بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۰۱)

نظریهٔ زبان‌ها و ماشین‌ها

حسین فلسفین

Informally speaking, an algorithm is a collection of simple instructions for carrying out some task. Commonplace in everyday life, algorithms sometimes are called procedures or recipes.

Even though algorithms have had a long history in mathematics, the notion of algorithm itself was not defined precisely until the twentieth century. Before that, mathematicians had an intuitive notion of what algorithms were, and relied upon that notion when using and describing them. But that intuitive notion was insufficient for gaining a deeper understanding of algorithms.

Hilbert's Problems

In 1900, mathematician David Hilbert delivered a now-famous address at the International Congress of Mathematicians in Paris. In his lecture, he identified 23 mathematical problems and posed them as a challenge for the coming century. ***The tenth problem on his list concerned algorithms.***

یادآوری مفهوم چندجمله‌ای

A polynomial is a sum of terms, where each term is a product of certain variables and a constant, called a coefficient. For example,

$$6 \cdot x \cdot x \cdot x \cdot y \cdot z \cdot z = 6x^3yz^2$$

is a term with coefficient 6*, and*

$$6x^3yz^2 + 3xy^2 - x^3 - 10$$

is a polynomial with four terms, over the variables x*,* y*, and* z*. For this discussion, we consider only coefficients that are integers. A root of a polynomial is an assignment of values to its variables so that the value of the polynomial is* 0*. This polynomial has a root at* $x = 5$*,* $y = 3$*, and* $z = 0$*. This root is an integral root because all the variables are assigned integer values. Some polynomials have an integral root and some do not.*

مسئلهٔ دهم هیلبرت

Hilbert's tenth problem was to devise an algorithm that tests whether a polynomial has an integral root. He did not use the term algorithm but rather "a process according to which it can be determined by a finite number of operations." Interestingly, in the way he phrased this problem, Hilbert explicitly asked that an algorithm be "devised." Thus he apparently assumed that such an algorithm must exist—someone need only find it. As we now know, no algorithm exists for this task; it is algorithmically unsolvable.

توضیحی مختصر دربارهٔ تصمیم‌پذیری یک مسئلهٔ ساده‌تر

Let

$$D = \{p | p \text{ is a polynomial with an integral root}\}.$$

Hilbert's tenth problem asks in essence whether the set D is decidable. The answer is negative. In contrast, we can show that D is Turing-recognizable. Before doing so, let's consider a simpler problem. It is an analog of Hilbert's tenth problem for polynomials that have only a single variable, such as $4x^3 - 2x^2 + x - 7$. Let

$$D_1 = \{p | p \text{ is a polynomial over } x \text{ with an integral root}\}.$$

Here is a TM M_1 that recognizes D_1:

M_1 = "On input $\langle p \rangle$: where p is a polynomial over the variable x.
1. Evaluate p with x set successively to the values $0, 1, -1, 2, -2, 3, -3, \ldots$. If at any point the polynomial evaluates to 0, accept."

If p has an integral root, M_1 eventually will find it and accept. If p does not have an integral root, M_1 will run forever. For the multi-variable case, we can present a similar TM M that recognizes D. Here, M goes through all possible settings of its variables to integral values. Both M_1 and M are recognizers but not deciders. We can convert M_1 to be a decider for D_1 because we can calculate bounds within which the roots of a single variable polynomial must lie and restrict the search to these bounds.

In Problem 3.21 you are asked to show that the roots of such a polynomial must lie between the values $\pm k\frac{c_{\max}}{c_1}$, where k is the number of terms in the polynomial, $c_{\max}$ is the coefficient with the largest absolute value, and c_1 is the coefficient of the highest order term. If a root is not found within these bounds, the machine rejects. Matijasevich's theorem shows that calculating such bounds for multivariable polynomials is impossible.

