COMS W3261
Computer Science Theory
Chapter 8 Notes

Alexander Roth

2014–10–10

Introduction to Turing Machines

Until now, we have been interested primarily in simple classes of languages and the ways that they can be used for relatively constrained problems, such as analyzing protocols, searching text, or parsing programs. Now, we shall start looking at the question of what languages can be defined by any computational device whatsoever. This question is tantamount to the question of what computers can do, since recognizing the strings in a language is a formal way of expressing any problem, and solving a problem is a reasonable surrogate for what it is that computers do.

We begin with an informal argument to show that there are specific problems that we cannot solve using a computer. These problems are called "undecidable." We then introduce a venerable formalism for computers, called the Turing machine. The Turing machine has long been recognized as an accurate model for what any physical computing device is capable of doing.

Problems That Computers Cannot Solve

The purpose of this section is to provide an informal, C-programming-based introduction to the proof of a specific problem that computers cannot solve. The particular problem we discuss is whether the first thing a C program prints is `hello, world`. Although we might imagine that simulations of the program would allow us to tell what the program does, we must in reality contend with programs that take an unimaginably long time before making any output at all. This problem − not knowing when, if ever, something will occur − is the ultimate cause of our inability to tell what a program does. However, proving formally that there is no program to do a stated task is quite tricky, and we need to develop some formal mechanics.

Programs that Pring “Hello, World”

Below is the first C program met by students who read Kernighan and Ritchie’s classic book. It is rather easy to discover that this program prints `hello, world` and terminates.

```
main() {
    printf("hello, world\n");
}
```

However, there are other programs that also print `hello, world`; yet the fact that they do so is far from obvious

```
int exp(int i, n)
/* computes i to the power n*/
{
    int ans, j;
    ans = 1;
    for (j=1; j<=n; j++) ans *= i;
    return(ans);
}

main()
{
    int n, total, x, y, z;
    scanf("%d", &n);
    total = 3;
    while (1) {
        for (x=1; x<=total-2; x++)
            for (y=1; y<=total-x-1;y++) {
                z = total - x - y;
                if (exp(x,n) + exp(y,n) == exp(z,n))
                    printf("hello, world\n");
            }
        total++;
    }
}
```

This program takes as input n, and looks for positive integer solutions to the equation $x^n + y^n = z^n$ If it finds one, it prints `hello, world`

To understand what this program does, first observer that `exp` is an auxiliary function to compute exponentials. The main program needs to search through triples (x, y, z) in an order such that we are sure we get to every triple of positive integers eventually. To organize the search properly, we use a fourth variable, `total`, that starts at 3 and, in the while-loop, is increased one unit at a time, eventually reaching any finite integer. Inside the while-loop, we divide `total` into three positive integers x, y, and z, by first allowing x to range from 1 to `total-2`, and within that for-loop allowing y to range from 1 up to one less than what x has not already taken from `total`. What remains, which must be between 1 and `total-2` is given to z.

In the innermost loop, the triple (x, y, z) is texted to see if $x^n + y^n = z^n$. If so, the program prints `hello, world`.

If the value of n that the program reads is 2, then it will eventually find combinations of integers such as `total` $= 12$, $x = 3$, $y = 4$, and $z = 5$, for which $x^n + y^n = z^n$. Thus, for input 2, the program will *does* print `hello, world`. However, for any integer $n > 2$, the program will never find a triple of positive integer to satisfy $x^n + y^n = z^n$.

Let us define the *hello-world problem* to be: determine whether a given C program, with a given input, prints `hello, world` as the first 12 characters that it prints.

Why Undecidable Problems Must Exist

Recall that a "problem" is really membership of a string in a language. The number of different languages over any alphabet of more than one symbol is not countable. That is, there is no way to assign integers to the languages such that every language has an integer, and every integer is assigned to one language.

On the other hand programs, being finite strings over a finite alphabet *are* countable. That is, we can order them by length, and for programs of the same length, order them lexicographically. Thus, we can speak of the first program, the second program, and in general, the ith program for any integer i.

As a result, we know there are infinitely fewer program than there are problems. If we picked a language at random, almost certainly it would be an undecidable problem. The only reason that most problems *appear* to be decidable is that we rarely are interested in random problems. Rather, we tend to look at fairly simple,, well- structured problems, and indeed these are often decidable. However, even among the problems we are interested in and can state clearly and succinctly, we find many that are undecidable; the hello-world problem is a case in point.

