cantor, there are an *uncountable infinity* of sets. Turing reasoned that there are only a *countable infinity* of computer programs and therefore almost all sets are **not computable**. He now faced a serious difficulty – how could he describe a set which he could not compute?

> *If to be computable means that something has a description that is a computer program, a set without a program seems to indescribable.*

In the next few sections we will see how Turing was able to solve this dilemma. But first, we will follow Turing in making some distinctions. He realized that when you give an input to a program, one of two things would happen – either the program eventually produces an answer and stops, or the program may continue computing forever. In common parlance, a program which does not halt is said – depending on one's choice of metaphor – to be in an "infinite loop" or to be in a "dead loop".

Because of this possibility of non-termination, Turing was able to distinguish several possible definitions for a computable set.

First, there are sets which are defined by programs which halt on each and every input. Such sets are called **Recursive** sets. The program which always halts and says whether or not the input is in the set is called a **recognizer**. Said another way, a set has a recognizer exactly when the set is a recursive set.

Second, there are sets which are defined by programs which only halt for some inputs. There are two obvious ways to describe a set using a sometimes halting program. We could decide that the set is defined as the input on which the program halts or we could define the set by inputs on which the program fails to halt.

When we define a set by the inputs on which the program halts, we say that the program *accepts* the set, and we call the program an **acceptor** for the set. That is, an **acceptor** for a set S is a program which halts when given any element x of S and which fails to halt when given any y which is not in S. To allow seemingly greater flexibility, we can also look at the output and say that an **acceptor** halts and says **YES** when given an element of S, but the acceptor can either fail to halt or halt and say **NO** when given an element not in S. (When we say *element* here we mean an element of the universe which by Turing's analysis we can take to be the natural numbers.) The class of sets which have acceptors is called **RE** which is an abbreviation for recursively enumerable. (See Exercise 1.7 for further explanation of the term *recursively enumerable.*)

To make our world of sets more symmetrical, we can also consider the sets defined by programs failing to halt. We say that a set S has a **rejector**, if there is a program which halts only when given an input which is **not** in S. That is, the rejector fails to halt when given an x which is in S. Again to add flexibility, we can look at the output of the rejector, and require that if y is not in S then the rejector must halt and say **NO**, but if x is in S then the rejector may either halt and say **YES** or fail to halt. We call the class of set which have rejectors the class **coRE** for complement of recursively enumerable.

In the following we summarize these distinctions using the notation:

$Rec_{\mathbf{S}}(x)$ for a recognizer for the set $\mathbf{S}$,
$Acc_{\mathbf{S}}(x)$ for an acceptor for the set $\mathbf{S}$, and
$Rej_{\mathbf{S}}(x)$ for a rejector for the set $\mathbf{S}$:

$$Rec_{\mathbf{S}}(x) = \begin{cases} \textbf{YES} & x \in S \\ \textbf{NO} & x \notin S \end{cases}$$

$$Acc_{\mathbf{S}}(x) = \begin{cases} \textbf{YES} \text{ (halts)} & x \in S \\ & \\ \textbf{doesn't halt} & x \notin S \\ \text{or } \{\textit{halts and says NO}\} & \end{cases}$$

$$Rej_{\mathbf{S}}(x) = \begin{cases} \textbf{doesn't halt} & x \in S \\ \text{or } \{\textit{ halts and says YES }\} & \\ & \\ \textbf{NO} \text{ (halts)} & x \notin S \end{cases}$$

1.3 The Halting Problem is Algorithmically Unsolvable

They said the job could not be done.
Some said they even knew it.
He faced the job that could not be done,
and, by God, he couldn't do it!

1.3.1 Turing's Proof

The proof of the algorithm unsolvablility of the Halting Problem, uses a diagonal argument which is a type of proof by contradiction. We asssume that there is an algorithm for this problem and from this assumption we derive a contradiction.

Assume that there is program $H(M, t)$ which takes as input a program M and also an input t for M. If $H(M, t)$ solves the halting problem, then when M halts given input t, H halts and says **YES**, and if M does **not** halt given input t, H halts and says **NO**. Notice that H always halts and always gives the correct answer.

Now from H we can build another program H_1 so that $H_1(M) = H(M, M)$. That is, H_1 reads its input (which should be a program M), makes a copy of M and feeds these two copies to the assumed program H. Then H_1 simply behaves like H. So H_1 will always halt and always give the correct answer to the question "Will program M halt when given itself (or, better, a copy of itself) as input?"

