jn 2004.07.13

6. Effective computability: Turing machines

Goals of this chapter: Rigorous definition of “algorithm” or “effective procedure”. Formal systems that capture this intuitive concept . Church-Turing thesis. Universal Turing machines. Concepts: computable, decidable, semi-decidable, enumerable.

6.1 What is an “algorithm”?

“.. and always come up with the right answer, so God will”, Muhammad ibn Musa, Al-Khowarizmi (whose name is at the origin of the words “algorithm” and “algebra”)

David Hilbert’s 10-th problem (1900): “10. Entscheidung der Lösbarkeit einer diophantischen Gleichung. Eine diophantischen Gleichung mit irgendwelchen Unbekannten und mit ganzen rationalen Zahlkoeffizienten sei vorgelegt; **man soll ein Verfahren angeben**, nach welchem sich mittels einer endlichen Anzahl von Operationen entscheiden lässt, ob die Gleichung in ganzen rationalen Zahlen lösbar sei.”
“Devise a process by which it can be decided, by a finite number of operations, whether a given multivariate polynomial has integral roots”.
E.g: x2 + y2 + 1 has no real integral roots, whereas xy + x - y -1 has infinitely many (e.g. x = 1, y arbitrary).

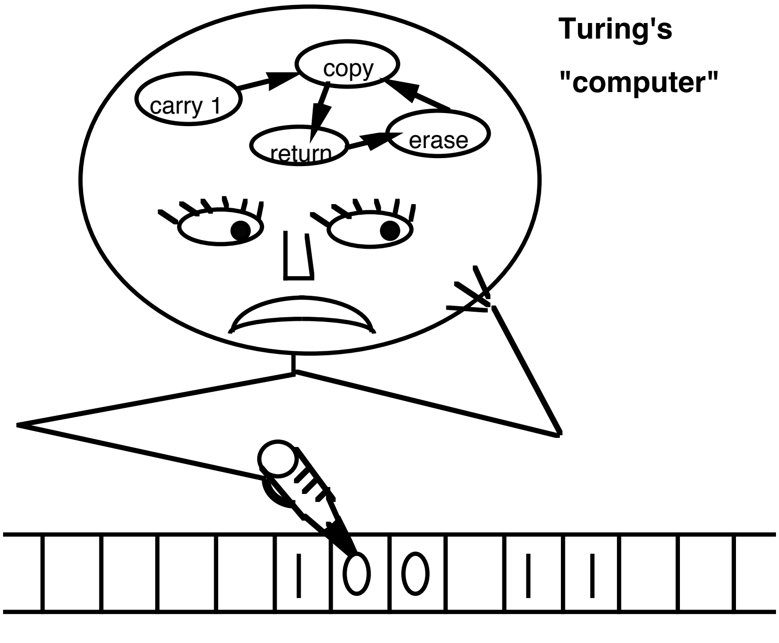
Hilbert’s formulation implies the assumption that such a decision process exists, waiting to be discovered. For polynomials in a single variable, P(x) = a0 + a1 x + a2 x^2 + ... + an x^n , such a decision process was known, based on formulas that provably bound the absolute value of any root x0, e.g. | x0| ≤ 1 + max | ai / an | , where max runs over the indices i = 0 to n-1 [A. Cauchy 1837]. Any such bound B yields a trivial decision procedure by trying all integers of absolute value < B.

It appears that mathematicians of 1900 could not envision the possibility that no such decision process might exists. Not until the theory of computability was founded in the 30s by Alonzo Church, Alan Turing and others, it became clear that Hilbert’s 10-th problem should be formulated as a question: “does there exist a process according to which it can be determined by a finite number of operations ...?”. In 1970 it was no longer a surprise when Y. Matijasevich proved the Theorem:

Hilbert’s 10-th problem is undecidable, i.e. there exists no algorithm to solve it.

For this to be a theorem, we need to define rigorously the concept of algorithm or effective procedure.

Turing’s definition of effective procedure follows from an analysis of how a **human(!) computer** proceeds when excuting an algorithm. Alan M. Turing: On computable numbers, with an application to the Entscheidungsproblem, Proc. London Math Soc., Ser. 2-42, 230-265, 1936.



Turing's "computer"

“Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e., on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent. The effect of this restriction on the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. ... The difference from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at a glance. .. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same.

The behavior of the computer at any moment is determined by the symbols which he is observing, and his 'state of mind' at the moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite.

Let us imagine the operations performed by the computer to be split up into 'simple operations' which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer, and the state of mind of the computer. We may suppose that in a simple operastion, not more than one symbol is altered. ..

Besides these changes of symbols, the simple operations must include changes of distributions of observed squares. I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount. Let us say that each of the new observed squares is within L squares of an immediately previously observed square. ...

The simple operations must therefore include:

(a) Changes of symbol on one of the observed squares

(b) Changes of one of the squares observed to another square within L squares of one of the previously observed squares.

The most general single operation must therefore be taken to be one of the following:

(A) A possible change (a) of symbol together with a possible change of state of mind.

(B) A possible change (b) of observed squares, together with a possible change of state of mind.

...**We may now construct a machine to do the work of this computer."**

6.2 The Church-Turing thesis

Turing machines

Alan M. Turing: On computable numbers, with an application to the Entscheidungsproblem, Proc. London Math Soc., Ser. 2, vol.42, 230-265, 1936; vol.43, 544-546, and

Computability and λ-definability, The J. of Symbolic Logic, vol.2, 153-163, 1937.

and other models of computation, such as

- **effective calculability, λ-calculus**

  Alonzo Church: An unsolvable problem of elementary number theory, American J. of Math. 58, 345-363, 1936, and

  A note on the Entscheidungsproblem, The J. of Symbolic Logic, vol.1, 40-41, corrected 101-102, 1936.

- **canonical systems**

  Emil Post: Formal reductions of the general combinatorial decision problem, American J. of Math. 65, 197-268, 1943.

- **recursive functions**

  Stephen C. Kleene: General recursive functions of natural numbers, Math Annalen 112, 727-742, 1936, and λ-definability and recursiveness, Duke Math. J. 2, 340-353, 1936.

- **Markov algorithms**

  A.A. Markov: The theory of algorithms (Russian), Doklady Akademii Nauk SSSR vol. 58, 1891-92, 1951, and Trudy Math. Instituta V.A.Steklova vol. 42, 1954.

turn out to be mathematically equivalent: any one of these models of computation can simulate any other. This equivalence strengthens the argument that each of them captures in a rigorous manner the

intuitive concept of “algorithm”

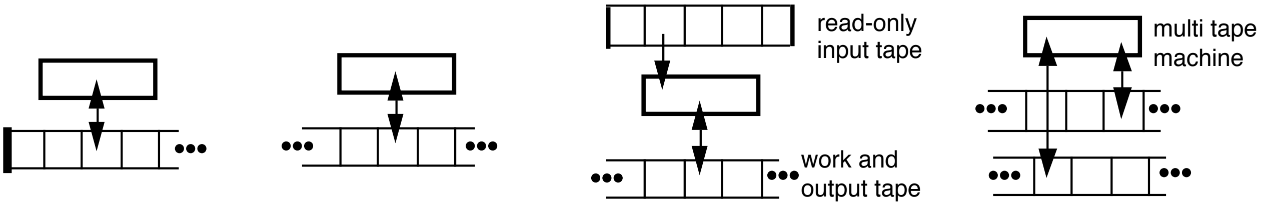
The question of effective computability was suddenly raised in the 30s and investigated by several logicians using different formalisms because of the crisis in the foundation of mathematics produced by

Gödel’s incompleteness theorem: Any system of logic powerful enough to express elementary arithmetic contains true statements that cannot be proven within that system (Kurt Gödel, Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, Monatshefte für Mathematik und Physik 38, 173-198, 1931). Natural question: what can and what cannot be proven (computed, decided, ..) within a given logical system (model of computation)?