The Hypothetical "Hello, World" Tester

The proof of impossibility of making the hello-world test is a proof by contradiction. That is, we assume there is a program, call it H, that takes as input a program P and an input I, and tells whether P with input I prints `hello, world`. In particular, the only output H makes is either to print the three characters `yes` or to print the two characters `no`. It always does one or the other.

If a problem has an algorithm like H, that always tells correctly an instance of the problem has answer "yes" or "no," then the problem is said to be "decidable." Otherwise, the problem is "undecidable."

In order to prove that statement by contradiction, we are going to make several changes to H, eventually constructing a related program called H_2 that we does not exist. Since the changes to H are simple transformations that can be done to any C program, the only questionable statement is the existence of H, so it is that assumption we have contradicted.

To simplify our discussion, we shall make a few assumptions about C programs. These assumptions make H's job easier so if we can show a "hello-world tester" for these restricted programs does not exist, then surely there is no such tester that could work for a broader class of programs. Our assumptions are:

1. All output is character-based, e.g., we are not using a graphics package or any other facility to make output that is not in the form of characters.
2. All character-based output is performed using `printf`, rather than `putchar()` or another character-based output function.

We now assume that the program H exists. Our first modification is to change the output `no`, which is the response that H makes when its input program P does not print `hello, world` as its first output in response to input I. As soon as H prints "n", we know it will eventually follow with "o". Thus, we can modify any `printf` statement in H that prints "n" to instead print `hello, world`. Another `printf` statement that prints "o" but not the "n" is omitted. As a result, the new program, which we call H_1, behaves like H, except it prints `hello, world` exactly when H would print `no`.

Our next transformation on the program is a bit trickier; it is essentially the insight that allowed Alan Turing to prove his undecidability result about Turing machines. Since we are really interested in programs that take other programs as input and tell something about them, we shall restring H_1 so it:

a) Takes only input P, not P and I.

b) Asks what P would do if its input were its own code, i.e., what would H_1 do on inputs P as program and P as input I as well?

The modifications we must perform on H_1 to produce the program H_2 are as follows:

1. H_2 first reads the entire input P and stores it in an array A, which it "malloc's" for the purpose.
2. H_2 then simulates H_1, but whenever H_1 would read input from P or I. H_2 reads from the stored copy in A. To keep track of how much of P and I H_1 has read, H_2 can maintain two cursors that mark positions in A.

We are now ready to prove H_2 cannot exist. Thus, H_1 does not exist, and likewise, H does not exist. The heart of the argument is to envision what H_2 does when given itself as input. Recall that H_2, given any program P as input, makes output `yes` if P prints `hello, world` as its first output.

Suppose that the H_2 makes the output `yes`. Then H_2 that is receiving as input H_2 is saying about its input H_2 that H_2, given itself as input, prints `hello, world` as its first output. But we just supposed that the first output H_2 makes in this situation is `yes` rather than `hello, world`.

Thus it appears that the output of the machine is `hello, world`, since it must be one or the other. But if H_2, given itself as input, prints `hello, world` first, then the output of the machine must be `yes`. Whichever output we suppose H_2 makes, we can argue that it makes the other output.

This situation is paradoxical, and we conclude that H_2 cannot exist. As a result, we have contradicted the assumption that H exists.

Reducing One Problem to Another

A problem that cannot be solved by computer is called *undecidable.* Suppose we want to determine whether or not some other problem is solvable by a computer. We can try to write a program to solve it, but if we cannot figure out how to do so, then we might try a proof that there is no such program.

Once we have one problem that we know is undecidable, we no longer have to prove existence of a paradoxical situation. It is sufficient to show that if we could solve the new problem, then we could use that solution to solve a problem we already know is undecidable. This technique is called the *reduction* of P_1 to P_2.

Suppose that we know problem P_1 is undecidable, and P_2 is a new problem that we would like to prove is undecidable as well. We suppose that there is a program that prints `yes` or `no`, depending on whether its input instance of problem P_2 is or is not in the language of that problem.

In order to make a proof that problem P_2 is undecidable, we have to invent a construction that converts instances of P_1 to instances of P_2 that have the same answer. That is, any string in the language P_1 is converted to some string in the language P_2, and any string over the alphabet of P_1 that is *not* in the language P_1 is converted to a string that is not in the language P_2. Once we have this construction, we can solve P_1 as follows:

1. Given an instance of P_1, that is, given a string w that may or may not be in the language P_1, apply the construction algorithm to produce a string x.

