1. Models of computation

jn 2003.04.03

Goals: An intuitive appreciation of the importance of the concept ‘model of computation’. Acquaintance with several interesting examples that mirror key aspects of realistic systems in the simplest possible setting.

1.0 Models of computation: purpose and types

Almost any statement about computation is true in some model, and false in others!
A rigorous definition of a model of computation is essential when proving negative results: “impossible...”.
Special purpose models vs. universal models of computation (can simulate any other model).

Algorithm and *computability* are originally intuitive concepts. They can remain intuitive as long as we only want to show that some specific result can be computed by following a specific algorithm. Almost always an informal explanation suffices to convince someone with the requisite background that a given algorithm computes a specified result. Everything changes if we wish to show that a desired result is *not computable*. The question arises immediately: "What tools are we allowed to use?" Everything is computable with the help of an oracle that knows the answers to all questions. The attempt to prove negative results about the nonexistence of certain algorithms forces us to agree on a rigorous definition of *algorithm*.

Mathematics has long investigated problems of the type: what kind of objects can be constructed, what kind of results can be computed using a **limited set of primitives and operations**. For example, the question of what polynomial equations can be solved using radicals, i.e. using addition, subtraction, multiplication, division, and the extraction of roots, has kept mathematicians busy for centuries. It was solved by Niels Henrik Abel (1802 - 1829) in 1826: The roots of polynomials of degree ≤ 4 can be expressed in terms of radicals, those of polynomials of degree ≥ 5 cannot, in general. On a similar note, we briefly present the historically famous problem of geometric construction using ruler and compass, and show how “tiny” changes in the assumptions can drastically change the resulting set of objects.

Whenever the tools allowed are restricted to the extent that “intuitively computable” objects cannot be obtained using these tools alone, we speak of a **special-purpose model of computation**. Such models are of great practical interest because the tools allowed are tailored to the specific characteristics of the objects to be computed, and hence are efficient for this purpose. We present several examples close to hardware design.

From the theoretical point of view, however, **universal models of computation** are of prime interest. This concept arose from the natural question "What can be computed by an algorithm, and what cannot?". It was studied during the 1930s by Emil Post (1897–1954), Alan Turing (1912–1954), Alonzo Church (1903-1995), and other logicians. They defined various formal models of computation, such as production systems, Turing machines, recursive functions, and the lambda calculus, to capture the intuitive concept of "computation by the application of precise rules". All these different formal models of computation turned out to be equivalent. This fact greatly strengthens **Church's thesis** that the intuitive concept of algorithm is formalized correctly by any one of these mathematical systems. The models of computation defined by these logicians in the 1930s are **universal** in the sense that they can compute anything computable by any other model, given unbounded resources of time and memory. This concept of universal model of computation is a product of the 20-th century which lies at the center of the theory of computation.

The standard universal models of computation were designed to be conceptually simple: Their primitive operations are chosen to be as weak as possible, as long as they retain their property of being universal computing systems in the sense that they can simulate any computation performed on any other machine. It usually comes as a surprise to novices that the set of primitives of a universal computing machine can be so simple, as long as these machines possess two essential ingredients: *unbounded memory* and *unbounded time*.

In this introductory chapter we present 2 examples of universal models of computation:
- Markov algorithms, which access data and instructions in a sequential manner, and might be the architecture of choice for computers that have only sequential memory.