3.21 Let $c_1x^n + c_2x^{n-1} + \cdots + c_nx + c_{n+1}$ be a polynomial with a root at $x = x_0$. Let $c_{\max}$ be the largest absolute value of a c_i. Show that

$$|x_0| < (n+1)\frac{c_{\max}}{|c_1|}.$$

چرا ارائهٔ یک تعریف دقیق برای مفهوم الگوریتم، ضرورت دارد؟

We have an intuitive notion of an algorithm which says that it is a sequence of simple instructions for carrying out a computational task. The intuitive notion tells us that an algorithm must be described unambiguously and hence be able to be executed mechanically to yield an output. So the intuitive notion is adequate when we are required to design an algorithm to compute a certain problem. We are always sure about whether a procedure designed to compute a given problem could be an algorithm or not. In contrast, when we are required to prove that there exists no algorithm to compute a certain problem, the intuitive notion of algorithms is not adequate. In such a situation we need to have a precise definition of an algorithm. Informally speaking, this is because, if we are able to prove that a certain problem cannot be computed by any algorithm, that argument necessarily somehow refers to all of what is called an algorithm, which is impossible unless we have a precise definition of what an algorithm is.

ارائهٔ تعریفی دقیق برای مفهوم الگوریتم توسط چرچ و تورینگ

For mathematicians of that period to come to this conclusion with their intuitive concept of algorithm would have been virtually impossible. The intuitive concept may have been adequate for giving algorithms for certain tasks, but it was useless for showing that no algorithm exists for a particular task. Proving that an algorithm does not exist requires having a clear definition of algorithm. Progress on the tenth problem had to wait for that definition. The definition came in the 1936 papers of Alonzo Church and Alan Turing. Church used a notational system called the λ-calculus to define algorithms. Turing did it with his "machines." These two definitions were shown to be equivalent.

بد نیست نگاهی هم به تاریخچهٔ ***Entscheidungsproblem*** و روندی که برای اثبات تصمیم‌ناپذیری‌اش طی شد بیاندازید.

This connection between the informal notion of algorithm and the precise definition has come to be called the Church–Turing thesis. The Church–Turing thesis provides the definition of algorithm necessary to resolve Hilbert's tenth problem. In 1970, Yuri Matijasevich, building on the work of Martin Davis, Hilary Putnam, and Julia Robinson, showed that no algorithm exists for testing whether a polynomial has integral roots.

Intuitive notion of algorithms	equals	*Turing machine algorithms*

☞ *Computation = expressible on a Turing machine*

☞ *Algorithm = expressible on a Turing machine that halts (a decider)*

نکات مهم و درخور ذکر دربارهٔ تز چرچ تورینگ

☞ *The Church-Turing thesis, proposed by logician Alonzo Church in 1936, asserts that any effective computation in any algorithmic system can be accomplished using a Turing machine.*

☞ *The Church-Turing thesis provides a general principle for algorithmic computation and, while not provable, gives strong evidence that no more powerful models can be found.*

☞ *With the acceptance of the Church-Turing thesis, it becomes obvious that the extent of algorithmic problem solving can be identified with the capabilities of Turing machines. Consequently, to prove a problem to be unsolvable it suffices to show that there is no Turing machine that solves the problem.*

☞ *What is an algorithm? Church used lambda calculus, Turing used Turing machines (not his name for them). It was fairly easily shown that these two characterizations of algorithms (Church's and Turing's) are equivalent.*

☞ *To say that the TM is a general model of computation means that any algorithmic procedure that can be carried out at all, by a human computer or a team of humans or an electronic computer, can be carried out by a TM. It is not a mathematically precise statement that can be proved, because we do not have a precise definition of the term algorithmic procedure. By now, however, there is enough evidence for the thesis to have been generally accepted.*

☞ *The CTT states that TMs can carry out algorithms done by any model of computation whatsoever, certainly including standard computer operations. It is in principle possible that someone may come up with a more powerful model. Many very different models have been proposed over the years. All have been shown to be no more powerful than the TM.*