2. Test whether x is in P_2, and give the same answer about w and P_1.

If w is in P_1, then x is in P_2, so this algorithm says `yes`. If w is not in P_1, then x is not in P_2, and the algorithm says `no`. Since we assumed that no algorithm to decide membership of a string in P_1 exists, we have a proof by contradiction that the hypothesized decision algorithm for P_2 does not exist; i.e., P_2 is undecidable.

The Turing Machine

The purpose of the theory of undecidable problems is not only to establish the existence of such problems but to provide guidance to programmers about what they might or might not be able to accomplish through programming. These problems, called "intractable problems," tend to present greater difficulty to the programmer and the system designer that do the undecidable problems. The reason is that, while undecidable problems are usually quite obviously so,and their solutions are rarely attempted in practice, the intractable problems are faced every day. Moreover, they often yield to small modifications in the requirements or to heuristic solutions. Thus, the designer is faced quite frequently with having to decide whether or not a problem is in the intractable class, and what to do about it, if so.

We need tools that will allow us to prove everyday questions undecidable or intractable. As a result, we need to rebuild our theory of undecidability, based not on programs in C or another language, but based on a very simple model of a computer, called the Turing machine. This device is essentially a finite automaton that has a single tape of infinite length on which it may read and write data. One advantage of the Turing machine over programs as representation of what can be computed is that the Turing machine is sufficiently simple that we can represent its configuration precisely, using a simple notation much like ID's of a PDA.

The Quest to Decide All Mathematical Questions

At the turn of the 20th century, the mathematician D. Hilbert asked whether it was possible to find an algorithm for determining the truth or falsehood of any mathematical proposition. In particular, he asked if there was a way to determine whether any formula in the first-order predicate calculus, applied to integers, was true.

However, in 1931, K. Gödel published his famous incompleteness theorem. He constructed a formula in the predicate calculus applied to integers, which asserted that the formula itself could be neither proved nor disproved.

The predicate calculus was not the only notation that mathematicians had for "any possible computation." In 1936, A. M. Turing proposed the Turing machine as a model of "any possible computation." This model is computer-like, rather than program-like, even though true electronic, or even electromechanical computers were several years in the future.

The unprovable assumption that any general way to compute will allow us to compute only the partial-recursive functions is known as *Church's hypothesis* or the *Church-Turing thesis.*

Notaiton for the Turing Machine

A Turing machine consists of a *finite control*, which can be in any of a finite set of states. There is a *tape* divided into squares or *cells*; each cell can hold any one of a finite number of symbols.

Initially, the *input*, which is a finite-length string of symbols chosen from the *input alphabet*, is placed on the tape. All other tape cells extending infinitely to the left and right, initially hold a special symbol called *blank*. The blank is a *tape symbol*, but not an input symbol, and there may be other tap symbols besides the input symbols and the blank as well.

There is a *tape head* that is always positioned at one of the tape cells. The Turing machine is said to be *scanning* that cell. Initially, the tape head is at the leftmost cell that holds the input.

A *move* of the Turing machine is a function of the state of the finite control and the tape symbol scanned. In one move, the Turing machine will:

1. Change state. The next state optionally may be the same as the current state.

2. Write a tape symbol in the cell scanned. This tape symbol replaces whatever symbol was in that cell. Optionally, the symbol written may be the same as the symbol currently there.

3. Move the tape head left or right.

The formal notation we shall use for a *Turing machine* (TM) is similar to that used for finite automata or PDA's.

$$M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

whose components have the following meanings:

Q: The finite set of *states* of the finite control.

Σ: The finite set of *input symbols.*

Γ: The complete set of *tape symbols*; Σ is always a subset of Γ.

δ: The *transition function.* The arguments of $\delta(q, X)$ are a state q and a tape symbol X. The value of $\delta(q, X)$, if it is defined, is a triple (p, Y, D), where:

   1. P is the next state, in Q.
   2. Y is the symbol, in Γ, written in the cell being scanned, replacing whatever symbol was there.
   3. D is a *direction*, either L or R, standing for "left" or "right," respectively, and telling us the direction in which the head moves.

q_0: The *start state*, a member of Q, in which the finite control is found initially.

B: The *blank* symbol. This symbol is in Γ but not in Σ; i.e., it is not an input symbol. The blank appears initially in all but the finite number of initial cells that hold input symbols.