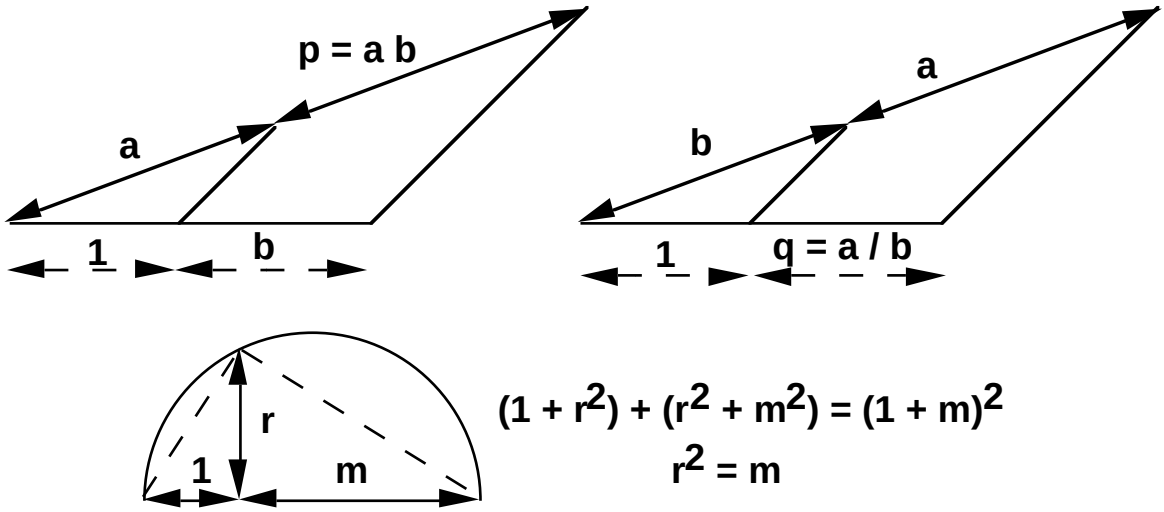
- A simple random access machines (RAM), based on a random access memory, whose architecture follows the von Neumann design of stored program computers.

Once one has learned to program basic data manipulation operations, such as moving and copying data and pattern matching, the claim that these primitive “computers” are universal becomes believable. Most simulations of a powerful computer on a simple one share three characteristics: It is straightforward in principle, it involves laborious coding in practice, and it explodes the space and time requirements of a computation. The weakness of the primitives, desirable from a theoretical point of view, has the consequence that as simple an operation as integer addition becomes an exercise in programming. The purpose of these examples is to support the idea that conceptually simple models of computation are as powerful, in theory, as much more complex models, such as a high-level programming language.

The theory of computability was developed in the 1930s, and greatly expanded in the 1950s and 1960s. Its basic ideas have become part of the foundation that any computer scientist is expected to know. But computability theory is not directly useful. It is based on the concept "computable in principle" but offers no concept of a "feasible in practice". And feasibility, rather than "possible in principle", is the touchstone of computer science. Since the 1960s, a theory of the complexity of computation has being developed, with the goal of partitioning the range of computability into complexity classes according to time and space requirements. This theory is still in full development and breaking new ground, in particular with (as yet) exotic models of computation such as quantum computing or DNA computing.

1.1 Construction with ruler and compass

- Starting with angles of 180° and 60°, construct additional angles by bisection, addition and subtraction.
- Starting with a segment of unit length, construct additional lengths by bisection, addition, subtraction, multiplication, division and square root.



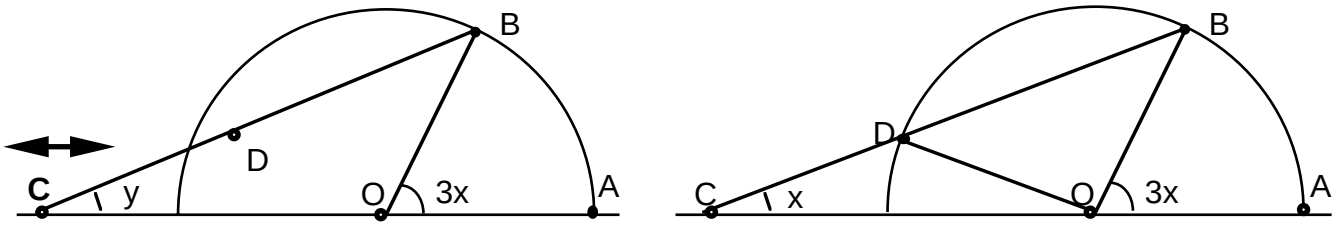
Thm: A proposed construction is possible by ruler and compass **iff** the numbers which define analytically the desired geometric elements can be derived from those defining the given elements by a finite number of rational operations and extractions of real square roots.

Possible construction: Regular polygon of 17 sides (C.F. Gauss)
Impossible constructions: Doubling a unit cube (solve $x^3 = 2$). Trisection of an angle. Squaring a circle.

