Problem Solving, Programming, Algorithms

Computer scientists are supposed to be able to program solutions to problems. These solutions are, whenever possible, required to be algorithms. Algorithm must

- Take finite inputs, if at all inputs are needed. We must not ask for an infinite amount of information before a finite result is given. Computer scientists use the notion of continuity.
- Produce finite outputs.
- Algorithms must be expressible in an effective manner—i.e., comprised of mechanical steps. It must not invoke “answer oracles.”
- Algorithms must be definite or deterministic, e.g., not rely on background radiation from the Big Bang to compute something.
- Algorithms must be finite, i.e., must terminate.

Algorithms that do everything except being finite (algorithms that may not terminate) are (must be) called procedures.

When can we say that an algorithm may not exist?

Consider the PCP instance

\[
\begin{array}{c}
4 & 3 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{array}
\]

Can we write a computer algorithm to solve all such unary PCP instances? Why? Can you list some solutions now?

Consider the PCP instance

\[
\begin{array}{c}
4 & 3 \\
1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1
\end{array}
\]

Here is a solution

Instance 1:

\[
\begin{array}{c}
4 & 3 & 10 & 1 & 0.005301 \\
1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1
\end{array}
\]

reverse of

\[
\begin{array}{c}
1 & 3 & 1 & 4 & 2 & 1 & 2 & 2 & 4 & 2 \\
2 & 4 & 2 & 2 & 1 & 2 & 4 & 1 & 3 & 1
\end{array}
\]

i.e.

\[
\begin{array}{c}
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1
\end{array}
\]

i.e.

\[
\begin{array}{c}
010010101101001101 \\
0100101011001101
\end{array}
\]

Similarly can we solve Diophantine equations always?

See a “solver” at

http://www.alpertron.com.ar/QUAD.HTM

For Coeffs 4, 27, 4, -7, 4, 3, one solution was 4027, -610 !!
• Any procedure we device for these (and thousands of other problems—for instance, algorithmically checking if a CFG is ambiguous) don’t seem to exploit the structure of the problem, and be able to “recurse into the problem.”
• Without terminating recursion or loops that terminate, we can’t argue that procedures are indeed algorithms.
• The solutions (“certificates”) are not bounded in size in any discernible way!
• These are symptoms that after all we may only have a procedure and not an algorithm.

Then what do we do?

Then we try to prove that something does not have an algorithm by reduction from an unsolvable problem.

Favorites:
• $A_{TM}$ reduced to suspect
• PCP reduced to suspect (PCP was shown algorithmically unsolvable by reduction from $A_{TM}$)

Need to understand the structure of languages; how many are there?

• The only way we can count infinite sets is through “barter” i.e., one-to-one onto (bijective) correspondence with sets of known cardinality
• There are ($\aleph_0$) Natural numbers
• The Schröder-Bernstein (“Mirror in front of mirror”) theorem allows us to claim a bijection with a pair of injections going each way. Example: there are as many C programs as there are Natural numbers:
  Injection from Nat to C
  0 → main();
  1 → main();{};
  2 → main();{};;;
  Injection from C to Nat
  For each C program, the concatenation of the ASCII sequence read as a single binary number is one such injection
• There are as many real numbers in $[0, 1)$ as there are in $[0, \infty)$
• There are real-number number of languages ($\aleph_1$)—a simple power-set argument.
• ($\aleph_0 \neq \aleph_1$ by diagonalization)
• There are only $\aleph_0$ RE languages
• There are $\aleph_1$ languages
• There are languages whose structures cannot be captured using any machine at all!

The language of a TM, RE languages, Recursive languages

I am going to present facts for you to checkoff as you follow each item. TM stands for Turing machine. $L(M)$ for a TM $M$ stands for the language of $M$.

1. □ For a given TM $M$, $w \in L(M)$ exactly when $M$ when started with $w$ on its tape halts in state $h_a$ (“halt with accept” state).
   • $M$ “goes to work” on $w$ (i.e., chugs along)
   • May or may not read $w$ fully (may even completely ignore $w$!!)
   • Eventually, $M$ may stop in state $h_a$ (“halt with accept” state). If/when it does so, then $w$ is deemed to have been accepted.
   • However, any TM that ignores $w$ has innate behavior built into it that
– either always makes it halt
– or makes it loop
In the former case, M’s language is said to be universal (is $\Sigma^*$) while in the latter case, it is of course empty (is $\emptyset$).