F: The set of *final* or *accepting* states, a subset of Q.

Instantaneous Descriptions for Turing Machines

In order to describe formally what a Turing machine does, we need to develop a notation for configurations or *instantaneous descriptions* (ID's), like the notation we developed for PDA's. After any finite number of moves, the TM can have visited only a finite number of cells, even though the number of cells visited can eventually grow beyond any finite limit. Thus, in every ID, there is an infinite prefix and an infinite suffix of cells that have never been visited. These cells must all hold either blanks or one of the finite number of input symbols.

In addition to representing the tape, we must represent the finite control and the tape-head position. To do so, we embed the state in the tape, and place it immediately to the left of the cell scanned. To disambiguate the tape-plus-state string, we have to make sure that we do not use as a state any symbol that is also a tape symbol. Thus, we shall use the string $X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n$ to represent an ID in which

1. q is the state of the Turing machine.
2. The tape head is scanning the ith symbol from the left.
3. $X_1X_2\cdots X_n$ is the portion of the tape between the leftmost and the rightmost nonblank. As an exception, if the head is to the left of the leftmost nonblank or to the right of the rightmost nonblank, then some prefix or suffix of $X_1X_2\cdots X_n$ will be blank, and i will be 1 or n respectively.

We describe moves of a Turing machine $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ by the $\underset{M}{\vdash}$ notation that was used for PDA's. When the TM M is understood, we shall use just $\vdash$ to reflect moves. As usual, $\overset{*}{\underset{M}{\vdash}}$, or just $\overset{*}{\vdash}$, will be used to indicate zero, one, or more moves of the TM M.

Suppose $\delta(q, X_i) = (p, Y, L)$; i.e., the next move is leftward. Then

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \underset{M}{\vdash} X_1X_2\cdots X_{i-2}pX_{i-1}YX_{i+1}\cdots X_n$$

Notice how this move reflects the change to state p and the fact that the tape head is now positioned at cell $i-1$. There are two important exceptions:

1. If $i = 1$, then M moves to the blank to the left of X_1. In that case,

$$qX_1X_2\cdots X_n \underset{M}{\vdash} pBYX_2\cdots X_n$$

2. If $i = n$ and $Y = B$, then the symbol B written over X_n joins the infinite sequence of trailing blanks and does not appear in the next ID. Thus,

$$X_1X_2\cdots X_{n-1}qX_n \underset{M}{\vdash} X_1X_2\cdots X_{n-2}pX_n$$

Now suppose $\delta(q, X_i) = (p, Y, R)$; i.e., the next move is rightward. Then

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \underset{M}{\vdash} X_1X_2\cdots X_{i-1}pYX_{i+1}\cdots X_n$$

Here, the move reflects the fact that the head has moved to cell $i+1$. Again there are two important exceptions:

1. If $i = n$, then the $i+1$st cell holds a blank, and that cell was not part of the previous ID. Thus, we instead have

$$X_1 X_2 \cdots X_{n-1} X q X n \underset{M}{\vdash} X_1 X_2 \cdots X_{n-1} Y p B$$

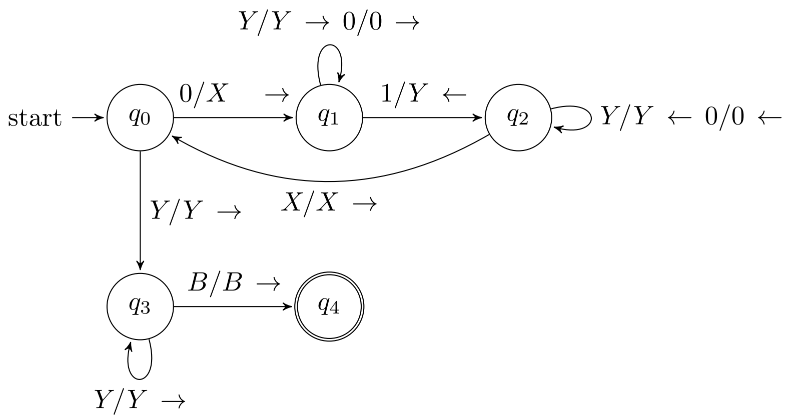
2. If $i = 1$ and $Y = B$, then the symbol B written over X_1 joins the infinite sequence of leading blanks and does not appear in the next ID. Thus,

$$q X_1 X_2 \cdots X_n \underset{M}{\vdash} p X_2 \cdots X_n$$

Transition Diagrams for Turing Machines

We can represents the transitions of a Turing machine pictorially, much as we did for the PDA. A *transition diagram* consists of a set of nodes corresponding to the states of the TM. An arc from state q to state p is labeled by one or more items of the form X/YD, where X and Y are tape symbols, and D is a direction, either L or R. That is, whenever $\delta(q, X) = (p, Y, D)$, we find the label X/YD on the arc from q to p. However, in our diagrams, the direction D is represented pictorially by $\leftarrow$ for "left" and $\rightarrow$ for "right."