Since by assumption we have H_1, we can construct another program H_2 from H_1, but this H_2 will be constructed so that it does **NOT** halt on each and every input. In fact, we want $H_2(M)$ to **HALT** exactly when $H_1(M)$ says **NO**, and, of course, we want $H_2(M)$ to **FAIL to HALT** exactly when $H_1(M)$ says **YES**. This is easy to accomplish. We look at each of the **STOP** instructions in H_1 and if the **STOP** is a **STOP with output NO** we leave this instruction alone, but if the **STOP** is a **STOP with output YES**, we replace this instruction with an instuction (or series of instructions) which is a **LOOP FOREVER**. So $H_2(M)$ will go into an "infinite loop" if $H_1(M)$ says **YES**.

Now for the killer. What does $H_2(H_2)$ do? Clearly, it either **HALTs** or **doesn't HALT**. We will try each of these possibilities and derive a contradiction to the claim that $H(M, t)$ correctly solves the Halting Problem for each pair M, t.

First, assume that $H_2(H_2)$ does halt. Then $H_1(H_2) = \textbf{NO}$, and also $H(H_2, H_2) = \textbf{NO}$. This says that if H_2 given H_2 as an input halts, then H incorrectly claims that H_2 given H_2 does not halt.

On the other hand, assume that $H_2(H_2)$ does not halt. Then $H_1(H_2) = \textbf{YES}$, and also $H(H_2, H_2) = \textbf{YES}$. This says that if H_2 given H_2 as an input fails to halt, then H incorrectly claims that H_2 given H_2 does halt.

Since these are the only two possibilities for $H_2(H_2)$ and we conclude in each case that $H(H_2, H_2)$ is wrong, our original assumption that H correctly solved the Halting Problem is **WRONG**!

Because all we assumed about H is that it was an algorithm for the halting problem, we have proved:

Theorem 1 (Turing)**.** *There is no algorithm which correctly solves the Halting Problem.*
(Any purported algorithm must be incorrect in at least some cases.)

The following is a more pictorial representation of this proof:

The Unsolvability of the Halting Problem

$$H(M,t) \in \{\textbf{YES}, \textbf{NO}\} \quad H \text{ correctly states whether or not } M \text{ halts given } t$$

$$H_1(M) = H(M, M)$$

$$H_2(M) = \begin{cases} \textbf{HALTS} & \text{iff } H(M,M) = \textbf{NO} \\ \textbf{DOES NOT HALT} & \text{iff } H(M,M) = \textbf{YES} \end{cases}$$

$$H_2(H_2) = \begin{cases} \textbf{HALTS} & \text{iff } H(H_2,H_2) = \textbf{NO} \text{ i.e. } H \text{ is } \textbf{WRONG} \\ \textbf{DOES NOT HALT} & \text{iff } H(H_2,H_2) = \textbf{YES} \text{ i.e. } H \text{ is } \textbf{WRONG} \end{cases}$$

1.3.2 What Have We Shown?

In the terminology of the last section, we have shown that there is no recognizer for the Halting set. The Halting set is the set of **YES** instances of the Halting

Problem, that is, the Halting set is the set of all pairs (M, t) of programs M and input t, so that the program M eventually halts when given input t.

The Halting set has an acceptor.
Let **HALT-Set** $= \{(M, t) | M$ halts when given t as input$\}$.
Our informal algorithm is:

```
SIMULATE $M$ with input $t$
IF this simulation finishes OUTPUT( YES ).
```

(Later we'll make this more formal by showing that there is a Universal Turing Machine which can, in fact, simulate any Turing machine.)
Notice that this simulation can simply go on forever, in which case, we cannot say **YES** or **NO** to whether $(M, t) \in$ **HALT-Set**. Obviously, if the simulation goes on forever, we'd like to say **NO**, but how can we tell that the simulation will go on forever? We might hope that we can determine that the simulation is in an "infinite loop". In fact, we can detect *some* forms of "infinite loop" by keeping track of the history of our simulation. Let the STATE of the simulation be **ALL** the information we use in computing the next action in the simulation. If STATE$_{17}$, the state after 17 steps, is identical to STATE$_{128,432}$, the state after 128,432 steps, then we can be sure that STATE$_{128,433}$ will be identical to STATE$_{18}$, and thus, that the simulation is in an "infinite loop". Unfortunately, there are other ways the simulation can fail to finish. For example, it might start computing with some small numbers, and successively compute with larger and larger numbers. If we knew that a simulation with large enough numbers would not finish, then we could correctly say **NO**. BUT, we can't be sure. After computing with some very large numbers, the simulation may surprise us by suddenly finishing or returning to some much smaller numbers. The whole point of the algorithmic unsolvability of the Halting Problem is that we can never tell using an algorithm whether or not our simulation will finish.