Thm: There is no algorithm to trisect an arbitrary angle with ruler and compass.

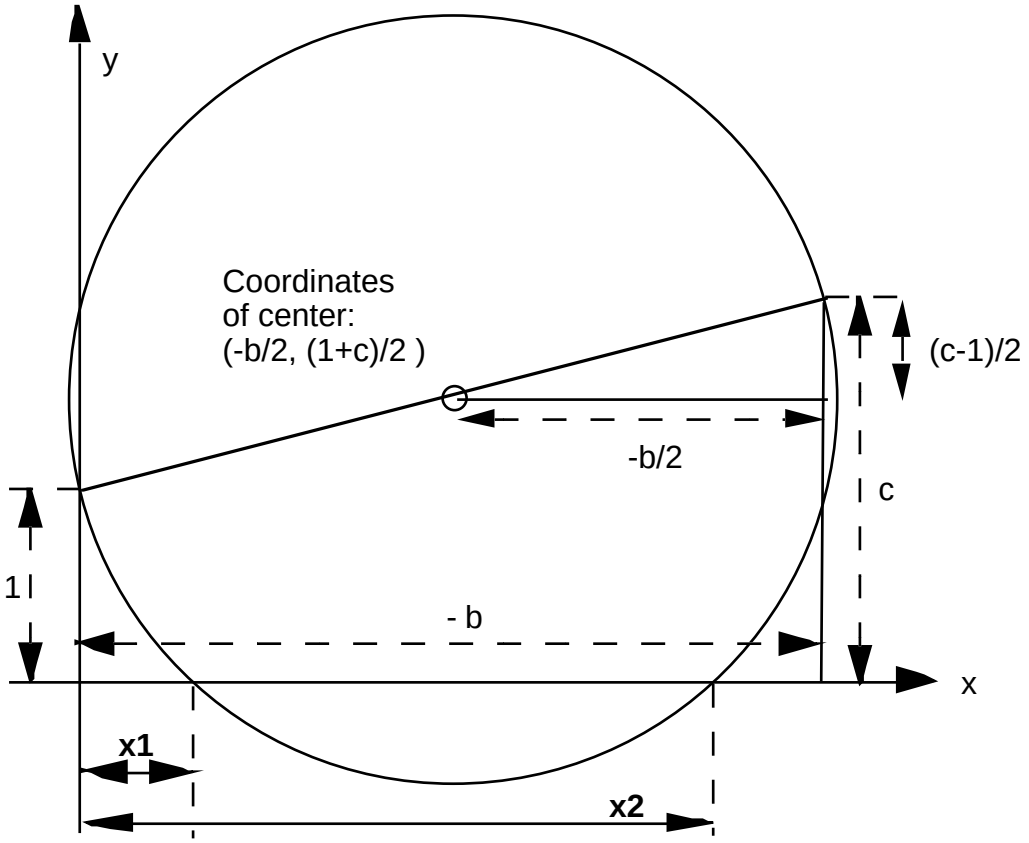
... and yet, Archimedes (~287 B.C. - ~ 212 B.C) found a procedure to trisect an arbitrary angle:

Given angle AOB = 3x in a unit circle. Ruler CB has a mark D at distance CD = radius = 1.
Slide C along the x-axis until D lies on the circle. Then, angle ACB = x.



Msg: “minor” changes in a model of computation may have drastic consequences - precise definition needed!

Hw 1.1: The quadratic equation $x^2 + bx + c = 0$ has roots x1, x2 = (-b ± sqrt(b^2 - 4c)) / 2. Prove that in the ruler and compass construction shown below, the segments x1 and x2 are the solutions of this equation.