As for other kinds of transition diagrams. we represent the start state by the word "Start" and an arrow entering that state. Accepting states are indicated by double circles. Thus, the only information about the TM one cannot read directly is the symbol used for the blank.



Transition diagram for a TM that accepts strings of the form $0^n 1^n$

The Language of a Turing Machine

The input string is placed on the tape, and the tape head begins at the leftmost input symbol. If the TM eventually enters an accepting state, then the input is accepted, and otherwise not.

More formally, let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a Turing machine. Then $L(M)$ is the set of strings w in Σ^* such that $q_0 w \overset{*}{\vdash} \alpha p \beta$ for some state p in F and any tape strings α and β.

The set of languages we can accept using a Turing machine is often called the *recursively enumerable languages* or RE languages. The term "recursively enumberable" comes from computational formalisms that predate the Turing machine but that define the same class of languages or arithmetic functions.

Turing Machines and Halting

There is another notion of "acceptance" that is commonly used for Turing machines: acceptance by halting. We say a TM *halts* if it enters a state q, scanning a tape symbol X, and there is no move in this situation; i.e., $\delta(q, X)$ is undefined.

Without changing the language accepted, we can make $\delta(q, X)$ undefined whether q is an accepting state. In general, without otherwise stating so:

- We assume that a TM always halts when it is in an accepting state.

Unfortunately, it is not always possible to require that a TM halts even if it does not accepts. Those languages with Turing machines that do halt eventually, regardless of whether or not they accept, are called *recursive.* Turing machines that always halt, regardless of whether or not they accept, are a good model of an "algorithm." If an algorithm to solve a given problem exists, then we say the problem is "decidable," so TM's that always halt figure importantly into decidability theory.

Notaional Conventions for Turing Machines

The symbols we normally use for Turing machines resemble those for the other kinds of automata we have seen.

1. Lower-case letters at the beginning of the alphabet stand for input symbols
2. Capital letters, typically near the end of the alphabet, are used for tape symbols that may or may not be input symbols. However, B is generally used for the blank symbol.
3. Lower-case letters near the end of the alphabet are strings of input symbols.
4. Greek letters are strings of tape symbols
5. Letters such as q, p, and nearby letters are states.

Programming Techniques for Turing Machines

Turing machines can perform a sort of calculations on other Turing machines. This “introspective” ability of Turing machines enables us to prove problems undecidable.

Storage in the State

We can use the finite control not only to represent a position in the “program” of the Turing machine, but to hold a finite amount of data. Suppose we have a Turing machine that has a finite control consisting of not only a “control” state q, but three data elements A, B, and C. The technique requires no extension to the TM model; we merely think of the state as a tuple: $[q, A, B, C]$.

Multiple Tracks

Another useful “trick” is to think of the tape of a Turing machine as composed of several tracks. Each track can hold one symbol, and the tape alphabet of the TM consists of tuples, with one component for each “track”.

Subroutines

A Turing-machine subroutine is a set of states that perform some useful process. This set of states includes a start state and another state that temporarily has no moves, and that serves as the “return” state to pass control to whatever other set of states called the subroutine.

Extensions to the Basic Turing Machine

Multitape Turing Machines

A multitape TM has a finite control (state), and some finite number of tapes. Each tape is divided into cells, and each cell can hold any symbol of the finite tape alphabet. The set of tape symbols includes a blank, and has a subset called the input symbols, of which the blank is not a member. The set of states includes an initial state and some accepting states. Initially:

1. The input, a finite sequence of input symbols, is placed on the first tape.

2. All other cells of all the tapes hold the blank.

3. The finite control is in the initial state.

4. The head of the first tape is at the left end of the input.

5. All other tape heads are at some arbitrary cell. Since tapes other than the first tape are completely blank, it does not matter where the head is placed initially; all cells of these tapes “look” the same.