6.3 Turing machines: concepts, varieties, simulation

A Turing machine (TM) is a finite state machine that controls one or more tapes, where at least one tape is of unbounded length. TMs come in many different flavors, and we use different conventions whenever it is convenient. This chapter considers **deterministic TMs** (DTM), the next one also **non-deterministic TMs** (NDTM). DTMs and NDTMs are equivalent in computing power, but not with respect to computing performance.

Apart from the distinction DTM vs. NDTM, the greatest differences among various TM models have to do with the number and access charactistic of tapes, e.g.: semi-infinite or doubly infinite tape; separate read-only input tape; multi-tape machines; input alphabet differs from work tape alphabet. Minor differences involve the precise specification of all the actions that may occur during a transition, such as whether halting is part of a transition, or requires a halting state. These differences affect the complexity of computations and the convenience of programming, but **not** the computational power of TMs in terms of what they can compute, given unbounded time and memory. Any version of a TM can simulate any other, with loss or gain of time or space complexity.



The “standard” TM model is deterministic with a single semi-infinite tape (it has one fixed end and is unbounded in the other direction) used for input, computation, and output. It is defined by the following components:

M = (Q, A, f: Q x A -> Q x A x {L, R, -, H}, q0, F}.

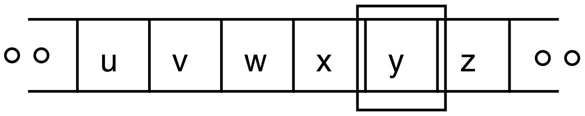
Q: finite state space; A: finite alphabet; f: transition function; q0: initial state; F ⊆ A accepting states.
{L, R, -, H}: tape actions: L = move left, R = move right, - stay put, optional H = halt.
If a tape action calls for the read/write head to drop off the fixed end of the tape, this action is ignored.

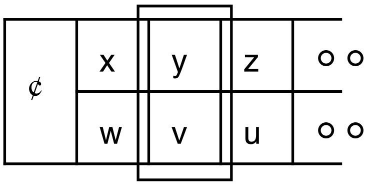
Non-deterministic TM: replace f: Q x A -> Q x A x {L, R, ..} by f: Q x A -> $2^{Q \times A \times \{L, R, ..\}}$.
A deterministic TM can simulate a non-deterministic TM N by **breadth-first traversal** of the tree of all possible executions of N. Why not depth-first traversal?

Multi-tape TMs are defined analogously: The transition function of a k-tape TM depends on all the k characters currently being scanned, and specifies k separate tape actions.

Df: A **configuration** of a TM M with tapes T1, .. Tk is given by the content of all the tapes, the position of all read/write heads, and the (internal) state of M. A configuration implicitly specifies the entire future of a TM.

Ex of a **simulation**: a TM M1 with semi-infinite tape can simulate a TM M2 = (Q, A, f, q0) with doubly infinite tape. Fold M2’s tape at some arbitrary place, and let M1 read/write two squares of the folded tape at once, thus expanding the alphabet from A to A x A. We use an additional character ¢ to mark the end of the folded tape. The internal state of M1 remembers whether M2’s current square is on the upper half or on the lower half of the tape; at any moment, M1 effectively reads/writes on only one of the “half-tapes”, though formally it reads/writes on both.





Given M2 = (Q, A, f, q0), construct M1 = (Q', A', f', q0') as follows:
Q' is the Cartesian product Q x {up, down}, which we abbreviate as Q' = { ui, di | for each qi ∈ Q},
i.e each state qi of M generates a state ui (up) and a state di (down). A' = A x A ∪ { ¢ } .

The transition function f': Q' x A' -> Q' x A' x {L, R, -, H} explodes quadratically.
Each transition qi, a -> qj, b, m of M2 generates 2 |A| transitions of the following form:
ui, (a, -) -> uj, (b, -), m , di, (-, a) -> dj, (-, b), m^{-1}
Here (a, -) stands for all symbols in A x A whose upper square contains an a, the lower square any letter in A. Analogously for (-, a). m^{-1} denotes the inverse motion of m, i.e. L and R are each others' inverse. Two additional transitions handle the U-turn at the left end of the tape: ui, ¢ -> di, ¢ , R ; di, ¢ -> ui, ¢, R

The initial state q0' of M1 is either u0 or d0 depending on the initial configuration of M2.
M1 simulates M2 "in real time", transition for transition, only occasionally wasting a transition for a U-turn.

Most other simulations involve a considerable loss of time, but for the purpose of the theory of computability, the only relevant issue about time is whether a computation terminates or doesn't. For the purpose of complexity theory (next chapter) speed-up and slow-down do matter. But complexity theory often considers a polynomial-time slow down to be negligible. From this point of view, simulating a multi-tape TM on a single-tape TM is relatively inexpensive, as the following theorem states.

Thm: A TM Mk with k tapes using time t(n) and space s(n) can be simulated by a single-tape TM M1 using time O($t^2(n)$) and space O(s(n)).

HW 6.1: Prove this theorem for the special case k = 2, by showing in detail how a TM M2 with 2 tapes using time t(n) and space s(n) can be simulated by a single-tape TM M1 using time O($t^2(n)$) and space O(s(n)).

Alan Turing defined TMs to work on a tape, rather than on a more conveniently accessible storage, in order to have a conceptually simple model. Notice his phrase ".. the two-dimensional character of paper is no essential of computation". For programming a specific task, on the other hand, a 2-dimensional "tape" is more convenient (Turing: "this paper is divided into squares like a child's arithmetic book"). In order to simplify many examples we also introduce 2-d TMs that work on a 2-dimensional "tape", i.e. on a grid of unbounded size, e.g. the discretized plane or a quadrant of the plane. The motions of the read/write head will then include the 4 directions of the compass, E, N, W, S.

General comments:

An unbounded tape can be viewed in two different ways: 1) At any moment, the tape is finite; when the read/write head moves beyond an extendable tape end, a new square is appended, or 2) the tape is actually infinite. In the second case, the alphabet A contains a designated symbol "blank", which is distinguished from all other symbols: blank is the **only** symbol that occurs infinitely often on the tape. Thus, at any moment, a TM tape can be considered to be finite in length, but the tape may grow beyond any finite bound as the computation proceeds. The graphic appearance of "blank" varies: ' ', or Ø; in the binary alphabet A = {0, 1}, we consider 0 to be the "blank".

TMs are usually interpreted either as computing some function, say f: N -> N from natural numbers to natural numbers, or as accepting a language, i.e. set of strings. For any computation, we must specify the precise format of the input data, e.g. how integer arguments x1, .., xn of a function y = f(x1, .., xn) are written on the tape; and the format of the result y if and when the computation terminates. Integers are typically coded into strings using **binary or unary number representation**. The representation chosen affects the complexity of a computation, but not the feasibility in principle. As later examples show, simple TMs can convert from any radix representation to unary and vice-versa.

A TM that **computes a function** receives its (coded) input as the initial content of its tape[s], and produces its output as the final content of its tape[s], **if it halts**. If it doesn't halt, the function is undefined for this particular value (**partial function**). A TM that always halts computes a **total function**.

A **TM acceptor** has halt states q_a = accept, q_r = reject, with 3 possible answers: **accept, reject, loop**.
A **TM decider** is an acceptor that **always halts**, i.e. always responds with either accept or reject.