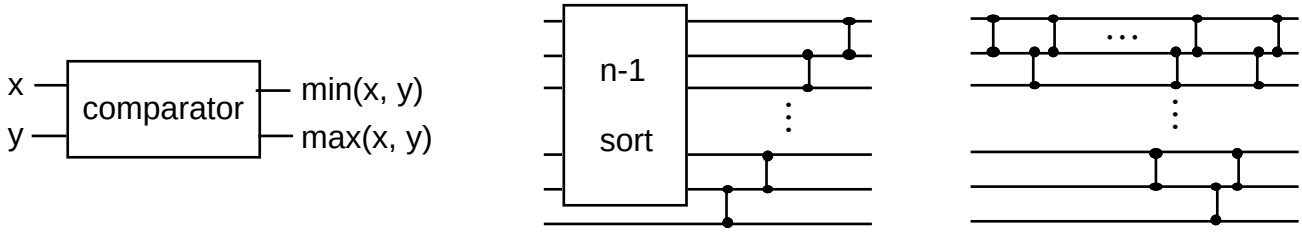


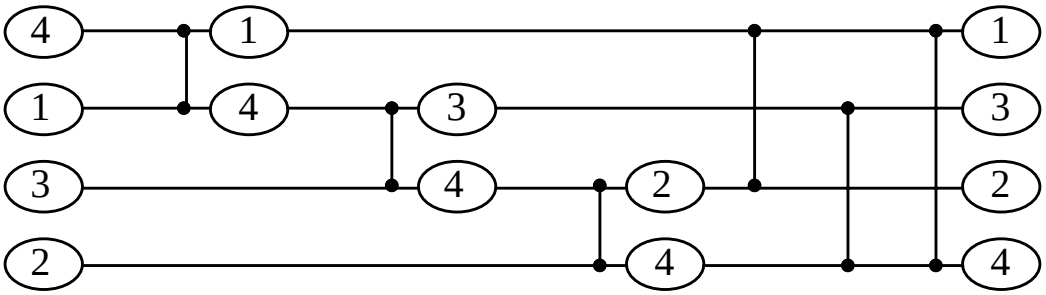
Solution 1.1: Substitute $x_c = - b/2$ and $y_c = (1 + c) / 2$ in the circle equation $(x - x_c)^2 + (y - y_c)^2$, set y = 0 to obtain the quadratic equation $x^2 + bx + c = 0$ whose roots are the desired values x_1 and x_2 .

1.2 Systolic algorithms, e.g. sorting networks

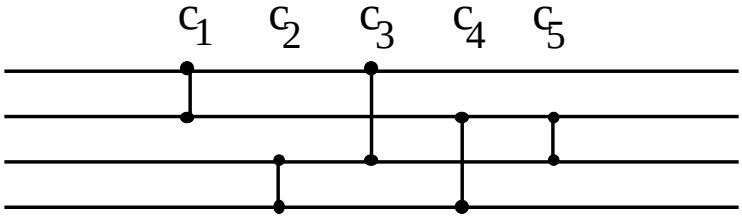
See D.E. Knuth: The art of computer programming, Vol.3 Sorting and searching, Addison-Wesley, 1973.

We consider networks made up of parallel wires on which numbers travel from left to right. At places indicated by a vertical line, a comparator gate guides the smaller (lighter) number to the upper output wire, and the larger (heavier) number to the lower output wire. Clearly, enough comparators in the right places will cause the network to sort correctly - but how many gates does it take, where should they be placed?





6 ill-placed comparators fail to sort, whereas ...



5 comparators suffice to sort 4 inputs

Lemma: Given f monotonic, i.e. $x \le y \Rightarrow f(x) \le f(y)$.
If a **network of comparators** transforms $\mathbf{x} = x_1, .., x_n$ into $\mathbf{y} = y_1, .., y_n$, then it transforms f(**x**) into f(**y**).

Thm (0-1 principle): If a network S with n input lines sorts all 2^n vectors of 0s and 1s into non-decreasing order, S sorts any vector of n arbitrary numbers correctly.

Proof by contraposition: If S fails to sort some vector $\mathbf{x} = x_1, .., x_n$ of arbitrary numbers, it also fails to sort some binary vector f(**x**). Let S transform **x** into **y** with a "sorting error", i.e. $y_i > y_{i+1}$ for some index i. Construct a binary-valued monotonic function f: $f(x) = 0$ for $x < y_i$, $f(x) = 1$ for $x \ge y_i$. S transforms f(**x**) = $f(x_1), .., f(x_n)$ into f(**y**)= $f(y_1), .., f(y_n)$. Since $f(y_i) = 1 > f(y_{i+1}) = 0$, S fails to sort the binary vector **x**. QED.