☞ *We have carefully used the word thesis here, rather than theorem. There exists no proof of the CTT because its statement depends on our informal definition of a computational algorithm.*

☞ *It is impossible to prove this claim (celebrated conjecture) because we do not have any precise definition of algorithms. In other words, we cannot provide convincing arguments to deny the possibility that there exists a procedure that we could call an algorithm, but cannot be described in terms of any Turing machine. The CTT proposes to take this claim for granted. The CTT is not what we can prove, but is what we admit.*

☞ *The Turing thesis is more properly viewed as a definition of what constitutes a mechanical computation: A computation is mechanical if and only if it can be performed by some Turing machine. If we take this attitude and regard the Turing thesis simply as a definition, we raise the question as to whether this definition is sufficiently broad. Is it far-reaching enough to cover everything we now do (and conceivably might do in the future) with computers? An unequivocal "yes" is not possible, but the evidence in its favor is very strong.*

شواهدی که بر صحت تز چرچ تورینگ دلالت دارند

Some arguments for accepting the Turing thesis as the definition of a mechanical computation are

1. Anything that can be done on any existing digital computer can also be done by a Turing machine.

2. No one has yet been able to suggest a problem, solvable by what we intuitively consider an algorithm, for which a Turing machine program cannot be written.

☞ *Since the introduction of the Turing machine, no one has suggested any type of computation that ought to be included in the category of "algorithmic procedure" and cannot be implemented on a TM.*

☞ *There has been no algorithm that cannot be described in principle in terms of any Turing machine. (Furthermore, after we have had enough experience to write Turing machines to solve many concrete problems, we come to have a feeling that any algorithm*

can be described in terms of a Turing machine.)

☞ *We might try to find some procedure for which we can write a computer program, but for which we can show that no Turing machine can exist. If this were possible, we would have a basis for rejecting the hypothesis. But no one has yet been able to produce a counterexample; the fact that all such tries have been unsuccessful must be taken as circumstantial evidence that it cannot be done.*

3. Alternative models have been proposed for mechanical computation, but none of them is more powerful than the Turing machine model.

☞ *At various times, other models have been proposed, some of which at first glance seemed to be radically different from Turing machines. Eventually, however, all the models were found to be equivalent.*

☞ *Several mathematical models have been proposed independently to define algorithms, such as the λ-calculus and recursive func-*

tions, which have been proved to be equivalent with each other in power to the Turing machines. The fact that several mathematical models proposed independently are equivalent with each other reveals that the models are natural, and robust in the sense that variance in the model keeps invariance in power.

The Church–Turing thesis states that every model of computation—even those not yet in existence—can be done by, and is equivalent to, a Turing machine. Note that there is no hope of proving this. It is not possible to prove something about a model of computation that has not yet been proposed. However, the thesis has been verified for all known models of computation. It is widely accepted in the communities of logic, mathematics, and computer science. But its unprovability is why it has the name thesis instead of theorem.

دربارهٔ شباهت تز چرچ تورینگ با قوانین حرکت نیوتن

These arguments are circumstantial, and Turing's thesis cannot be proved by them. In spite of its plausibility, Turing's thesis is still an assumption. But viewing Turing's thesis simply as an arbitrary definition misses an important point. In some sense, Turing's thesis plays the same role in computer science as do the basis laws of physics and chemistry. Classical physics, for example, is based largely on Newton's laws of motion. Although we call them laws, they do not have logical necessity; rather, they are plausible models that explain much of the physical world. We accept them because the conclusions we draw from them agree with our experience and our observations. Such laws cannot be proved to be true, although they can possibly be invalidated. If an experimental result contradicts a conclusion based on the laws, we might begin to question their validity. On the other hand, repeated failure to invalidate a law strengthens our confidence in it. This is the situation for Tur-

ing's thesis, so we have some reason for considering it a basic law of computer science. The conclusions we draw from it agree with what we know about real computers, and so far, all attempts to invalidate it have failed.