The definition of Turing machine is amazingly robust. For example, a 2-d or 3-d "tape" does not increase computational power (Turing: "the two-dimensional character of paper is no essential of computation").
A 2-d infinite array of squares can be linearized according to any space-filling curve, such as a spiral. A TM with an ordinary 1-d tape can simulate the 2-d array by traversing the plane along the spiral, while keeping track of longitude and latitude of its current location in the plane. This insensitivity to the access characteristics of its unbounded storage is in marked contrast to PDAs, where a second stack already increases power. When considering the time and space complexity of a computation, then of course the details of the TM definition matter a great deal.

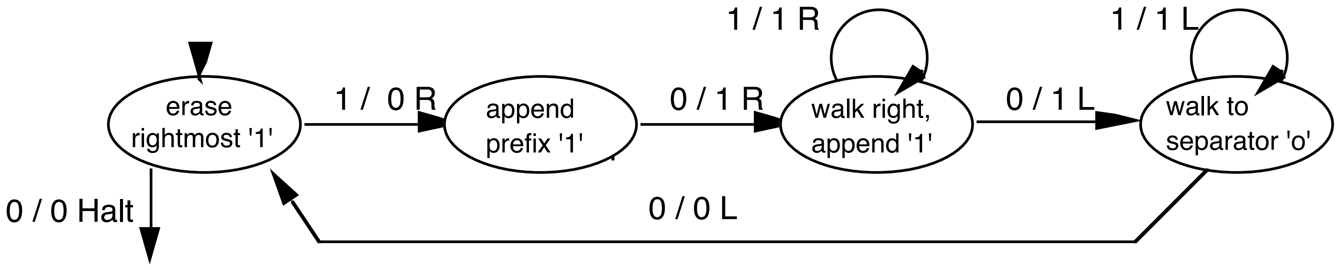
As a consequence of the equivalence of many versions of TMs, we will choose different conventions to simplify specific examples. Only in the case of well-known "TM competitions", such as finding "small" universal TMs or the "Busy Beaver TM", one must adhere to a precise syntax so that competing TMs can be fairly assessed.

Universal TM, UTM: A universal TM U is an interpreter that reads the description <M> of any arbitrary TM M and faithfully executes operations on data D precisely as M does. For single-tape TMs, imagine that <M> is written at the beginning of the tape, followed by D.

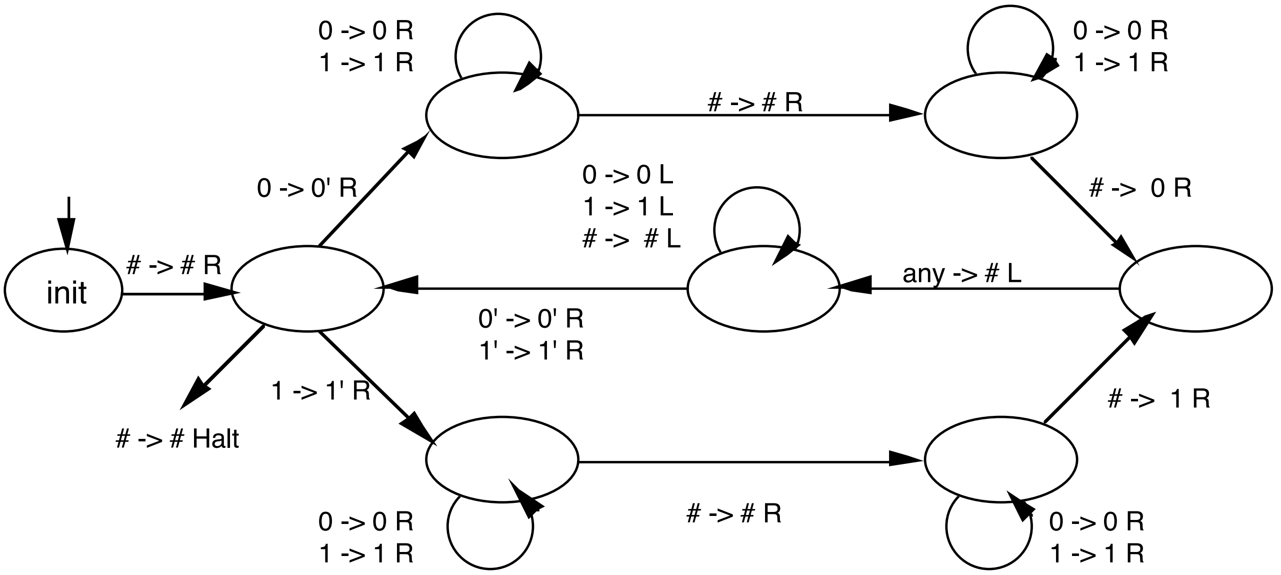
"Minimal TMs":
1) A 2-symbol alphabet A = {0, 1} is enough: code any larger alphabet as bit strings.
2) Claude Shannon: a 2-state fsm controler is enough!! Code "state information" into a large alphabet.
3) Small universal TMs: The complexity of a TM is measured by the "state-symbol product" |Q| x |A|.
There are UTMs with a state-symbol product of less than 30.

6.4 Examples

Ex1: f(x) = 2x in unary notation (x ≥ 1): double the length of a string of contiguous 1's
Configurations. A **bold** symbol indicates it is currently being scanned by the read-write head.
Initial: ..0 1 1 1 **1** 0 .. the read/write head scans the rightmost 1 of a string of contiguous 1s
Intermediate: ..0 1 1 1 0 1 1 0 0 .. there are two strings of contiguous 1's separated by a single 0.
1s have been deleted at left, twice that number have been written at right.
Next step: .. 0 1 1 0 1 1 1 1 0 .. the new string of 1s has been extended by a 1 at each end
Final: ..**0** 1 1 1 1 1 1 1 1 0 .. string of 1s erased, a new string twice as long has been created.



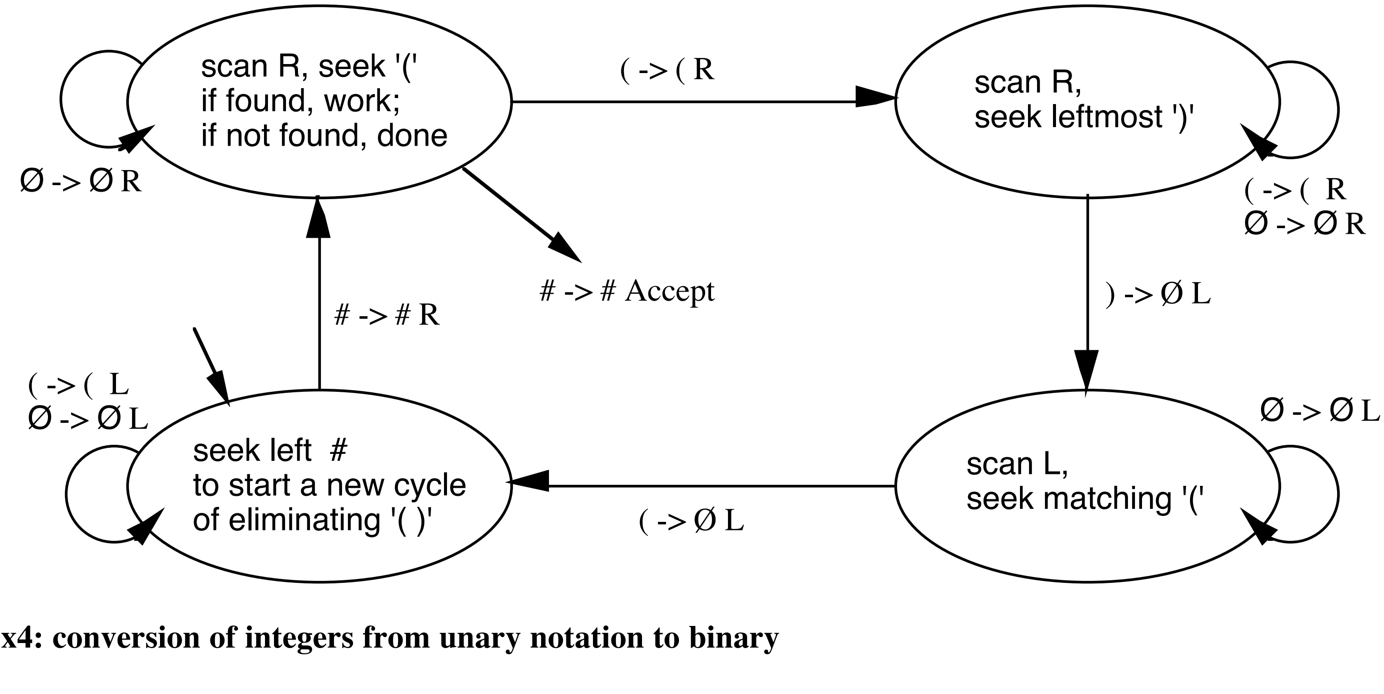
Ex2 duplicate a bit string
Initial: **#** 1 0 1 1 # #
Intermediate: # 1' 0' **1** 1 # 1 0 # (some bits have been replaced by 1' or 0' and have been copied)
Final: **#** 1' 0' 1' 1' **#** 1 0 1 1 # (should be followed by a clean up phase where 1' is changed to 1).