Since the Halting set does have an acceptor. there are problems which have acceptors, but do not have recognizers. From here, it's obvious that coHalt, the set of (M, t) pairs so that M does not halt when given t, is an example of a set which has a rejector, but does not have a recognizer. In fact, coHalt cannot have an acceptor because of the result in Exercise 1.3, that is a set has a recognizer iff it has both an acceptor and a rejector.

1.3.3 Why is this called a diagonal argument?

The above proof of the unsolvability of the Halting Problem is usually called a **diagonal argument**. This form of argument goes back at least to Georg Cantor in the late 1800's. Cantor used this technique to show that there were different size infinities. For example, he showed that there are *more* real numbers than there are natural numbers. And since there are an infinite number of natural numbers, there are at least two sizes of infinity.

In the form we have presented the unsolvability of the Halting Problem there does not seem to be a diagonal, but we'll show that one is implicit in our argument.

We can represent a function by a table. As the index into the table we use the input values for the function, and in the body of the table we put the values of the function, i.e. the output values. For a function of two variables, we can us a 2-dimensional table. Down one side of the table, we put the values for one of the variables, and across the top of the table we put the values of the other variable. If the first variable is x and the second variable is y, then we put $f(x, y)$ in the x^{th} row and the y^{th} column of the table. So, if we had this table, we could compute the function $f(x, y)$. To find $f(17, 32)$ we would go to row 17 and then across to column 32 and we would find the function's value at that point.

The diagonal of such a function table appears in the obvious place. When x and y have the same set of possible values, the diagonal occurs when $x = y$, and the diagonal function is $f(x, x)$ which is a function of only one variable.

In our proof, the assumed halting function, $H(M, t)$ is a function which has a 2-dimensional table. The function $H_1(M)$ is the diagonal function. Unfortunately, we now come to an awful example of terminology, even though $H_1(M)$ is the obvious diagonal function, it is standard practice to call $H_2(M)$ the "diagonal function". That is, the **diagonal function** is the function created by changing (complementing) the values on the diagonal of a function table.

In outline, a diagonal argument takes a set of functions as the rows of a table, uses the input values for these functions for the columns, and then creates a "diagonal function" by working down the diagonal of the table and making the "diagonal function's" values different from the values found on the diagonal of the table. In this way, the "diagonal function" is sure to differ from each function in the table in at least one place, and so the "diagonal function" cannot be in the table.

This may sound a little different from using the table as a table for a function of two variables, but we can fit the two variable function in as a listing of functions by considering $H(M, t)$ to be $H_M(t)$, that is, the first variable M picks out which function of t we want to talk about.

1.4 No Programming Language Exactly Specifies All Recognizers

Here we use a diagonal argument to show that the class of Recursive sets cannot be captured by a programming language. Of course, we're talking about a language which specifies each and every recognizer. (As we will see later there are many classes of recognizers like Primitive-Recursive recognizers and Polynomial-Time recognizers which can be characterized by specific programming languages. Here, we're showing that those classes omit some recognizers, and that if a programming language can permit each recognizer to be expressed, then there are some non-recognizers which can be expressed in the programming language.)

Assume that there is a programming language for the recognizers, then we can list each and every recognizer by writing down the programs in this language.

Since we are listing programs, we can order them in some way. For example, we could order by length and use alphabetic ordering to list programs with the same length, or we could convert the programs into binary and after prepending a 1 to capture leading 0's we could order the programs as binary numbers. We can also list all possible inputs. Since we have the programs and the inputs listed, we can index them by the natural numbers and speak of the i^{th} program P_i or the j^{th} input t_j. We would like to construct a two dimensional table, with the rows representing programs and the columns representing inputs. E.G.,

	t_0	t_1	t_2	t_3	t_4	$\dots$
P_0	$P_0(0)$	$P_0(1)$	$P_0(2)$	$P_0(3)$	$P_0(4)$	$\dots$
P_1	$P_1(0)$	$P_1(1)$	$P_1(2)$	$P_1(3)$	$\dots$	
P_2	$P_2(0)$	$P_2(1)$	$\dots$			
P_3	$P_3(0)$	$\dots$				
.	.					
.	.					
.	.					