There is always the possibility that someone will come up with another definition that will account for some subtle situations not covered by Turing machines but which still fall within the range of our intuitive notion of mechanical computation. In such an eventuality, some of our subsequent discussions would have to be modified significantly. However, the likelihood of this happening seems to be very small.

دربارهٔ سه سطح توصیف

The first is the formal description that spells out in full the Turing machine's states, transition function, and so on. It is the lowest, most detailed level of description. The second is a higher level of description, called the implementation description, in which we use English prose to describe the way that the Turing machine moves its head and the way that it stores data on its tape. At this level we do not give details of states or transition function. The third is the high-level description, wherein we use English prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head.

از اینجا به بعد بیشتر با توصیف سطح بالا (نوع سوم) سروکار داریم

☞ *We have come to a turning point in the study of the theory of computation. We continue to speak of Turing machines, but our real focus from now on is on algorithms. That is, the Turing machine merely serves as a precise model for the definition of algorithm. We skip over the extensive theory of Turing machines themselves and do not spend much time on the low-level programming of Turing machines. We need only to be comfortable enough with Turing machines to believe that they capture all algorithms.*

☞ *In fact, another way we will use the Church-Turing thesis in the rest of this course is to describe algorithms in general terms, without giving all the details of Turing machines that can execute them. A full-fledged application of the Church-Turing thesis allows us to stop with a precise description of an algorithm, without referring to a TM implementation at all.*

We begin to investigate the power of algorithms to solve problems

We begin to investigate the power of algorithms to solve problems. We demonstrate certain problems that can be solved algorithmically and others that cannot. Our objective is to explore the limits of algorithmic solvability.

You are probably familiar with solvability by algorithms because much of computer science is devoted to solving problems. The unsolvability of certain problems may come as a surprise.

Why should you study unsolvability?

After all, showing that a problem is unsolvable doesn't appear to be of any use if you have to solve it. You need to study this phenomenon for two reasons. First, knowing when a problem is algorithmically unsolvable is useful because then you realize that the problem must be simplified or altered before you can find an algorithmic solution. Like any tool, computers have capabilities and limitations that must be appreciated if they are to be used well. The second reason is cultural. Even if you deal with problems that clearly are solvable, a glimpse of the unsolvable can stimulate your imagination and help you gain an important perspective on computation.

Encoding

☞ The input to a TM is always a string. If we want to provide an object other than a string as input, we must first represent that object as a string. Strings can easily represent polynomials, graphs, grammars, automata, and any combination of those objects. A TM may be programmed to decode the representation so that it can be interpreted in the way we intend. Our notation for the encoding of an object O into its representation as a string is $\langle O \rangle$. If we have several objects $O_1, O_2, \ldots, O_k$, we denote their encoding into a single string $\langle O_1, O_2, \ldots, O_k \rangle$. The encoding itself can be done in many reasonable ways. It doesn't matter which one we pick because a TM can always translate one such encoding into another.

☞ If the input description is simply w, the input is taken to be a string. If the input description is the encoding of an object as in $\langle A \rangle$, the Turing machine first implicitly tests whether the input properly encodes an object of the desired form and rejects it if it doesn't.

The Problem View vs. The Language View

We have two equivalent ways to describe a question: as a language and as a problem. For example:

The Problem View: Given a TM M and a string w, is $w \in L(M)$?

The Language View:

$$A_{TM} = \{\langle M, w\rangle | M \text{ is a TM and } M \text{ accepts } w\}.$$

Example 1:

$E_{DFA} = \{\langle A \rangle | A \text{ is a DFA and } L(A) = \varnothing\}$.

Theorem: E_{DFA} *is a decidable language.*

Proof: *A DFA accepts some string iff reaching an accept state from the start state by traveling along the arrows of the DFA is possible. (If the accepting states are all separated from the start state, then the language is empty.)*

$T =$ "*On input* $\langle A \rangle$, *where* A *is a DFA:*

1. *Mark the start state of* A.
2. *Repeat until no new states get marked:*
    3. *Mark any state that has a transition coming into it from any state that is already marked.*
4. *If no accept state is marked, accept; otherwise, reject.*"

Example 2: $E_{CFG} = \{\langle G \rangle | G \text{ is a CFG and } L(G) = \varnothing\}$.