A move of the multitape TM depends on the state and teh symbol scanned by each of the tape heads. In one move, the multitape TM does the following:

1. The control enters a new state, which could be the same as the previous state.
2. One each tape, a new tape symbols is written on the cell scanned. Any of these symbols may be the same as the symbol previously there.
3. Each of the tape heads makes a move, which can be either left, right, or stationary. The heads move independently, so different heads may move in different directions, and some may not move at all.

Equivalence of One-Tape and Mutitape TM's

Theorem 1. Every language accepted by a multitape TM is recursively enumerable.

Running Time and the Many-Tapes-to-One Construction

The *running time* of TM M on input w is the number of steps that M makes before halting. If M doesn't halt on w, then the running time of M on w is infinite. The *time complexity* of TM M is the function $T(n)$ that is the maximum, over all inputs w of length N of the running time of M on w. For Turing machines that do not halt on all inputs, $T(n)$ may be infinite for some or even all n.

Theorem 2. The time taken by the one-tape TM N of Theorem 1 to simulate n moves of the k-type TM M is $O(n^2)$.

Nondeterministic Turing Machines

A *nondeterministic* Turing machine (*NTM*) differs from deterministic variety by having a transition function δ such that for each state q and tape symbol X, $\delta(q, X)$ is a set of triples

$$\{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \ldots, (q_k, Y_k, D_k)\}$$

where k is any finite integer. The NTM can choose, at each step, any of the triples to be the next move. It cannot, however, pick a state from one, a tape symbol from another, and the direction from yet another.

M accepts an input w if there is any sequence of choices of move that leads from the initial ID with w as input, to an ID with an accepting state.

Theorem 3. If M_N is a nondeterministic Turing machine, then there is a deterministic Turing machine M_D such that $L(M_N) = L(M_D)$.

The constructed deterministic TM may take exponentially more time than the nondeterministic TM.

Restricted Turing Machine

Turing Machines With Semi-infinite Tapes

We can assume the tape is *semi-infinite*, that is, there are no cells to the left of the initial head position. The trick behind the construction is to use two tracks on the semi-infinite tape. The upper track represents the cells of the original TM that are at or to the right of the initial head position. The lower track represents the positions left of the initial position, but in reverse order.

We shall also assume that this Turing machine never writes a blank. This simple restriction, coupled with the restriction that the tape is only semi-infinite, means that the tape is at all times a prefix of nonblank symbols followed by an infinity of blanks. Further, the sequence of nonblanks always begins at the initial tape position.

Theorem 4. Every language accepted by a TM M_2 is also accepted by a TM M_1 with the following restrictions:

1. M_1's head never moves left of its initial position.
2. M_1 never writes a blank.

Multistack Machines

A k-stack machine is a deterministic PDA with k stacks. It obtains its input, like the PDA does, from an input source, rather than having the input placed on a tape or stack, as the TM does. The multistack machine has a finite control, which is in one of a finite set of states. It has a finite stack alphabet, which it uses for all its stacks. A move of the multistack machine is based on:

1. The state of the finite control.
2. The input symbol read, which is chosen from the finite input alphabet.
3. The top stack symbol on each of its stacks.

In one move, the multistack machine can:

1. Change to a new state.
2. Replace the top symbol of each stack with a string of zero or more stack symbols. There can be a different replacement string for each stack.

Thus, a typical transition rule for a k-stack machine looks like:

$$\delta(q, a, X_1, X_2, \ldots, X_k) = (p, \gamma_1, \gamma_2, \ldots, \gamma_k)$$

The interpretation of this rule is that in state q, with X_i on top of the ith stack for $i = 1, 2, \ldots, k$, the machine may consume a form its input, go to state p, and replace X_i on top of the ith stack by string y_i, for each $i = 1, 2, \ldots, k$. The multistack machine accepts by entering a final state.

We add one capability that simplifies input processing by this deterministic machine: there is a special symbol \$, called the *endmarker*, that appears only at the end of the input and is not part of that input.

Theorem 5. If a language L is accepted by a Turing machine, then L is accepted by a two-stack machine.