In the table at position (i, j) we would like to put the output when P_i is given input t_j.[1] By our assumption that that these programs are recognizers, the output value is in $\{\textbf{YES}, \textbf{NO}\}$ and is defined because recognizers halt on each input. Of course, our assumption that we have a programming language allows us to assume that when we have P_i and t_j, we can execute P_i given t_j. (This execution may be only theoretical because the execution may take more time than our lifetime or the lifetime of the universe, or use more memory space than is available or will be available in the foreseeable future.) After the execution we can fill in the (i, j) position in the table with the value $P_i(t_j)$. To save a little space we will call this $P_i(j)$.

Now that we have the table, the familiar diagonal technique can be applied. We build a new recognizer P^* so that:

$$P^*(i) = \begin{cases} \textbf{YES} & \text{iff } P_i(i) = \textbf{NO} \\ \textbf{NO} & \text{iff } P_i(i) = \textbf{YES}. \end{cases}$$

Clearly, P^* is a recognizer because it always gives a **YES/NO** answer, but P^* is different from each of the recognizers $P_0, P_1, P_2, \dots$ because P^* differs from P_i on the i^{th} input. (P^* may also differ from P_i on other inputs.)

How can this be? Our claim that P^* is a recognizer really depends on whether we can compute $P^*(i)$. We can compute $P^*(i)$ if we can compute $P_i(i)$. To compute $P_i(i)$ we need to find the program for P_i and we need to execute this program on the i^{th} input. So, if we really had a programming language for recognizers, the above diagonal construction builds a recognizer P^* which is *not* represented in this programming language. Therefore, our assumption that we had a programming language for exactly the recognizers must be false.

[1] Perhaps we should use $P_i(t_j)$ in this table, but $P_i(j)$ seems easier to read.

We could have a programming language which represents **SOME** of the recognizers but doesn't represent **ALL** recognizers. In particular, P^* would be a non-represented recognizer. Or, we could have a programming language which represents **ALL** the recognizers and some other programs which fail to halt on some inputs. In such a programming language P^* would not be a recognizer, that is, P^* would fail to halt on some inputs.

1.5 The Universal Turing Machine

Perhaps the most important of Turing's ideas is the **UNIVERSAL MACHINE** – a single machine which can do anything that any other machine can do. (Of course, there is not a unique universal machine, there are lots of universal machines.) The idea, which is familiar to computer programmers, is that one computer can be programmed to behave like any other computer. This idea of "general-purpose" machine was clear to Turing, but it took a decade or so for engineers to realize the importance of this idea and to embody it in electronic machines.

Specifically, Turing's universal machine $\mathcal{U}$ takes two inputs, M a program and t an input for M, and $\mathcal{U}$ can simulate the computation of M on t, so that

$$\mathcal{U}(M,t) \;=\; M(t),$$

that is, the output of $\mathcal{U}$ given the two inputs M and t is the same as the output of M given the input t.

1.5.1 Pairing Functions

Of course, this distinction between one and two inputs should strike us as inconsequential and it is. We can always think of two inputs as one and conversely we can always think of one input as two. We can capture this (to us) obvious statement mathematically as the claim that there is a one-to-one onto function from pairs of naturals $\mathbb{N} \times \mathbb{N}$ to single naturals $\mathbb{N}$. Let us call such a function a **pairing function.** As we might expect, there are lots of pairing functions, but we'll only describe one of them. We want:

$$PAIR : \mathbb{N} \times \mathbb{N} \overset{\text{one-to-one}}{\xrightarrow{\text{onto}}} \mathbb{N}$$

We argue diagrammatically that such a function exists and that it is computable. Consider the table:

	0	1	2	...	...
0					
1					
2					
.					
.					
.					

which has both rows and columns indexed by the naturals. We want to fill in the table so that each natural appears exactly once inside the table. As our old friend Cantor pointed out, this can be done in a simple manner by considering the counter-diagonals (those diagonals which go from Northeast to Southwest). So we fill in the table as:

	0	1	2	3	4	...
0	0	1	3	6	10	
1	2	4	7			
2	5	8				
3	9					
.						
.						