Thm (testing proves correctness!): If a sorting network S that uses **adjacent comparisons only** sorts the "inverse vector" $x_1 > x_2 > ... > x_n$, it sorts any arbitrary vector.

Msg: Serial and parallel computation lead to entirely different resource characteristics.

Hw1.2: Prove: a sorting network using only **adjacent comparisons** must have ≥ n-choose-2 comparators.

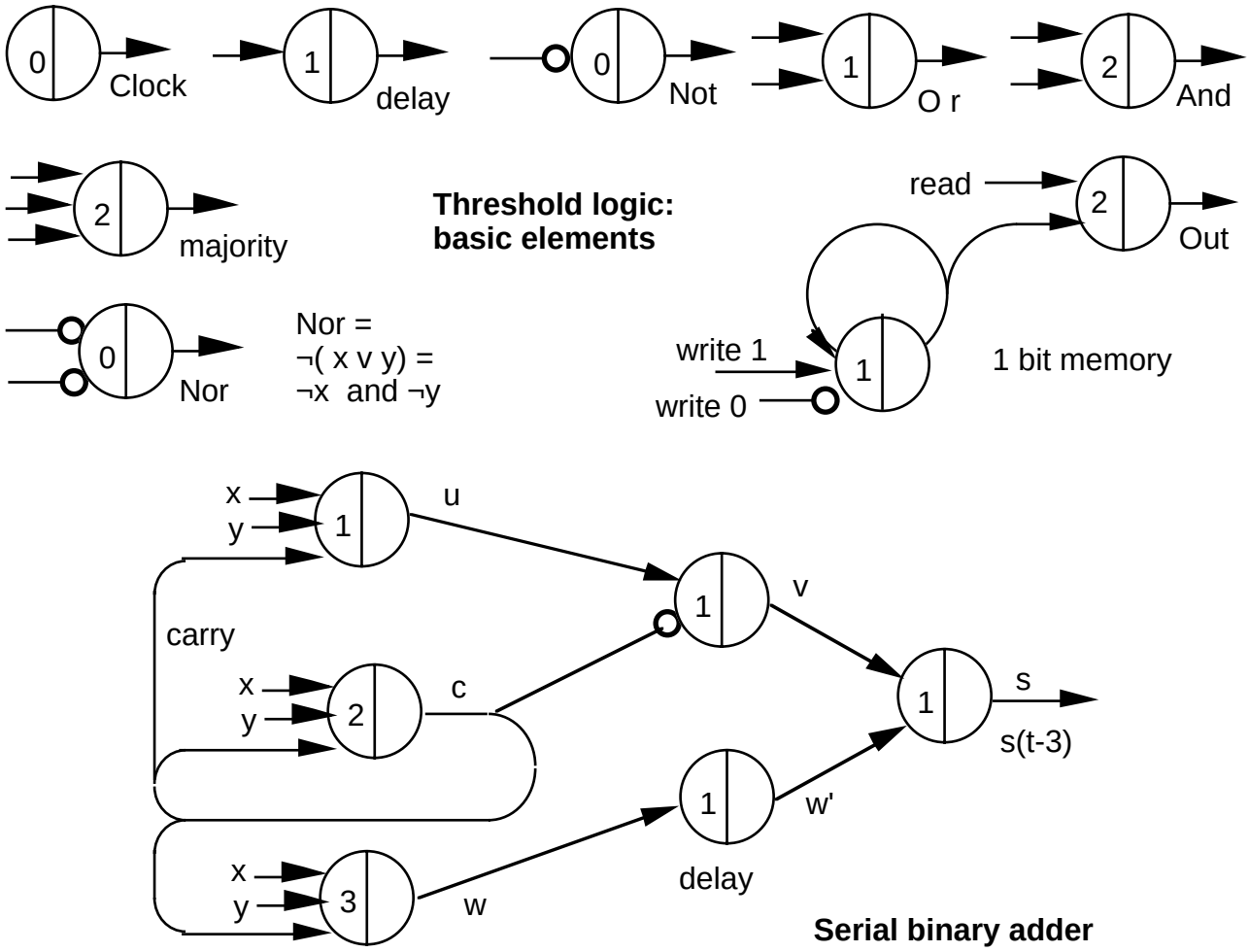
Solution 1.2: A **transposition** in a sequence $x_1, x_2, ..., x_n$ is a pair (i, j) with $i < j$ and $x_i > x_j$. The inverse vector has n-choose-2 transpositions. Every comparator reduces the number of transpositions by at most 1.