Counter Machines

A *counter machine* may be though of in one of two ways:

1. The counter machines has the same structure as the mutlistack machine, but in place of each stack is a counter. Counters hold any nonnegative integer, but we can only distinguish between zero and nonzero counters. That is, the move of the counter machine depends on its state, the input symbol, and which, if any, of the counters are zero. In one move, the counter machine can:
   (a) Change state.
   (b) Add or subtract 1 from any of its counters, independently.
2. A counter machine may also be regarded as a restricted multistack machine. The restrictions are as follows:
   (a) There are only two stack symbols, which are Z_0 (the *bottom-of- stack marker*), and X.
   (b) Z_0 is initially on each stack.
   (c) We may replace Z_0 only by a string of the form $X^i Z_0$, for some $i \geq 0$.
   (d) We may replace X only by X^i for some $i \geq 0$. That is, Z_0 appears only on the bottom of each stack, and all other stack symbols, if any, are X.

The Power of Counter Machines

There are a few observations about the languages accepted by counter machines:

- Every language accepted by a counter machine is recursively enumerable.
- Every language accepted by a one-counter machine is a CFL.

Two counters are enough to simulate a Turing machine and therefore to accept every recursively enumerable language.

Theorem 6. Every recursively enumerable language is accepted by a three counter machine.

Theorem 7. Every recursively enumerable language is accepted bya two-counter machine.

Turing Machines and Computers

1. A computer can simulate a Turing machine
2. A Turing machine can simulate a computer, and can do so in an amount of time that is at most some polynomial in the number of steps taken by the computer.

Simulating a Turing Machine by Computer

Given a particular TM M, we must write a program that acts like M. One aspect of M is its finite control. Since there are only a finite number of states and a finite number of transition rules, our program can encode states as character strings and use a table of transitions, which it loos up to determine each move. Likewise, the tape symbols can be encoded as character strings of a fixed length, since there are only a finite number of tape symbols.

Let us assume that memory is not an issue and that we can replace hard drives with empty, but otherwise identical, hard disks. Since there is no obvious limit on how many disks we can use, let us assume that as many disks as the computer needs is available. One stack holds the data in the cells of the Turing machine that are located significantly to the left of the tape head, and the other stack holds data significantly to the right of the tape head.

If the tape head of the TM moves sufficiently far to the left that it reaches cells that are not represented by the disk currently mounted in the computer, then it prints a message "swap left." The disk is then replaced by the disk on top of the left stack and computation resumes.

Repeat the same process for the right stack. If either stack is empty when the computer asks that a disk from that stack be mounted, then the TM has entered an all-blank region of tape.

Simulating a Computer by a Turing Machine

Let us give a realistic but informal model of how a typical computer operates.

- Suppose that the storage of a computer consists of an indefinitely long sequence of *words*, each with an *address*.
- Assume that the program of the computer is stored in some of the words of memory. These words each represent a simple instruction, as in the machine or assembly language of a typical computer.
- Assume that each instruction involves a limited number of words, and that each instruction changes the value of at most one word.
- We will allow any operation to be performed on any word.

This TM uses several tapes, but it could be converted to a one-tape TM.

The first tape represents the entire memory of the computer. Both addresses and contents are written in binary.

The second tape is the “instruction counter.” This tape holds one integer in binary, which represents one of the memory locations on tape 1. The value stored in this location will be interpreted as the next computer instruction to be executed.

The third tape holds a “memory address” or the contents of that address after the address has been located on tape 1. To execute an instruction, the TM must find the contents of one or more memory addresses that hold data involved in the computation.

Our TM will simulate the *instruction cycle* of the computer, as follows.

1. Search the first tape for an address that matches the instruction number on tape 2.
2. When the instruction address is found, examine its value.
3. If the instruction requires the value of some address, then that address will be part of the instruction.
4. Execute the instruction, or the part of the instruction involving this value.
5. After performing the instruction, and determining that the instruction is not a jump, add 1 to the instruction counter on tape 2 and begin the instruction cycle again.

A fourth tape holding the simulated input to the computer, since the computer must read its input from a file. The TM can read from this tape instead.

A scratch tape is also used for efficiency. Finally, we assume that the computer makes an output that tells whether or not its input is accepted. When the TM simulates the execution of the computer accepting, it enters an accepting state of its own and halts.

Comparing the Running Times of Computers and Turing Machines

- The issue of running time is important because we shall use the TM not only to examine the question of what can be computed at all, but what can be computed with enough efficiency that a problem’s computer-based solution can be used in practice.
- The dividing line separating the *tractable* – that which can be solved efficiently – from the *intractable* – problems that can be solved, but not fast enough for the solution to be usable – is generally held to be between what can be computed in polynomial time and what requires more than any polynomial running time.
- Thus, if a problem can be solved in polynomial time on a typical computer, then it can be solved in polynomial time by a Turing machine and conversly.