We leave as an exercise (see Exercise 1.9) finding a simple formula for this table. One look at this counter-diagonal method suggests that the inverse function

$$PAIR^{-1} : \mathbb{N} \longrightarrow \mathbb{N} \times \mathbb{N}$$

should also be easy to compute. We also leave this as an exercise for the reader (see Exercise 1.10). Notice that $PAIR^{-1}(n)$ also defines two functions $PAIR_1^{-1}(n)$ and $PAIR_2^{-1}(n)$ which each map $\mathbb{N} \longrightarrow \mathbb{N}$ so that

$$PAIR(\,PAIR_1^{-1}(n),\, PAIR_2^{-1}(n)\,) \;=\; n.$$

1.5.2 Back to Universal Machines

With these pairing functions we can set up a universal $M_{\mathcal{U}}$ which computes so that

$$M_{\mathcal{U}}(\,PAIR(M,t)\,) \;=\; \mathcal{U}(M,t) \;=\; M(t).$$

Here we are using our assumption that our machines are programs written in some programming language, and for the programming language we assume that there is a reasonable syntax, so we can reasonably tell which strings are syntactically valid programs. With these assumptions, we can reasonably talk about the i^{th} program, e.g. we can order the strings first by length and then alphabetically and then throw out the strings which have syntax errors. This allows our pairing function to deal only with natural numbers.

How does a universal machine work? Without giving the details, the universal machine simply "simulates" other machines. If we have our strings represented in some programming language, all our universal machine has to do is execute programs written in that programming language. Of course, the somewhat paradoxical idea is that the universal machine is also a program written in the same programming language. So we could, somewhat perversely, have

the universal machine simulating
a universal machine simulating
a universal machine simulating

$\vdots$

a universal machine.

For classification purposes, the universal machine serves as the *acceptor* for the **HALT** set. That is, if we want to find out if M halts when given t, we can run $\mathcal{U}(M,t)$ which will halt exactly when M given t halts.

The universal machine also serves as *rejector* for the **coHALT** set. The **coHALT** set is the set of all (M,t) where M is a program and t is an input, and M does **not** halt given t. So to see if (M,t) is **NOT** in **coHALT**, we run $\mathcal{U}(M,t)$ and if this halts, we know that M given t halts, and thus that (M,t) is **NOT** in **coHALT**. When(M,t) is in **coHALT**, $\mathcal{U}(M,t)$ computes forever and does not halt.

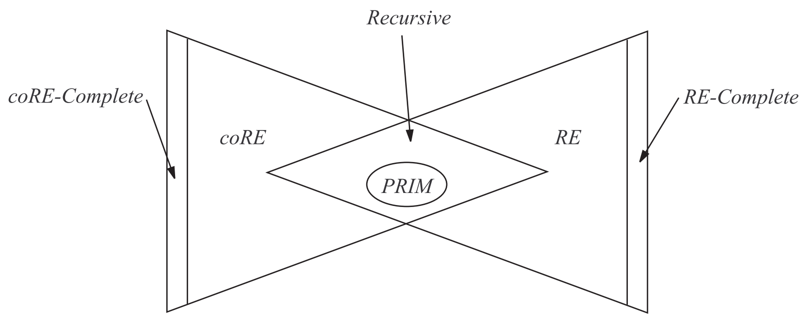
1.6 Primitive Recursion (Informal)

As we have seen above the recognizers cannot be exactly specified by a programming language. Our standard programming languages allow both acceptors and rejectors as well as recognizers. To limit ourselves to recognizers, we want a programming language which allows only halting programs. We can do this by outlawing unbounded loops. So, **WHILE** and **REPEAT ... UNTIL** are out. We still want to allow looping, but it must be bounded looping. In some languages, we could allow **FOR** because a **FOR** loop must terminate and by looking at the program we can even calculate an upper bound on how many times the loop may be repeated. Unfortunately, **C** uses **FOR** as its only loop construct and it allows unbounded looping. So for **C** we would have to allow only bounded **FOR** loops. As you are aware, some languages use recursion instead of looping. For these languages we have to be careful because the recursion may allow non-termination. To avoid this, we can require that any recursion is on a natural number parameter which is always strictly decreasing. Then all the recursions will come to an end, and the programs will always halt.

Because we are placing a limitation on the form of allowed recursion, the recognizers written in these restricted programming languages are called **primitive recursive** recognizers, and the class of sets which are defined by such recognizers are called **primitive recursive** sets. For short, we will call this class of sets **Prim**, and note that **Prim** is a proper subclass of **Recursive**. That is, every primitive recursive set is recursive because the set has a recognizer, but there are some recursive sets whose recognizers are not primitive recursive recognizers. These recursive but not primitive recursive sets are somewhat *exotic* in that their recognizers can be represented in a general programming language, but their recognizers cannot be represented in a programming language with only bounded looping.

1.7 Inclusion Diagram

We finish this chapter with a diagram which indicates the inclusions we have demonstrated. The big triangles indicate the class **RE** and **coRE**. As we have shown and the diagram indicates these classes are distinct but they do have an intersection. This intersection, the diamond shaped area in the picture, is the class **Recursive**. (See Exercise 1.3.) Finally, the circle within the diamond is the class **Primitive Recursive**. The two outer bands with **-Complete** labels contain the "hardest" problems for their class. For example, the **HALT** set is in the band labeled **RE-Complete**.



