[Overall Information]

In today’s class, we continued our study of undecidable problems. We proved that the halting problem is undecidable, and showed how to use reductions to prove other problems undecidable. Then, we learned the Recursion Theorem, and used it to prove that there is a program printing its own source code.

[Announcements]

1. The last homework set, homework#6, was assigned in today’s class. Printed handouts of homework questions were distributed after the class. A list of questions can also be found on the class website. The corresponding solutions will be provided in next class.

2. The quiz#6 is scheduled on 04/26/2006.

3. A programming contest will take place this Saturday (04/22/2006) from 1:00 to 6:00pm. Interested students should send email to the professor to register.

4. The material of recent lectures is not in the textbook, so please read the class notes on the course website.

5. Students interested in these topics are encouraged to read the
supplementary reading material: “Introduction to the Theory of Computation” by Michael Sipser, as well as other texts listed on the homework handout.

[During the Lecture]

1. The halting problem is to decide whether a program P on input x will terminate or not. More specifically, the language of all pairs (P,x) such that program P on input x terminates is called the “General Halting
Problem”. To avoid writing such long sentences all the time, we use the following notation:

(1) $P(x)\downarrow$ means that program $P$ on input $x$ terminates;

(2) $P(x)\uparrow$ means that program $P$ on input $x$ does not terminate.

The theorem we are going to prove is that the halting problem can not be decided by any algorithm no matter how much memory space or time it gets to use.

The theorem was proved by contradiction. We assume that some program $H$ is able to decide, for all programs $P$ and inputs $x$, whether $P(x)\downarrow$. Then we design a new program using $H$ as a subroutine. At the beginning of the new program, after reading in the input program $P$, it calls $H$ to tell whether $P$ terminates on input $P$ or not (that is, it finds out what $P$ would do if it were given its own source code as input). If the answer is yes (it does terminate), then our new program deliberately enters an infinite loop. If the answer is no ($P$ would terminate on input $P$), our new program outputs 1 and terminates.
2. This is a legal program: if the subroutine \( H \) exists, then we are only adding one comparison and an infinite loop, both of which are easy to program. Let \( \Phi \) be the source code of the entire program we just generated.

Next, we ask ourselves what would happen if \( \Phi \) got its own source code as input:

(1) If \( \Phi \) did terminate on its own source code as input, then the subroutine would answer “Yes”, so \( \Phi \) would enter an infinite loop and would not terminate.

(2) If it did not terminate on its own source code as input, then the subroutine would answer “No”, so \( \Phi \) would output 1 and terminate.

Both cases are obviously impossible.
So the only conclusion this allows is that no such program $\Phi$ can exist.

But since we know that adding the comparison and infinite loop is always possible, the reason must be that the subroutine $H$ does not exist. This proves that no program can solve the Halting Problem.

In fact, we have proved the slightly stronger statement that no program/algorithm can decide $\text{HALT} := \{ P \mid P(P) \downarrow \}$. This $\text{HALT}$ problem can be viewed as a special case of general halting problem. So the general halting problem is also undecidable. The special version $\text{HALT}$ we defined here will turn out to be a little easier to reduce from.

Notice that the halting problem is undecidable only in the case of infinite memory. Otherwise, we can just keep track of the memory status in order
to see if the program runs into the same memory state, and thus a cycle. If not, then it must terminate in $2^m$ steps (where $m$ is the total number of memory bits). While it is true that the halting problem is decidable for any finite memory (and undecidable for infinite memory), any memory size of a few thousand bits is infinite for our purposes, as keeping track of $2^{256000000}$ states is absolutely impossible.

4. Next, we wanted to use reductions from the halting problem to prove other problems undecidable. As an example, we defined the following problem $42 := \{P \mid P$ outputs 42 for at least one input}. We will prove that “42” is also an undecidable problem using a reduction from the
HALT problem.

The notion of a reduction is almost the same as what we used in NP-hardness proofs: a function \( f \) that maps “Yes” instances to “Yes” instances, and “No” instances to “No” instances. The difference is that the function \( f \) need not be computable in polynomial time here (we don't care about running time); all we require is that \( f \) be computable by some algorithm. Such reductions are also called “many-to-one” reductions (to distinguish them from one-to-one reductions where no two instances of the first problem can be mapped to the same instance of the second; for our purposes in this class, this is not an important distinction, but the notation \( \leq_m \) is fairly standard).