1.3 Threshold logic, perceptrons, artificial neural networks

W.S. McCulloch, W. Pitts: A logical calculus of the ideas immanent in nervous activity, Bull. Math. Biophysics, Vol 5, 115-137, 1943.

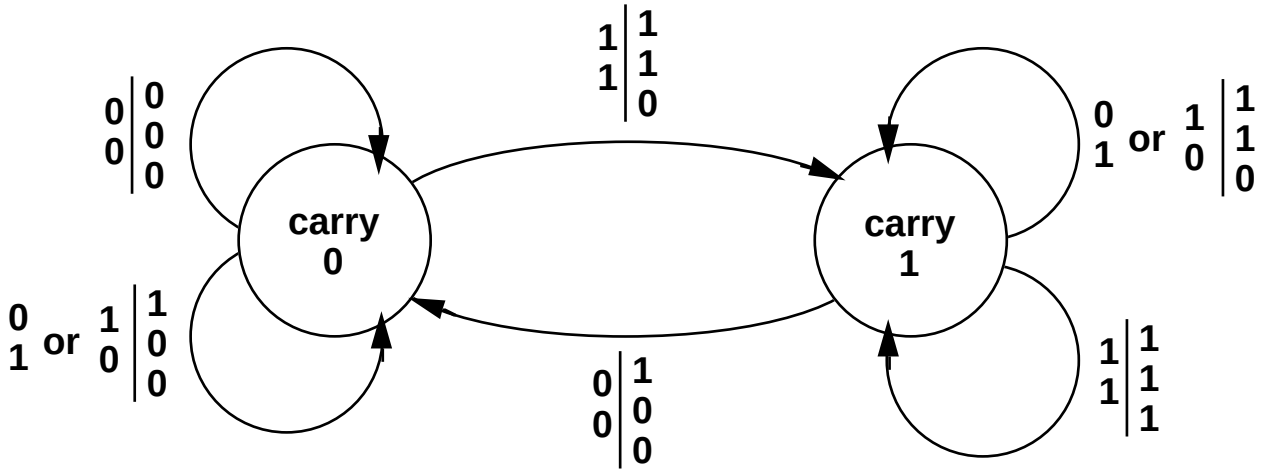
A threshold gate or "cell" is a logic element with multiple inputs that is characterized by its threshold value t. It counts, or adds up, the current input values and "fires", i.e. produces an output of 1, iff the sum equals or exceeds t. Threshold gates are more useful if they are generalized to include two kinds of inputs, excitatory and inhibitory inputs. A cell's output fires iff at least t excitatory inputs are on, and no inhibitor is on. In this model, which is used in the following examples, each inhibitor has veto power! (A variant called "subtractive inhibition" adds input values with positive or negative sign.)

We assume synchronous operation, with a gate delay of 1 clock tick. A threshold gate with inhibitory inputs is a universal logic element: the following figure shows how to implement and, or, not gates; delay and memory elements.



Msg: Logic design: how to assemble primitive elements into systems with a desired behavior. The theory of computation deals primarily with "black box behavior". But it insists that these black boxes **can** be built, at least in principle, from the simplest possible building blocks. Thus, the theory of computation involves an intricate interplay between abstract postulates and detailed "microprogramming".