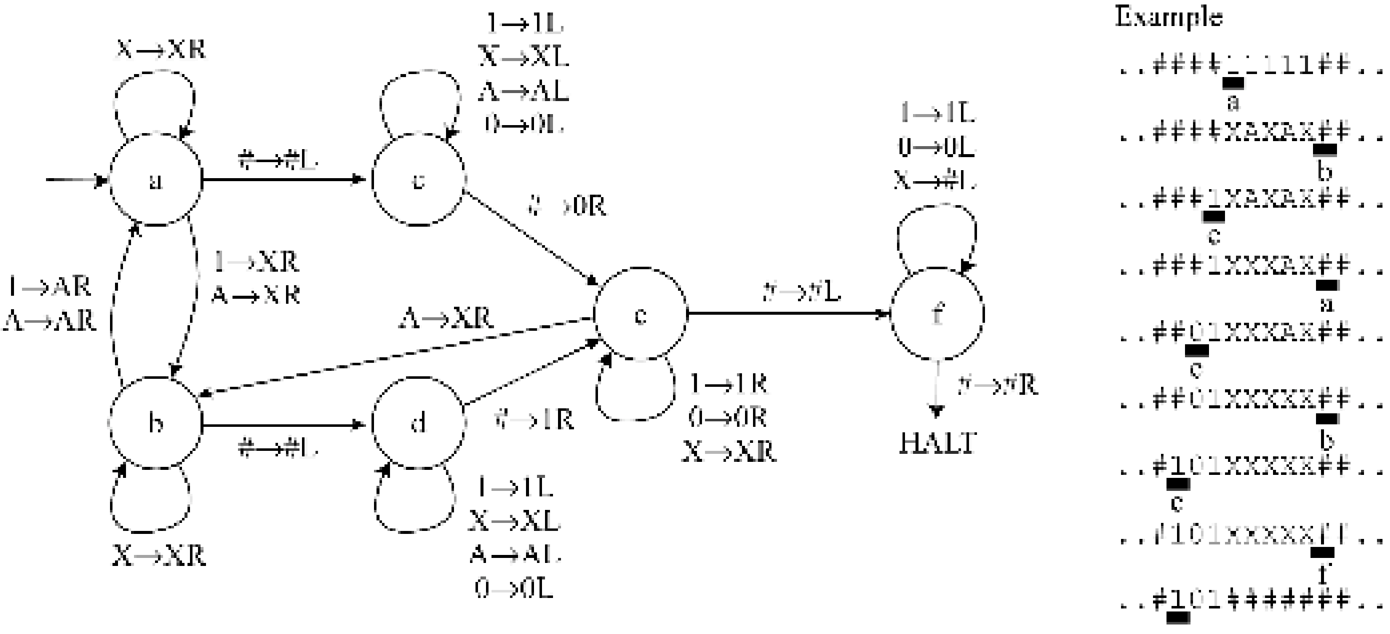
Ex3 parentheses expressions: For a simple task such as checking the syntactic correctness of parentheses expressions, the special purpose device PDA, with its push-pop access to a stack, is more convenient. A PDA throws away a pair of matching parentheses as soon as it has been discovered and never looks at it again. On a tape, on the other hand, "throwing away" leaves a gap that somehow has to be filled, making the task of "memory management" more complicated.

The following TM M with alphabet { (,), Ø, # } successively replaces a pair of matching parentheses by blanks Ø. We assume the string of parentheses is bounded on the left and the right by #, and the read-write head starts scanning the left #. Thereafter, the head runs back and forth across the string, ignoring all blanks that keep gettingcreated. It runs towards the right looking for the leftmost ')' and replaces it by a blank; then runs left looking for the first '(' and replaces it by a blank. As soon as a pair of outermost parentheses has been erased, M returns to the left boundary marker # to start searching another pair of parentheses.

Start configuration: # <parentheses> #, i.e. all parentheses are compressed between the #s, and the r/w head scans the left #. The nullstring is also considered a correct p-expression
Accept configuration: #Ø Ø .. Ø #. Any transition not shown explicitly below leads to a reject state.

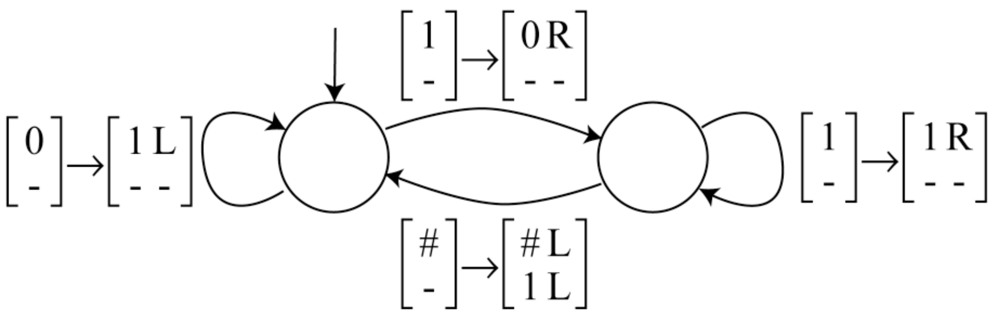


Ex4: conversion of integers from unary notation to binary



Ex5: conversion of integers from binary notation to unary

There are many ways to attack any programming problem. We can analyze a given binary integer .. b2 b1 b0 bit by bit, build up a string of 1s of successive length 1, 2, 4, .. using the doubling TM of Ex1, and catenate those strings of 1s of length 2^k where bk = 1. The 2-state TM below is a candidate for the simplest TM that does the job, thanks to its use of two tapes. The upper tape initially contains the input, a positive integer with most significant bit 1. It is also used as a counter to be decremented past 0 down to #11..1#, a number that can be interpreted as -1. At each decrement, an additional '1' is written on the output tape. The counter is decremented by changing the tail 100..0 to 011..1.



```
 input:  #1001#          #1000#        #0111#              #1111#
output:  ######  →...→  ####1#  →...→ ###11#  →...→  #111111111#
```

Ex6 multiplication in unary notation (z = xy, x ≥ 1, y ≥ 1):

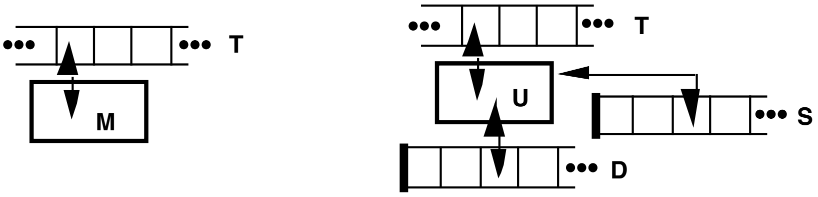
Configurations. A **bold** symbol indicates it is currently being scanned by the read-write head.
Initial: .   ..0 **1** 1 1 X 1 1 0 ..
Intermediate: ..0 0 0 1 X 1 1 1" 1" 1" 1" 0 ..
Intermediate: ..0 0 0 0 X 1 1 1" 1" 1" 1" 1" 1" 0 ..
Final: ..0 0 0 0 X 1 1 1 1 1 1 0 ..
The multiplier x = 3 is to the left of the marker X, the multiplicand y = 2 to the right. Find the leftmost 1 of x, change it to 0, then append a "pseudo-copy" of y to the far right. This "pseudo-copy" consists of as many 1" as y contains 1s, and is produced by a slightly modified copy machine of Ex2 (omit the clean up phase,etc).