The importance of this diagram is that we can prove each of the indicated inclusions. Further we can identify problems which appear in various areas of the diagram. And most importantly, this diagram with appropriate re-labeling seems to hold for resource restricted complexity classes. A similar diagram can be constructed and proved for the Context-Free Languages and related classes of languages. A similar diagram has been conjectured for $\mathcal{NP}$ problems and related classes of problems. In the $\mathcal{NP}$ case, none of the inclusion has been proved, but the analogy with diagrams for other classes leads most computer scientists to suspect that the diagram for $\mathcal{NP}$ is similar to the diagrams for other classes.

1.8 Oracle Turing Machines and a Hierarchy of Unsolvable Problems

When Americans have a problem which is too hard for them to solve, they can always call on Superman, a being with abilities far beyond those of mortal men. Most civilizations have stories about such super-humans. The ancient Greeks had the *oracles* who could answer questions which were too difficult for mortal men. To the chagrin of many Greeks the oracles often gave answers with several possible interpretations, so no matter how events transpired, the oracle was always right.

In the context of programs, we know that there are some problems, like the halting problem, which are too difficult to be solved by any program. What if we *imagine* an oracle, the halting-oracle, that can correctly solve the halting problem? If we allowed programs to make subroutine calls to this imagined halting-oracle, then these programs equipped with oracles could solve problems which ordinary programs could not solve.

Specifically, we could build a recognizer for **HALT** with such an oracle equipped program. Clearly we could also build a recognizer for **coHALT** by simply complementing the output of the recognizer for **HALT**. We can imagine a whole class of sets which have the oracle equipped recognizers. Let us say that $\mathbf{Recursive}^0$ is the class of sets which have recognizers *without* oracles, and let $\mathbf{Recursive}^1$ be the class of sets which have recognizers with **HALT**-oracles. Clearly $\mathbf{Recursive}^1$ strictly contains $\mathbf{Recursive}^0$. In fact, $\mathbf{Recursive}^1$ contains all of **RE** and all of **coRE** and may include some other sets outside of $\mathbf{RE} \cup \mathbf{coRE}$.

But, once we start with oracle-recognizers, we are almost forced to consider oracle-acceptors and oracle-rejectors. Then, similar to our definitions of $\mathbf{Recursive}^0$ and $\mathbf{Recursive}^1$, we can define $\mathbf{RE}^0$ and $\mathbf{coRE}^0$ as the classes which respectively have acceptors and rejectors without oracles, and define $\mathbf{RE}^1$ and $\mathbf{coRE}^1$ as the classes which respectively have acceptors and rejectors with **HALT**-oracles.

Let's now go back to Section 1.3 and look at the argument that **HALT** cannot have a recognizer. Exactly the same argument shows that the **HALT**-set for machines with **HALT**-oracles cannot have a oracle-recognizer. (We just have to consider the programs in the proof to be programs which have the **HALT**-oracle as a subroutine.)

It's only a small stretch of the imagination to suppose that there could be a stronger oracle which could solve the halting problem for programs which have **HALT**-oracle subroutines. We could call this a $\mathbf{HALT}^2$-oracle and define $\mathbf{Recursive}^2$ to be the class of sets which have recognizers with a $\mathbf{HALT}^2$-oracle as a subroutine. Similarly, we define $\mathbf{RE}^2$ as the class of sets which have acceptors with $\mathbf{HALT}^2$-oracle subroutines, and we define $\mathbf{coRE}^2$ as the class of sets which have rejectors with $\mathbf{HALT}^2$-oracle subroutines.

From here we can inductively continue to define a whole hierarchy of classes of sets, $\mathbf{Recursive}^K$, $\mathbf{RE}^K$, $\mathbf{coRE}^K$ for each natural number K, so that these classes have respectively recognizers, acceptors, or rejectors with $\mathbf{HALT}^K$-oracles. This hierarchy is sometimes called the **unsolvability hierarchy** because the classes above level 0 all contain problems which cannot be solved by ordinary programs without oracles. Further, level $K + 2$ contains problems which cannot be solved by programs with $\mathbf{HALT}^K$-oracles.

We will return to this hierarchy later and see that by analogy with this hierarchy, researchers have proposed that there are analogous hierarchies for resource bounded programs.

