The NP vs P Problem

the second of two talks – why it is hard

Appendix A

by

Charles W. Neville, October 2000

1. The Halting Problem TwistedPprogram t

The twisted program t in our proof of the undecideability of the Halting Problem is actually quite simple,

```
program t;
    procedure s ( string p );
        begin {* s *}
            the body of s goes here
        end; {* s *}
    begin {* main *}
        on input p,
            if s outputs "runs forever" then stop
            else while true do; {* infinite loop *}
    end. {* main *}
```

Here we have chosen a Pascal like programming language. Recall from the talk that we posited the existence of the super program s which, on input of the text of a given program p, successfully predicts whether p, on itself as input, halts or runs forever, and recall that t is supposed to behave in exactly the opposite manner from the predictions of s. It is clear that t behaves as advertised.

Before going further, it might be a good idea to give a real world example of a program which takes the text of other programs as input:

A Pascal compiler. Such a program takes the text of Pascal programs as input and produces machine language translations. A Pascal compiler could easily have its own text as input. It would then produce a machine language translation of itself.

2. The Exponential Diagonalization Twisted Program t

The twisted program t in our proof that there are programs which run in exponential time but not in polynomial time is somewhat like the Halting Problem twisted program above. The main differences are,

a. In place of s, t has a built in Pascal interpreter which it calls on input of a program p to run p.
b. The interpreter has a built in counter, and the interpreter uses this counter to halt after e^n steps, where n is the number of bits in the text of p, in case the program p has not already halted.

Point (a) allows our twisted program t to run Pascal programs. Point (b) insures that t runs in exponential time. Recall that both of these are necessary in the diagonalization proof that there are programs which run in exponential time, but not in polynomial time.

There is an apparent snag in this. Although e^n is eventually larger than Mn^q for any constant M and any exponent q, n may have to be quite large before this happens. However, this snag is only apparent, because the text of legal programs p in $\mathcal{P}$ may begin with any number of leading blanks, without changing the way the program runs. Thus if the number of bits n_1 in the the text of program p_1 in $\mathcal{P}$ is too small for p_1 to finish in time e^{n_1}, simply note that there will be an equivalent program p_2, consisting of the text of program p_1 padded with enough leading blanks so n_2 will be so large that p_2 will be able to finish in time e^{n_2}. Here n_2 is the number of bits in the text of p_2.

You should check that the diagonalization argument still goes through because t still disagrees with the yes/no output of every program p in $\mathcal{P}$ on the text of an equivalent program as input.

3. Turing Machine vs RAM Machine Time Complexity

The theorem in the first talk on the quadratic time slowdown for Turing machines vs RAM machines needs more explanation. First, if a RAM machine is allowed to fetch arbitrarily long words at once, then a quadratic slowdown no longer holds, so we need to assume a RAM machine can only fetch or store a fixed number of bits, say 128, at once. (However, it should still deal with arbitrarily long addresses in one operation.) Second, the quadratic slowdown theorem holds true for multi-tape Turing machines. For single tape Turing machines, there is a cubic slowdown (because of integer multiplication and division). Refer to the book of Aho, Hopcroft and Ullman, or the book of Hopcroft and Ullman, both listed in the talk references, for more information.

4. Computable Sequences

In the talk, we asserted without proof that most sequences of 0's and 1's are not computable. The proof is quite simple. An (infinite) sequence of 0's and 1's is computable by a program p if p, on input n, outputs the n^{th} term of the sequence. A sequence is computable if it is computable by some program. Thus the sequence of binary digits of π is computable. On the other hand, there are uncountably many uncomputable sequences, and thus most sequences are not computable.

Here is the simple proof. The Cantor second diagonal argument shows there are uncountably many (infinite) sequences of 0's and 1's. But there are only countably many programs because the text of each program is a finite sequence of ASCII characters, and there are only countably many such finite sequences. Since each computable sequence must be computable by a program, and each program computes at most one sequence, there are only countably many computable sequences. Thus the set of uncountable sequences is the complement of a countable set in an uncountable set, and so is uncountable.