6.5 A universal Turing machine

A universal TM U simulates any arbitrary TM M, given its description <M>. The existence of U is often used in proofs of undecidability by saying "TM X simulates TM Y, and if Y halts, does so-and-so". <M> can be considered to be a program for an interpreter U. Naturally, U may be a lot slower than the TM M it simulates, since U has to run back and forth along its tape, finding the appropriate instruction from <M>, then executing it on M's data.

When designing U, we have to specify a code suitable for describing arbitrary TMs. Since U has a fixed alphabet A, whereas arbitrary TMs may have arbitrarily large alphabets, the latter must be coded. We assume this has been done, and the TM M to be simulated is given by
M = (Q, A, f: Q x{0, 1} -> Q x {0, 1} x {L, R, ..}, q_0, ..}.

U can be constructed in many different ways. For simplicity of understanding, we let U have 3 tapes: T, D, S.



U's 3 tapes have the following roles:
1) U's tape T is at all times an exact copy of M's tape T, including the position of the read/write head.

2) D = <M> is a description of M as a sequence of M's tuples, in some code such as #q, a -> q', b, m#. Here q and q' are codes for states in Q. For example, qk ∈ Q may be coded as the binary representation of k. Similarly, m is a code for M's tape actions, e.g. L or R. #, comma, and -> are delimiting markers. In order to make M's tuples intuitively readable to humans, we have introduced more distinct symbols than necessary - a single delimiter, e.g. # is sufficient. Whatever symbols we introduce define an alphabet A'.
In principle, U only needs read-only access to D, but for purposes of matching strings of arbitrary length it may be convenient to have read/write access, and temporarily modify the symbols on D.

3) The third tape S contains the pair (q, a) of M's current state q and the currently scanned symbol a on T. The latter is redundant, because U has this same information on its own copy of T. But having the pair (q, a) together is convenient when matching it against the left-hand side of M's tuples on D.

Thus, U = (P, A2, g: P x A2 -> P x A2 x LR3, p0) looks somewaht complicated. P is U's state space, p0 is U's initial state. A2 = {0, 1} x A' x A' is the alphabet, and LR3 = {L, R, ..} x {L, R, ..}x {L, R, ..} is the set of possible tape actions of this 3-tape machine. U starts in an initial configuration consisting of p0, tapes T, D, S initialized with the proper content, and appropriate head positions on all 3 tapes. The interpreter U has the following main loop:

```
while no halting condition arises do
begin
 1) match the pair (q, a) on S to the left-hand side of a tuple #q, a -> q', b, m# on D
 2) write b onto T, execute the tape action m on T, scan the current symbol c on T
 3) write the string  (q', c) onto S
end.
```

Halting conditions depend on the precise definition of M, such as entering a halting state or executing a halting transition. In summary, a universal TM needs nothing more complex than copying and matching strings.

Designing a universal TM becomes tricky if we aim at "small is beautiful". There is an ongoing competition to design the smallest possible universal TM as measured by the **state-symbol product** |Q| x |A|.
Ex: Yurii Rogozhin: A Universal Turing Machine with 22 States and 2 Symbols, Romanian J. Information Science and Technology, Vol 1, No 3, 259 - 265, 1998.
Abstract. Let UTM(m,n) be the class of universal Turing machines with m states and n symbols. It is known that universal Turing machines exist in the following classes: UTM(24,2), UTM(10,3), UTM(7,4), UTM(5,5), UTM(4,6), UTM(3,10), and UTM(2,18). In this paper it is shown that universal Turing machine exists in the class UTM(22,2), so previous result UTM(24,2) is improved.

6.6 2-dimensional Turing machines

A TM with a 2-d "tape", or a multi-dimensional grid of cells, is no more powerful than a usual TM with a 1-dimensional tape. An unbounded tape has the capacity to store all the info on a grid, and to encode the geometry of the grid in the form of a space filling curve. Obviously, a 2-d layout of the data can speed up computation as compared to a 1-d layout, and the programs are easier to write. We illustrate this with examples of arithmetic. E, N, W, S denote motions of the read-write head to the neighboring square East, North, West, South. We allow multiple motions in a single transition, such as NE for North-East.

Ex: Define a 2-d TM that adds 2 binary integers x, y to produce the sum z. Choose a convenient layout.

Ex: multiplication by doubling and halving: x * y = 2x * (y div 2) [+ x if y is odd], y > 0

Doubling and halving in binary notation is achieved by shifting. In this case, doubling adds a new least significant 0, halving drops the least significant bit. If the bit dropped is a 1, i.e. if y is odd, we have to add a correction term according to the formula x * y = 2x * (y div 2) + x.
Outline of solution 1): The example below shows the multiplication 9 * 26 in decimal and in binary. In binary we ask for y to be written in reversed order: 26 = 11010, 26_{rev} = 01011. The reason is that the change in both x and y occurs at the least significant bit: as x gains a new bit, y loses a bit, and the only changes occur near the # that separates the least significant bits of x and y.

```
  9 * 26       1001 # 01011
 18 * 13      10010 # 1011
 36 *  6     100100 # 011
 72 *  3    1001000 # 11
144 *  1   10010000 # 1
---   --------
234   11101010
```

Solution 2 (courtesy of Michael Breitenstein et al.): **x * y = (x div 2) * 2y [+ y if x is odd], y > 0**
Idea: The multiplication proceeds on two lines: the top line contains the current product x * y, the bottom line the current partial sum. Example 27*7 = 189, intermediate stages shown from left to right.

```
27 * 7     13 * 14    6 * 28     3 * 56     1 * 112
0     7      21     21     77     189
```

Solutions to these problems, and many other examples, can be found in our software system Turing Kara.

6.7 Non-computable functions, undecidable problems

Lots of questions about the behavior of TMs are **undecidable**. This means that there is no algorithm (no TM) that answers such a question about any arbitrary TM M, given its description <M>. The halting problem is the protypical undecidabilty result. Decidability and computability are essentially the same concept - we use "decidable" when the requested answer is binary, "computable" when the result comes from a larger range of values. Thus there are lots of non-computable functions - in fact, almost no functions are computable!

1) Almost nothing is computable

This provocative claim is based on a simple counting argument: there is only a countable infinity of TMs, but there is an uncountable infinity of functions - thus, there are not enough TM to compute all the functions.

Once we have defined our code for describing TMs, each TM M has a description <M> which is a string over some alphabet. Lexicographic ordering of these strings defines a 1-to-1 correspondence between the natural numbers and TMs, thus showing that the set of TMs is countable.