1.9 Computable Functions

In most of this chapter we have discussed sets and some of the senses in which sets are computable. For various purposes, we will also need computable functions. As for sets, the inputs will be natural numbers. For sets we limited the outputs to be in {**YES, NO**} and allowed for the possibility of no output because the computation does not halt. For functions we enlarge the possible outputs to be all of $\mathbb{N}$, but we still have the possibility of no output if a computation does not halt.

At this stage in our discussion, it seems worthwhile to define three types of computable function: the **partial recursive functions**, the (total) **recursive functions** , and the **primitive recursive functions**.

The partial recursive functions are the most general. Each partial recursive function is specified by a program which takes a natural number as an input. The program may halt with natural number as output or the program may fail to halt. A nice example of such a function is

$$f_{\text{HALT}}(PAIR(M,t)) \;=\; \begin{cases} PAIR(M,t)+1 & \text{if} \quad M \text{ halts given } t \\ \textbf{NOT HALT} & \text{if} \quad M \ \text{ does not halt given } t \end{cases}$$

or said another way

$$f_{\text{HALT}}(n) \;=\; \begin{cases} n+1 & \text{if} \quad PAIR_1^{-1}(n) \\ & \qquad \text{halts given } \ PAIR_2^{-1}(n) \\ \textbf{NOT HALT} & \text{if} \quad PAIR_1^{-1}(n) \\ & \qquad \text{does not halt given } \ PAIR_2^{-1}(n). \end{cases}$$

Of course, we would like to have functions which are always defined, that is, for every input the function gives an output. We say that a function is **recursive** (sometimes called **total recursive**) if there is a program which correctly computes all values of the function. That is, the function $f : \mathbb{N} \to \mathbb{N}$ is **recursive** if there is a program, say M_f, so that for each natural number n which is input, M_f eventually halts and outputs $f(n)$. Note that every recursive function is also a partial recursive function, but a special partial recursive function which has a value for each input.

Further, there are partial recursive functions which are necessarily partial, that is, there is no way to fill in the values in the **NOT HALT** cases to obtain a total recursive function. For example, if we tried to compute $f_{HA}(n)$ in the following manner

$$f_{HA}(n) \;=\; \begin{cases} f_{\text{HALT}}(n) & \text{if} \quad f_{\text{HALT}}(n) \ \text{has an output} \\ 0 & \text{if} \quad f_{\text{HALT}}(n) \ \text{ does not halt} \end{cases}$$

we would find that $f_{HA}(n)$ does **not** have a program which always gives the correct value. Because, if we had such a program, we could use it as a recognizer for the **HALT** set, and as we showed in Section 1.3 there is no such recognizer.

The total recursive functions can be rather *strange* and given a program we can not always determine if it computes a total recursive function. For these reasons, we define the primitive recursive functions as those total recursive functions which can be computed by a program with only bounded looping (see Section 1.6). While all primitive recursive functions can be computed by programs with unbounded looping, there are total recursive functions which can **only** be computed by programs with unbounded looping and cannot be computed by programs with bounded looping.

What do primitive recursive functions look like? They look like any reasonable function you can describe. For example, $f_1(n) = 127n^2 + 37$ or

$$f_2(n) = \gcd(PAIR_1^{-1}(n^n), PAIR_2^{-1}(n!)).$$

Some seemingly unreasonable functions are also primitive recursive as the following examples show:

$$L_1(n) = \begin{cases} n * L_1(n-1) & \text{if } n > 1 \\ 1 & \text{if } n \le 1 \end{cases}$$

$$L_2(n) = \begin{cases} L_1(L_2(n-1)) & \text{if } n \ge 1 \\ 5 & \text{if } n = 0 \end{cases}$$

$$L_3(n) = \begin{cases} L_2(L_3(n-1)) & \text{if } n \ge 1 \\ 5 & \text{if } n = 0 \end{cases}$$

$$L_4(n) = \begin{cases} L_3(L_4(n-1)) & \text{if } n \ge 1 \\ 5 & \text{if } n = 0 \end{cases} .$$

Try computing $L_4(5)$. Also look at the exercises 1.17 and 1.18.

1.10 Exercises

Ex 1.1. Find all the natural number solutions to

$$x + y = 6.$$

Show that there are an infinite number of integer solutions to this equation.

Ex 1.2. Show that

$$x^4 + y^2 = 6$$

has **NO** natural number solutions and has **NO** integer solutions.

Ex 1.3. Show that a set has a recognizer *iff* the set has both an acceptor and a rejector.