2. □ Therefore any TM $M$ whose language is neither universal nor empty must read its input at least partially, and base its actions on what it read.

3. □ A TM is said to accept a string. It is said to recognize its language.

4. □ A Recursively Enumerable Set (RE set) $S$ or an RE language $L$: An RE set—or an RE language—mean one and the same thing. It is a language $L$ such that it is also the language of some TM $M$.
   - Any TM will do; so long as one TM $M$ has $L$ as its language (exactly $L$; no omissions of strings from $L$; no strings beyond what $L$ contains), you can call $L$ an RE set.

5. □ Question: Which of these are RE languages?
   - The empty language
   - The universal language
   - The language $\{0^n1^n \mid n \geq 0\}$
   - The language of pairs $(M, w)$ such that $M$ is a string completely describing a TM $M$ (like the program text of $M$; think of a C, Java, or C# program as $M$, if you wish), and $w$ is a string on which $M$ operates, such that $M$ accepts $w$.
     That is, when $M$ is run on $w$, it will halt in state $h_a$.
   - The above language is called $A_{TM}$. It is defined with respect to a TM $M$, of course. This $M$ is implicit.

   Answers:
   - The empty language is RE
   - The universal language is RE
   - The language $\{0^n1^n \mid n \geq 0\}$ is RE
   - $A_{TM}$ is RE. A TM $U$ can be designed to accept $M$ and $w$. TM $U^1$ simply runs $M$ on $w$, serving as an interpreter for $M$. When $M$ halts on $h_a$, $U$ in turn goes and halts in its own state $h_a$. Ha!

6. □ A Recursive Set $S$ or a recursive language $L$: A recursive language $L$ is one where $L$ and $\overline{L}$ are RE.

7. □ A Recursive Set $S$ or a recursive language $L$: It is also the language of a TM that halts on every input.

8. □ How can we show that the above two are equivalent definitions?
   - Clearly for an RE set to be non-trivial (non-universal or non-empty), their TMs must read at least some of the input.
   - If a TM ignores a given $w$ and jumps to $h_a$, it accepts ANYTHING!
   - Suppose we are given some $M$ and $w$ such that $M$ does not accept $w$. When $U$ runs $M$ on $w$, we might find $M$ halting in $h_r$; at that time, $U$ can also halt in its $h_r$ state. However, $M$ might loop on $w$. In this case, $U$ also loops.

---

0 Called a “universal TM”—would be called TM $Y$ at BYU.
0 An interpreter is not always someone who translates Chinese into Russian.
• We just proved that $A_{TM}$ is RE (using $U$). We are yet to prove that $A_{TM}$ is not recursive. We will do that much later through a wonderful procedure called diagonalization.

• Intuitively though, $A_{TM}$ cannot be recursive, because it then proves our ability to enumerate what is in $A_{TM}$ (all those $\langle M, w \rangle$ pairs where $M$ accepts $w$).

Which are RE and which are Recursive?

• The set of all legal C programs
• The set of all legal SS numbers
• The sequence of all legal Chess games (encoded as strings) ending in a Checkmate
• The set of all CFGs whose language is universal
• The set of all TMs $M$ (TM descriptions $\langle M \rangle$) such that $M$ halts (accepts or rejects) string 0101.
• $L_{\text{loop}}$, the set of all TMs $M$ (TM descriptions $\langle M \rangle$) such that $M$ loops on string 0101.
• Complement of $L_{\text{loop}}$
• The set of all TMs $M$ that halts on input 0101 in 10 steps.

A Printer TM and RE sets

A printer TM is one that prints symbols on a printer tape. One can design a TM that keeps outputting anything on its printer tape. This idea was historically how RE was defined. A set $S$ is RE iff there is a printer TM that outputs precisely $S$.

Proof: The proof has two parts.

(1) Given a printer TM $M$ that outputs precisely $S$, show that there is a regular TM $N_M$ that has $S$ as its language.

$N_M$ given $x$ runs $M$. If/when $M$ prints $x$, $N_M$ accepts $x$. This way, $N_M$ serves as the TM for $S$. Therefore $S$ is RE in the standard way.

(2) Given a regular TM $N$, we build a printer TM $M_N$. $M_N$ keeps generating inputs from $\Sigma^*$. It runs one step of $N$ on all the strings generated so far. It alternates string generation and running one step in this manner. If and when $N$ accepts an $x$, $M$ outputs $x$ on its printer tape.