Georg Cantor (1845-1918, founder of set theory) showed that the set of functions from natural numbers to natural numbers is uncountable. Cantor's diagonalization technique is a standard tool used to prove undecidability results. By way of contradiction, assume that all functions from natural numbers to natural numbers have been placed in 1-to-1 correspondence with the natural numbers 1, 2, 3, ..., and thus can be labeled f1, f2, f3, ...

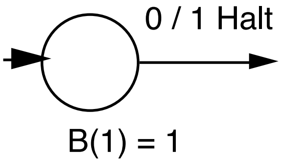
f1:	**f1(1)**	f1(2)	f1(3)	f1(4)	...	f1:	**f1(1) + 1**	f1(2)	f1(3)	f1(4)	...
f2:	f2(1)	**f2(2)**	f2(3)	f2(4)	...	f2:	f2(1)	**f2(2) + 1**	f2(3)	f2(4)	...

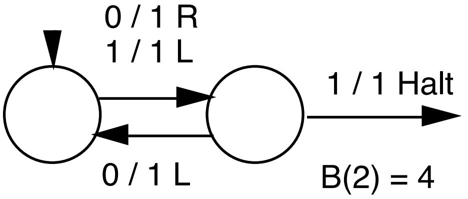
f3: f3(1) f3(2) **f3(3)** f3(4) ... f3: f3(1) f3(2) **f3(3) + 1** f3(4) .
..

Consider the “diagonal function” g(i) = fi(i) shown at left. This could be one of the functions fk - no problem so far. Now consider a “modified diagonal function” h(i) = fi(i) + 1 shown at right. The function h differs from each and every fk, since h(k) = fk(k) + 1 ≠ fk(k). This contradicts the assumption that the set of functions could be enumerated, and shows that there are an uncountable infinity of functions from natural numbers to natural numbers.

It takes more ingenuity to define a specific function which is provably non-computable:

2) The Busy Beaver problem T. Rado: On non-computable functions, Bell Sys. Tech. J. 41, 877-884, 1962
Given n > 0, consider n-state TMs M = (Q, {0, 1}, f, q1) that start with a blank tape. The “Busy Beaver function” B(n) is defined as the largest number of 1s an n-state TM can write and still stop. The precise value of B(n) depends on the details of “Turing machine syntax”. We code halting as an action in a transition. The following examples prove B(1) = 1 and B(2) = 4 by exhaustive analysis.





0 0 0 **0** 0 0
0 0 0 1 **0** 0
0 0 0 **1** 1 0
0 0 **0** 1 1 0
0 **0** 1 1 1 0
0 1 **1** 1 1 0

Lemma: B(x) is total, i.e. defined for all x ≥ 1 (obviously), and **monotonic**, i.e. B(x) < B(x+1) for all x ≥ 1.
Pf: Consider a TM M with x states that achieves the maximum score B(x) of writing B(x) 1s before halting. Starting in its initial state on a blank tape, M will trace some path thru its state space, finishing with a transition of the form q, a -> -, 1, Halt. The dash “-” stands for the convention that a halting transition leads “nowhere”, i.e. we need no halting state. Writing a 1 before halting cannot hurt. M can be used to construct M’ with x+1 states that writes an additional 1 on the tape, as follows. The state space of M’ consists of the state space of M, an additional state q’, and the following transitions: M’s transition q, a -> -, 1, Halt is modified to q, a -> q’, 1, R. Two new transitions: q’, 1 -> q’, 1, R and q’, 0 -> -, 1, Halt cause the read/write head to skip over the 1s to its right until it finds a 0. This 0 is changed to 1 just before halting, and thus M’ writes B(x)+1 “1s”.

Thm (Rado, 1962): B(x) is not computable. Moreover, for any Turing computable (total recursive) function f: N -> N, where N = {1, 2, ..}, f(x) < B(x) for all sufficiently large x.

Like most impossibility results (there is no TM that computes B), the reasoning involved is not intuitively obvious, so we give two proofs. When discussing TMs that compute a function from integers to integers, we assume all integers x are written in some (reasonable) notation <x>. Typically, <x> is in binary, bin(x), or in unary notation u(x), i.e. a string of x 1s.

Pf1 B() is not computable: Assume B(x) is computable, i.e. there is a TM which, when started on a tape initialized with u(x), halts with u(f(x)) on its tape. If B(x) is computable, so is D(x) = B(2x). Let M be a TM that computes D, let k be the number of states of M.
For each value of x, let Nx be a TM that starts on a blank tape, writes u(x), and halts. Nx can be implemented using x states (fewer states suffice, but we don’t need that).
Now consider TM Mx, a combination of Nx and M with x + k states. Nx starts on a blank tape and writes u(x). Instead of halting, Nx passes control to M, which proceeds to compute D(x). Thus, Mx starts on a blank tape and halts with u(D(x)) = (a string of B(2x) 1s). Since Mx, with x + k states, is a candidate in the Busy Beaver competition that produces B(2x) 1s, and by definition of B(), we have B(x + k) ≥ B(2x). For values of x > k, thanks to the monotonicity of B(), we have B(x + k) < B(2x). These two inequalities lead to the contradiction B(2x) ≤ B(x + k) < B(2x) for all x > k, thus proving B() non-computable.

Pf2 B() grows faster than any computable function: Let f: N -> N be any Turing computable function, and let M be a TM with k states that computes f. We hope that a fast-growing function that writes its result f(x) in unary can be turned into a serious competitor in the Busy Beaver race. Instead, we will conclude that f is a slow-growing function as compared to the Busy Beaver function B(). More precisely, we ask: “for what values of x might a modified version of M that produces f(x) in unary notation be a strong Busy Beaver competitor?” We will conclude that this might be the case for a finite number of “small” values of x, but M has no chance for alles values of x beyond some fixed x0. The technical details of this argument follow.

For each value of x, let Nx be a TM that starts on a blank tape, writes u(x), and halts. Nx can be implemented with $\lceil \log x \rceil + c$ states, for some constant c, as follows: Nx first writes bin(x), a string of $\lceil \log x \rceil$ bits, using $\lceil \log x \rceil$ initialization states. Then Nx invokes the binary to unary converter BU, a TM with c states.

Now consider TM Mx, a combination of Nx and M with $\lceil \log x \rceil + c + k$ states. Nx starts on a blank tape, writes u(x), then passes control to M, which proceeds to compute f(x) in unary notation, i.e. produces f(x) 1s.

Mx, with $\lceil \log x \rceil + c + k$ states, is an entry in the Busy Beaver competition with score f(x), hence $f(x) \leq B(\lceil \log x \rceil + c + k)$. Since $\lceil \log x \rceil + c + k < x$ for all sufficently large x, and due to the monotonicity of B() proved in the lemma, $B(\lceil \log x \rceil + c + k) < B(x)$, and hence $f(x) < B(x)$ for all sufficently large x. QED

It is important to distinguish the theoretical concept “not computable” from the pragmatic “we don’t know how to compute it”, or even the stronger version “we will never know how to compute it”. Example: is the function f(n) = min [B(n), 109999 } computable or not? Yes, it is. Since B() is monotonic, f() first grows up to some argument m, then remains constant: B(1) < B(2) < B(3) < ... < B(m) ≤ 109999, 109999, ... A huge but conceptually trivial TM can store the first m function values and the constant 109999, and print the right answer f(n) given the argument n. There are at least two reasons for arguing that “we will never be able to compute f(n)”. First, the numbers involved are unmanageable. Second, even if we could enumerate all the Busy Beaver competitors until we find one, say Mk with k states, that prints at least 109999 1s; we could never be sure whether we missed a stronger competitor Mc with c < k states that also prints at least 109999 1s and halts. Among the TMs with c states, there might have been some that printed more than 109999 1s, but we dismissed them as not halting. And although it is possible to prove some specific TM to be halting or non-halting, there is no general algorithm for doing so. Hence, there will be some TMs whose halting status remains unclear.
Don’t confuse the pragmatic “will never be able to compute x” with the theoretical “ x is not computable”.