Theorem: E_{CFG} *is a decidable language.*

☞ *To find an algorithm for this problem, we might attempt to use the CYK algorithm. It states that we can test whether a CFG generates some particular string* w*. To determine whether* $L(G) = \varnothing$*, the algorithm might try going through all possible* w*'s, one by one. But there are infinitely many* w*'s to try, so this method could end up running forever.*

☞ *In order to determine whether the language of a grammar is empty, we need to test whether the start variable can generate a string of terminals. The algorithm does so by solving a more general problem. It determines for each variable whether that variable is capable of generating a string of terminals. When the algorithm has determined that a variable can generate some string of terminals, the algorithm keeps track of this information by placing a mark on that variable. First, the algorithm marks all the terminal*

symbols in the grammar. Then, it scans all the rules of the grammar. If it ever finds a rule that permits some variable to be replaced by some string of symbols, all of which are already marked, the algorithm knows that this variable can be marked, too. The algorithm continues in this way until it cannot mark any additional variables.

$R =$ *"On input* $\langle G \rangle$*, where* G *is a CFG:*

1. *Mark all terminal symbols in* G*.*
2. *Repeat until no new variables get marked:*
    3. *Mark any variable* A *where* G *has a rule* $A \to U_1 U_2 \cdots U_k$ *and each symbol* $U_1, U_2, \ldots, U_k$ *has already been marked.*
4. *If the start variable is not marked, accept; otherwise, reject."*

عیناً همان کاری است که برای تعیین متغیرهای **generating** صورت می‌پذیرفت. اینجا، **generating** بودن متغیر استارت بررسی می‌شود.

Example 3:

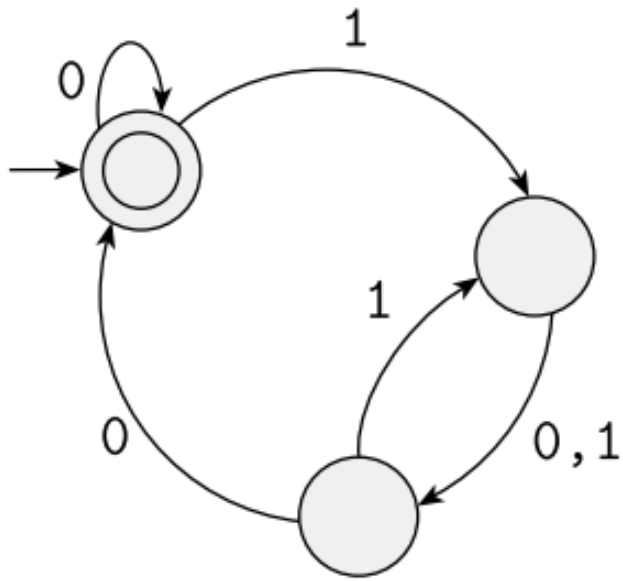
$$E_{TM} = \{\langle M \rangle | M \text{ is a TM and } L(M) = \varnothing\}.$$

Theorem: E_{TM} **is undecidable.**

چطور باید این حقیقت را اثبات کرد؟ هنوز مسلط نیستیم.

تمرین:

A**4.1** Answer all parts for the following DFA M and give reasons for your answers.



a. Is $\langle M, 0100 \rangle \in A_{\text{DFA}}$?

b. Is $\langle M, 011 \rangle \in A_{\text{DFA}}$?

c. Is $\langle M \rangle \in A_{\text{DFA}}$?

d. Is $\langle M, 0100 \rangle \in A_{\text{REX}}$?

e. Is $\langle M \rangle \in E_{\text{DFA}}$?

f. Is $\langle M, M \rangle \in EQ_{\text{DFA}}$?