Theorem 8. If a computer:

1. Has only instructions that increase the maximum word length by at most 1, and

2. Has only instructions that a multitape TM can perform on words of length k in $O(k^2)$ steps or less, then the Turing machine described can simulate n steps of the computer in $O(n^3)$ of its own steps.

Theorem 9. A computer of the type described in Theorem 8 can be simulated for n steps by a one-tape Turing machine, using at most $O(n^6)$ steps of the Turing machine.

Summary of Chapter 8

The Turing Machine: The TM is an abstract computing machine with the power of both real computers and of other mathematical definitions of what can be computed. The TM consists of a finite-state control and an infinite tape divided into cells. Each cell holds one of a finite number of tape symbols, and one cell is the current position of the tape head. The TM makes moves based on its current state and the tape symbol at the cell scanned by the tape head. In one move, it changes state, overwrites the scanned cell with some tape symbol, and moves the head one cell left or right.

Acceptance by a Turing Machine: The TM starts with its input, a finite-length string of tape symbols, on its tape, and the rest of the tape containing the blank symbol on each cell. The blank is one of the tape symbols, and the input is chosen from a subset of the tape symbols, not including blank, called the input symbols. The TM accepts its input if it ever enters an accepting state.

Recursively Enumerable Languages: The languages accepted by TM's are called recursively enumerable (RE) languages. Thus, the RE languages are those languages are those languages that can be recognized or accepted by any sort of computing device.

Instantaneous Descriptions of a TM: We can describe the current configuration of a TM by a finite-length string that includes all the tape cells from the leftmost to the rightmost nonblank. The state and the position of the head are shown by placing the state within the sequence of tape symbols, just to the left of the cell scanned.

Storage in the Finite Control: Sometimes, it helps to design a TM for a particular language if we imagine that the state has two or more components. One component is the control component, and functions as a state normally does. The other components hold data that the TM needs to remember.

Multiple Tracks: It also helps frequently if we think of the tape symbols as vectors with a fixed number of components. We may visualize each component as a separate track of the tape.

Multitape Turing Machines: An extended TM model has some fixed number of tapes greater than one. A move of this TM is based on the state and on the vector of symbols scanned by the head on each of the tapes. In a move, the multitape TM changes state, overwrites symbols on the cells scanned by each of its tape heads, and moves any or all of its tape heads one cell in either direction. Although able to recognize certain languages faster than the conventional one-tape TM, the multitape TM cannot recognize any language that is not RE.

Nondeterministic Turing Machines: The NTM has a finite number of choices of next move (state, new symbol, and head move) for each state and symbol scanned. It accepts an input if any sequence of choices leads to an ID with an accepting state. Although seemingly more powerful than the deterministic TM, the NTM is not able to recognize any language that is not RE.

Semi-infinite-Tape Turing Machines: We can restrict a TM to have a tape that is infinite only to the right, with no cells to the left of the initial head position. Such a TM can accept any RE language.

Multistack Machines: We can restrict the tapes of a multitape TM to behave like a stack. The input is on a separate tape, which is read once from left-to-right, mimicking the input mode for a finite automaton or PDA. A one- stack machine is really a DPDA, which a machine with two stacks can accept any RE language.

Counter Machines: We may further restrict the stacks of a multistack machine to have only one symbol other than a bottom-marker. Thus, each stack functions as a counter, allowing us to store a nonnegative integer, and to test whether the integer stored is 0, but nothing more. A machine with two counters is sufficient to accept any RE language.

Simulating a Turing Machine by a real computer: It is possible, in principle, to simulate a TM by a real computer if we accept that there is a potentially infinite supply of a removable storage device such as a disk, to simulate the nonblank portion of the TM tape. Since the physical resources to make disks are not infinite, this argument is questionable. However, since the limits on how much storage exists in the universe are unknown and undoubtedly vast, the assumption of an infinite resource, as in the TM tape, is realistic in practice and generally accepted.

Simulating a Computer by a Turing Machine: A TM can simulate the storage and control of a real computer by using one tape to store all the locations and their contents: register, main memory, disks, and other storage devices. Thus, we can be confident that something not doable by a TM cannot be done by a real computer.