Ex 1.4. Show that

$$\textbf{recursive} = \textbf{RE} \cap \textbf{coRE}.$$

Ex 1.5. Show that almost all sets are;

(a) **NOT** recursive.

(b) **NOT** recursively enumerable.

(c) **NOT** in the oracle Turing machine hierarchy.

Ex 1.6. Show that S is recursive *iff* both S and $\overline{S}$ are recursively enumerable.

Ex 1.7. To see where the name *recursively enumerable* comes from, define an **enumeration** as an onto function, E, where $E : \mathbb{N} \rightarrow S$. That is, for each n, $E(n)$ is an element of S, and for each $x \in S$, there is an n so that $E(n) = x$. To make this computational we assume that there is an algorithm which computes E, that is, when the program for E is given a natural number as input, the program eventually halts having output the value $E(n)$. Since by Church's Thesis every computable function is a recursive function, we call this computable enumeration, a recursive enumeration, and say that S is *recursively enumerable.*
Show that our definition of **RE** as those sets with acceptors and the definition of recursively enumerable sets define the same class of sets.
That is, show that if a set has an acceptor then it has a recursive enumerator, and show that if a set has a recursive enumerator then it has an acceptor.

Ex 1.8. There is a minor difficulty in with the previous problem. Consider the empty set. Does it have a recursive enumerator? For this exercise, restate the definitions of the class **RE**

(a) in terms of acceptors

(b) in terms of recursive enumerators

so that the empty set and all finite sets are in **RE**.

Ex 1.9. Show that $\mathbb{N} \times \mathbb{N}$ is the same size as $\mathbb{N}$, by giving an easy to compute function $PAIR(i, j)$ which maps $\mathbb{N} \times \mathbb{N}$ one-to-one onto $\mathbb{N}$.

Ex 1.10. Show that $PAIR_1^{-1}(n)$ and $PAIR_2^{-1}(n)$ are easy to compute where

$$PAIR(\, PAIR_1^{-1}(n),\, PAIR_2^{-1}(n)\,) \;=\; n$$

and $PAIR(i, j)$ is the function you created in Exercise 1.9.

Ex 1.11. Use you pairing function to show that: for each positive integer K, $\mathbb{N}^K$ is the same size as $\mathbb{N}$. Here, $\mathbb{N}^K$ is the set of K-tuples of natural numbers.

Ex 1.12. Extend your argument from Exercise 1.11 to show that $\mathbb{N}^*$ is the same size as $\mathbb{N}$. Here, $\mathbb{N}^*$ is the set of finite tuples of natural numbers. Unlike $\mathbb{N}^K$ the number of elements in a tuple in $\mathbb{N}^*$ is **NOT** fixed.

Ex 1.13. We can ask for a more restrictive idea of recursive enumeration. We would like our function E to be both one-to-one and onto. That is, we want each element of S to be enumerated exactly once. Let's call such a recursive enumeration a *strong enumeration.*

(a) What class of sets have strong enumerators?

(b) Re-state the definitions of **RE** in terms of strong enumerators.

Ex 1.14. Let $S \subseteq \mathbb{N}$. Let $E : \mathbb{N} \rightarrow S$ be a recursive function that is one-to-one and onto S.

(a) Show that E is an invertible function and that $E^{-1} : S \rightarrow \mathbb{N}$ is a recursive function.

(b) Is $E : \mathbb{N} \rightarrow \mathbb{N}$ an invertible function?

(c) Define

$$E_2(n) = \begin{cases} E(n/2) & \text{if } n \text{ is even} \\ \text{the least } m & \text{if } n \text{ is odd} \\ \text{so that } m > E_2(n-2) & \\ \text{and } m \notin S. & \end{cases}$$

Show that $E_2(n)$ is an invertible function

(d) Discuss whether $E_2(n)$ or $E_2^{-1}(n)$ are or are not recursive functions.

Ex 1.15. Show that

$$\mathbf{Recursive}^K = \mathbf{RE}^K \cap \mathbf{coRE}^K.$$

Ex 1.16. Show that

$$\mathbf{Recursive}^{K+1} \supseteq \mathbf{RE}^K \cup \mathbf{coRE}^K.$$

Ex 1.17. Show that for any primitive recursive function, there is a primitive recursive function which grows much more quickly. I.E., given a primitive recursive function $f(n)$ show how to compute $g(n)$ with bounded looping or bounded recursion so that $g(n) > f(n)$ for all n and

$$\lim_{n\rightarrow\infty} \frac{f(n)}{g(n)} = 0.$$

Ex 1.18. Show that there is a recursive function which grows more quickly than each and every primitive recursive function.