2) The halting problem: An impossible program (C. Strachey in The Computer J.)

Sir, A well-known piece of folklore among programmers holds that it is impossible to write a program which can examine any other program and tell, in every case, if it will terminate or get into a closed loop when it is run. I have never actually seen a proof of this in print, and though Alan Turing once gave me a verbal proof (... in 1953), I unfortunately and promptly forgot the details. This left me with an uneasy feeling that the proof must be long or complicated, but in fact it is so short and simple that it may be of interest to casual readers. The version below uses CPL, but not in any essential way.

Suppose T[R] is a Boolean function taking a routine (or program) R with no formal or free variables as its argument and that for all R, T[R] = True if R terminates if run and that T[R] = False if R does not terminate. Consider the routine P defined as follows:

```
        rec routine P
               §L:  if T[P] go to L
                       Return §
```

If T[P] = True the routine P will loop, and it will only terminate if T[P] = False. In each case T[P] has exactly the wrong value, and this contradiction shows that the function T cannot exist. Yours faithfully, C. Strachey

Concise formulation of the same argument:
Assume T(P) as postulated above. Consider Cantor: if T(Cantor) then loop end. Will Cantor halt or loop?

Hw 6.2: Surf the web in search of small universal TMs and Busy Beaver results. Report the most interesting findings, along with their URL.
Hw 6.3: Investigate the values of the Busy Beaver function B(2), B(3), B(4). A.K. Dewdney: The Turing Omnibus, Computer Science Press, 1989, Ch 36: Noncomputable functions, 241-244, asserts:
B(1) = 1, B(2) = 4, B(3) = 6, B(4) = 13, B(5) ≥ 1915. Have you found any new records in the competition to design Turing machines that write a large number of 1s and stop?

6.8 Turing machine acceptors and decidability

Deterministic TM: M = (Q, A, f: Q x A -> Q x A x {L, R}, q0, qa , qr }.
Tape actions: L = move left, R = move right.

Designated states: q_0 = start state, q_a = accept, q_r = reject. q_a and q_r are **halting states.**
A Turing machine acceptor has **3 possible answers: accept, reject, loop**.

Df: M accepts $w \in A^*$ iff M's computation on w ends in q_a . $L(M) = \{ w \in A^* / M \text{ accepts } w \}$
Df: $L \subseteq A^*$ is **Turing recognizable** (recursively enumerable, semi-decidable) iff
there is a TM M with L = L(M).
Df: A TM acceptor M that always halts (in one of its states q_a or q_r) is called a **decider** for its language L(M).
Df: $L \subseteq A^*$ is **Turing decidable** (recursive) iff there is a TM decider M with L = L(M).
Notice: Any TM acceptor M is a recognizer for its language L(M), but only some of them are deciders.

Thm: L is Turing decidable iff both L and its complement ¬L are Turing recognizable

Pf ->: If M is a decider for L, then M' with $q'_a = q_r$ and $q'_r = q_a$ is a decider for ¬L.
Since deciders are also recognizers, both L and ¬L are Turing recognizable.

Pf <-: Let M be a recognizer for L and ¬M a recognizer for ¬L. Construct a decider Z ("zigzag") for L as follows. Z has 3 tapes: input tape with w on it, tape T (M's tape), and tape ¬T (¬M's tape).
Z on input w alternates simulating a transition of M using T, and a transition of ¬M using ¬T. One of M and ¬M will accept w and halt, and at this moment Z halts also, announcing the verdict "accept" or "reject" depending on which recognizer accepted w. QED

The **word problem** for an automaton M or language L (of any kind): given M and $w \in A^*$, does M accept w? given L and $w \in A^*$, is $w \in L$? Denote the word problem and their corresponding languages) for FAs, PDAs, and TMs by WFA, WPDA, WTM, respectively.

Ex: WDFA = { <M, w> | M is a DFA , M accepts w } is **decidable**. A TM simulates the action of M on w.

Ex: WNFA = { <M, w> | M is a NFA , M accepts w } is **decidable**.
A TM traces all legal paths labeled w through M's state space.

Ex: WNPDA = { <M, w> | M is a NDPA , M accepts w } is **decidable**. A TM that merely simulates the action of M on w is no decider, since a PDA can loop, and thus, simulation may never terminate. However, we can convert an NPDA into an equivalent CFG G (e.g. in Chomsky normal form) and test all derivations that produce words of length |w|.

Thm (the word problem for TMs is undecidable):
WTM = { <M, w> | M is a TM , M accepts w } is **not** decidable.
Pf: Assume WTM is decidable, and D is a decider for WTM , i.e.
D(<M, w>) = accept, if M accepts w, and
D(<M, w>) = reject, if M rejects w or M loops on w.
From this assumption we will derive a contradiction using a technique that is standard in logic, set theory, and computability. Before proceeding, consider some explanatory historical comments.

Many undecidability proofs proceed by self-reference, e.g. by having an assumed TM (or algorithm in another notation, as we had seen in the halting problem) examine itself. In order for self-reference to yield a proof of impossibility, we have to craft the core of a contradiction into the argument. Such contradictions lead to well-known ancient paradoxes such as "the Cretan Epimenides who states that Cretans always lie", or "the barber who shaves all people that do not shave themselves". Is Epimenides lying or telling the truth? Does the barber shave himself or not?

The type of self-reference that leads to paradoxes was formalized by Cantor in the form of his "diagonalization technique", used to prove many impossibility or non-existence results. Why the term "diagonalization"? In terms of the example above, consider all pairs <M, w> arranged as a doubly infinite array, with TM M as rows, and tapes or words w as columns. The description <M> of TM M is also a word over the aphabet we are considering. Thus, some of the w's are also <M>'s, and it is natural to call the pairs <M, <M>> the "diagonal" of the array. In the case of WTM , diagonal entries mean "M is a TM that accepts its own description <M>". Nothing suspicious so far, just as there is no paradox if Epimenides says "Cretans always tell the truth" (this statement might be untrue, but it is consistent - in an ideal world, it might be true). Deriving a contradiction requires devious cleverness. By analogy with the paradoxes, we construct TM Cantor which derives a contradiction from the assumed existence of a decider D for WTM .

Cantor(<M>) interprets its input as the description <M> of a TM M according to some fixed coding scheme. If the input is not the description of any TM, Cantor halts with the message "syntax error". On a syntactically correct input <M>, Cantor simulates D on input <M, <M>>, deciding whether M will halt if fed its own description. After having heard D's answer, Cantor contrives to state the opposite:
If D(<M, <M>>) accepts, Cantor(<M>) rejects; If D(<M, <M>>) rejects, Cantor(<M>) accepts.
This holds for all M, so consider the special case M = Cantor. We find that Cantor (<Cantor>) accepts, if Cantor (<Cantor>) rejects, and vice versa. This contradiction in Cantor's behavior forces us to reject the weakest link in the argument, namely, the unsupported assumption that a decider D exists for WTM. QED

In section 6.6 we presented a proof of the undecidability of the halting problem that used the same diagonalization technique as above. Now we prove the same result by "problem reduction".

Thm: $HALT_{TM}$ = { <M, w> | M is a TM that halts on input w } is **not** decidable.
Pf: Assume $HALT_{TM}$ is decidable using a decider H, i.e. H(<M, w>) halts, accepting or rejecting depending on whether M halts on w, or loops. Construct TM R ("reduce"), a decider for WTM , as follows:
1) R(<M, w>) simulates H(<M, w>), a process that halts under the assumption that the decider H exists.
2a) if H rejects <M, w>, we know M(w) loops, so R rejects <M, w>;
2b) if H accepts <M, w>, we know M(w) will halt, so R simulates M(w) and reports the result, accept or reject.
Thus, the existence of H implies the existence of R, a decider for the word problem WTM , a problem we have just proven to be undecidable. Contradiction -> QED.