In practice \( m \) is finite but large (256,000,000 these days), and no computer could keep track of 256,000,000 states.

\[ \begin{align*}
\text{Claim: } \text{Halting is also undecidable.} \\
\text{Idea: } & \text{Use a reduction - show that if Halting is decidable, so is HALT.} \\
\text{Notion of reduction here: Function } f \text{ that always terminates and takes input } x \text{ to } x', \text{ transforms input } y=f(x) \text{ to } y'.
\end{align*} \]
Using this notion of reduction, we are ready to prove $\text{HALT} \leq_m \text{“}42\text{”}$. We need to define a “program rewrite function” $f$ that performs the reduction. It takes a program $P$ as input, and produces a new program $P'=f(P)$ which does the following: it first runs $P$ on its own source code as input. If and when that terminates, $P'$ outputs 42. Of course, if $P(P)\uparrow$, then $P'$ will also not terminate, and will never output 42.

Notice the difference between this program rewrite function and the above program $\Phi$ when we proved the undecidability of the halting problem.
problem. In defining \( \Phi \), we assumed that we had a subroutine \( H \) that would tell us if \( P \) would terminate on input \( P \), and allowed us to take corresponding actions in both cases. Here, the new program \( P' \) calls \( P(P) \) first, and if \( P \) never terminates, then \( P' \) cannot do anything else after running \( P \). If \( P \) does terminate, then \( P' \) does get to perform something else. So the program \( P' \) here is very easy to write when given \( P \), but \( \Phi \) would have required the impossible halting problem solver.

As for the correctness proof, if \( P \) is a “yes” instance of HALT, \( P' \) will always output 42. Otherwise, \( P' \) will neither terminate, and thus also never output 42. Notice that \( P' \) completely ignores its input. It is one of
two really easy programs. Either it is the program \{\texttt{printf ("42");}\}, or it is the program \{\texttt{while (1==1) do;}\} The difficulty is in deciding which of those two programs it is, because the programs are written in a very very obfuscated way, by first calling P.

To summarize the reduction,

1. If P is a “yes” instance, P’ always outputs 42, so P’ is a “yes” instance;
2. If P is a “no” instance, P’ never outputs 42, so P’ is a “no” instance.

This completes the correctness proof.

8. Next, we introduced a few standard key definitions of recursive function theory. A function f is said to be \textbf{recursive} if it is computed by some program (the terminology dates back to pre-computer areas, when mathematicians were trying to characterize what functions would be humanly computable, and used the concept of recursively defined functions. It turns out that those are exactly the same functions computable by programs). We say a function is \textbf{total} if f(x) ↓ for all x. A set S is \textbf{decidable} if its characteristic function “f(x)=1 if x∈S, f(x)=0 otherwise” is recursive, i.e., if there is a program always terminating, and deciding correctly if the given element x is in S.

A set S is \textbf{recursive enumerable} (r.e) if the function “f(x)=1 if x∈S, f(x) ↑ otherwise” is recursive. That is, there is a function which will correctly
conclude that \( x \) is in the set if it is, and otherwise will never terminate. (The name enumerable comes from the fact that one can use such a function as a subroutine to write a program that will eventually output or enumerate the entire set.)

9. While the halting problem \( \text{HALT} \) is not decidable, it is easily seen to be r.e., as the function can simply simulate \( \text{P}(\text{P}) \). If it terminates, output 1, and otherwise, the subroutine does not terminate either.

10. Finally, we introduced Kleene's Recursion Theorem. One of the interesting consequences of the Recursion Theorem is that there is a
program that prints its own source code. The recursion theorem is the following: “For every total recursive function \( f \), there exists an index \( i \) such that \( P_i \equiv P_{f(i)} \)” (where the \( \equiv \) sign means that the two programs terminate on exactly the same inputs, and produce the same output on all inputs on which they terminate). Intuitively, we can think of \( f \) as a program rewriting function. It takes some program, and performs whatever operations it wants on it (adding commands, removing commands, changing values, completely garbling the code, etc.) If we apply this to program number \( i \), we get a new program (which might only produce garbage, or never terminate) with number \( f(i) \). Kleene's Recursion Theorem then states that for each such program rewrite function, there is some program that computes the same function before and after the rewrite. Thus, while the source code of that program may be changed, its behavior is not changed by \( f \).
9. From Kleene's Recursion Theorem, we can deduce the following corollary: there is a program P such that P(x)=P. That is, there is a program that outputs its own source code (and ignores the input).

To prove this theorem, we define a rewrite function f: f(P) := \{output the source codes of P\}. That is, given a program P, the new program does not actually run or simulate P, but rather outputs the source code of P, and doesn't do anything else. Since f is recursive and total (it transforms each program to some new program), we can apply the Recursion Theorem. It promises that there is a “fixed point” P with P≡f(P). But we know what f(P) does: it outputs the source code of P. And because P does exactly the same thing, P too must output the source code of P. So P outputs its own source code.

It may amuse you to try and actually write a program that outputs its own source code. It's not that easy, and takes at least a few dozen lines of code. Of course, we are not allowing disk or memory access to read off the file.