Ex: Prove that $REGULAR_{TM}$ = { M | M is a Turing machine and L(M) is regular } is undecidable.
Pf: Assume there is a TM R that decides $REGULAR_{TM}$. From R we derive a TM W that decides the "word problem" WTM = { <M, w> | M accepts w }, which has been proven to be undecidable -> contradiction.
First, consider a 2-d family of TMs $N_{M, w}$, where M ranges over all TMs, in some order, and w ranges over all words ∈ A*. These artificially created TMs $N_{M, w}$ will never be executed, they are mere "Gedanken-experiments". Some $N_{M, w}$ accept the non-regular language 0^n1^n, n ≥ 1, the other $N_{M, w}$ accept the regular language A*. Thus, the assumed TM R that decides $REGULAR_{TM}$ can analyze any $N_{M, w}$ and tell whether it accepts the non-regular language L($N_{M, w}$) = { 0^n1^n} or the regular language L($N_{M, w}$) = A*. The goal of the proof is to arrange things such that L($N_{M, w}$) is regular or irregular depending on whether M accepts w or rejects w. If this can be arranged, then TM R that decides $REGULAR_{TM}$ can indirectly decide the "word problem" WTM , and we have the desired contradiction to the assumption of R's existence.

So let us construct $N_{M, w}$. $N_{M, w}$ (x) first does a syntax check on x, then proceeds as follows:
a) if x = 0^n1^n, for some n ≥ 1, $N_{M, w}$ accepts x.
b) if x ≠ 0^n1^n, for any n ≥ 1, $N_{M, w}$ simulates M on w. This relies on $N_{M, w}$ containing a universal TM that can simulate any TM, given its description. This simulation of M on w can have 3 outcomes: M accepts w, M rejects w, or M loops on w. $N_{M, w}$'s action in each of these case is as follows:
M accepts w: $N_{M, w}$ accepts x. Notice: in this case $N_{M, w}$ accepts all x ∈ A*, i.e. L($N_{M, w}$) = A*.
M rejects w: $N_{M, w}$ rejects x. Notice: in this case $N_{M, w}$ accepts ONLY words of the form 0^n1^n by rule a).
M loops on w: $N_{M, w}$ loops. Notice: $N_{M, w}$ has no other choice. If M loops, $N_{M, w}$ never regains control.
The net effect is that L($N_{M, w}$) is regular iff M accepts w, and irregular iff M rejects w. The assumed TM R that decides $REGULAR_{TM}$, given a description < $N_{M, w}$>, indirectly decides whether M accepts w.
In order to complete the proof, we observe that it is straightforward to construct a TM W that reads <M, w>, from this constructs a description < $N_{M, w}$>, and finally calls R to decide the word problem, known to be undecidable. Contradiction -> QED.

Hw 6.4: Decidable and recognizable languages

Recall the definitions: A language L over an alphabet A is **decidable** iff there is a Turing machine M(L) that accepts the strings in L and rejects the strings in the complement A* - L. A language L is **recognizable**, or recursively enumerable, iff there is a Turing machine M(L) that accepts all and only the strings in L.

a) Explain the difference between decidable and recognizable. Define a language L1 that is decidable, and a language L2 that is recognizable but not decidable.

b) Show that the class of decidable languages is closed under catenation, star, union, intersection and complement.

c) Show that the class of recognizable languages is closed under catenation, star, union, and intersection, but not under complement.

d) Assume you are given a Turing machine M that recognizes L, and a Turing machine M’ that recognizes the complement A* - L. Explain how you can construct a Turing machine D that decides L.

6.9 On computable numbers, ...

L. Kronecker (1823-1896): “God made the integers; all the rest is the work of man.”

Paradox: “the smallest integer whose description requires more than ten words.”

G. Peano (1858-1932): axioms for the natural numbers:
1) 1 is a number. 2) To every number a there corresponds a unique number a’, called its successor.
3) If a’ = b’ then a = b. 4) For every number a, a’ ≠ 1.
5) (Induction) Let A(x) be a proposition containing the variable x. If A(1) holds and if, for every number n, A(n’) follows from A(n), then A(x) holds for every number x.

Can each and every number be “described” in some way? Certainly not! Any notation we invent, spoken or written, will be made up of a finite number of atomic “symbols” of some kind (e.g. phonemes). As Turing argued: “If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent”, and we could not reliably distinguish them. In turn, any one description is a structure (e.g. a sequence) consisting of finitely many symbols, therefore the set of all descriptions is countable. But the set of real numbers is not countable, as Cantor proved with his diagonalization technique. Thus, it is interesting to ask what numbers are “describable”. Although this obviously depends on the notation chosen, the Church-Turing thesis asserts that there is a preferred concept of “describable in the sense of effectively computable”.

Turing computable real numbers: Fix any base, e.g. binary. For simplicity’s sake, consider reals x, 0 ≤ x ≤ 1.
Df 1: A real number x is computable iff there is a TM M which, when started on a blank tape, prints the digits in sequence, starting with the most significant digit.

For 0 ≤ x ≤ 1, TM M must prints the digits b1 b2 .. of x = .b1 b2 .. in sequence, starting with the most significant digit of weight 2^{-1}. The alphabet A includes the symbols ‘blank’, 0, 1 and others.
Example: A single-state TM with the tuple (q, blank -> q, 1, R) computes x = 1 = .11111..: .

Since these TMs do not halt, we assume the following convention: When M has written the k-th digit bk, that digit must never be changed - e.g. by writing the digits on a one-way output tape separate from the work tape.

Equivalent Df 2: A real number x is computable iff there is a TM M which, started on a tape initialized with (the representation of) an integer k ≥ 1, prints the k-th digit bk of x and halts.

Hw 6.5: Prove that the two definitions of Turing computable are equivalent. Prove that any rational number x = n/m is computable by outlining a non-halting TM that prints the digits in sequence, and a halting TM that prints k-th digit bk and halts, given k.

Because the computable numbers are countable, whereas the real numbers have the cardinality of the continuum, almost all real numbers are non-computable. It is straightforward to define **a specific non-computable number** x by relating x to some known undecidable problem. We know that the halting problem for TMs that start on a blank tape is undecidable. Let M1, M2, .. be an enumeration of all the TMs over some fixed alphabet. Define the k-th digit bk of x = .b1 b2 ...: bk = 0 if Mk loops, bk = 1 if Mk halts. If x were computable, the TM M that computes x would also decide this version of the halting problem -> contradiction.

Df: An infinite sequence x1, x2, .. of real numbers is enumerable iff there is a TM M that, given a representation of an integer k ≥ 1, prints the digits of xk.

Thm: The computable numbers, while countable, cannot be effectively enumerated!

Pf: Assume there exists TM M that enumerates **all** computable real numbers. By Cantor’s diagonalization technique, we construct a new TM M’’ that computes a new computable number y, y ≠ xk for all k. This contradiction proves the non-existence of M. Consider the infinite array of digits
x1 = x11 x12 x13 ..
x2 = x21 x22 x23 ..

x3 = x31 x32 x33 ..
..

Modify M to obtain a TM M’ which, given k, prints the digit xkk. This merely involves computing the digits of xk in sequence as M does, throwing away the first k-1 digits and stopping after printing xkk. Now modify M’ to obtain M’’ which prints the diagonal sequence of digits x11 x22 x33 ..... Finally, modify M’’ to obtain M’’’ which prints the **sequence of complements of the bits xkk**. This sequence represents a new computable number y which differs from all xk - contradiction, QED.

END Ch6