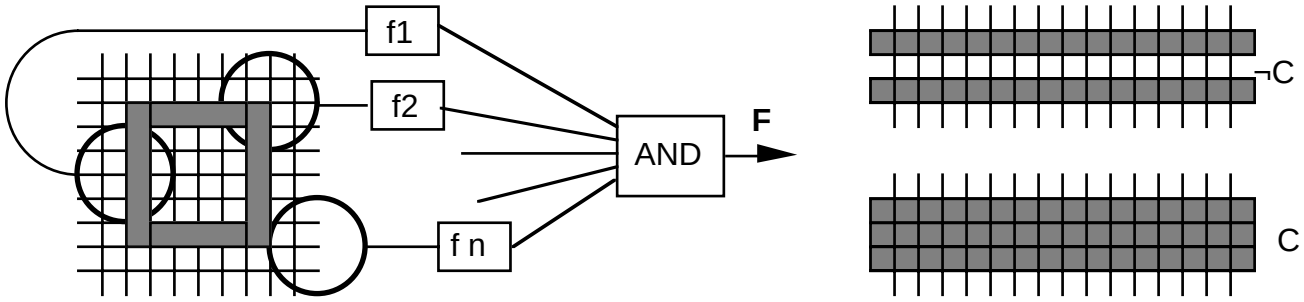
Example: the black-box behavior of the circuit above is specified by the finite state machine below:



Artificial neural nets applied to picture recognition

F. Rosenblatt: Principles of neurodynamics, Spartan Books, NY 1962. "Model of the eye's retina".
M. Minsky, S. Papert: Perceptrons, MIT Press, 1969.

We consider digital pictures, i.e. 2-d arrays of black and white pixels. We define devices, called perceptrons, that recognize certain classes of pictures that share some specific property, such as "convex" or "connected". Perceptrons are an example of parallel computation based on combining local predicates. What geometric properties can be recognized by such devices? A simple variant of "perceptrons", called 'conjunctively local', is illustrated in the figure below. Let f1, ..., fn be predicates with a bounded fanout, i.e. each fi looks at ≤ k pixels. The perceptron uses AND as the threshold operator, which means that the output F = AND (f1, ..., fn).



Convexity can be recognized. Introduce an f for each triple a, b, c of points on a straight line, with b between a and c. Define f = 0 iff a and c are black, b white; f = 1 otherwise.

Connectedness cannot be recognized by any perceptron of bounded fanout k. Pf by contradiction: Consider the 2 pictures above labeled C and ¬C that extend for > k pixels. If perceptron P can distinguish them, there must be an f* that yields f* = 0 on ¬C, and f* = 1 on C. Since f* looks at ≤ k pixels, there is a pixel p in the center row not looked at by P. By blackening this pixel p, we change ¬C into a connected picture C'. But f* "does not notice" this change, keeps voting f* = 0, and causes F to output the wrong value on C'.

Conclusion: convexity can be determined as a cumulation of local features, whereas connectedness cannot.

Msg: Combinational logic alone is weak. A universal model of computation needs unbounded memory, and something like feedback loops or recursion.

Hw 1.3: Define an interesting model suitable for computing pictures on an infinite 2-d array of pixels. Study its properties. Is your model "universal", in the sense that it is able to compute any "computable picture"?

Ex: Let N = {1, 2, ..} be the natural numbers. Consider the infinite pixel array N x N, pictures P over it, i.e. P : N x N -> {0, 1}, and the 4 operations: Set (row, column) k to (0, 1). Characterize the class C of pictures generated by finite programs, i.e. finite sequences of operations of the type above. Give a decision procedure that decides whether a given picture P is in C; and if it is, provides lower and upper bounds on the complexity of P, i.e. on the length of its shortest program that generates P.

1.4 Grammars and "languages": Chomsky's hierarchy

N. Chomsky: Three models for the description of language, IRE Trans. Information Th. 2, 113-124, 1956.

Motivating example:
Sentence -> Noun Verb Noun, e.g.: Bob loves Alice
Sentence -> Sentence Conjunction Sentence, e.g.: Bob loves Alice and Rome fights Carthage

Grammar G(V, A, P, S). V: alphabet of non-terminal symbols, "variables, "grammatical types";
A: alphabet of terminal symbols, $S \in V$: start symbol, "sentence";
P: unordered set of productions of the form L -> R, where $L, R \in (V \cup A)^*$

Rewriting step: for $x, y, y', z \in (V \cup A)^*$, u -> v iff u = xyz, v = xy'z and y -> y' $\in$ P
Derivation: "->*" is the transitive, reflexive closure of "->", i.e.
u ->* v iff $\exists$ w0, w1, ... wj, with $j \geq 0$, u = w0, w(i-1) -> wi, wj = v
Language defined by G: $L(G) = \{ w \in A^* \mid S \rightarrow^* w \}$

Various restrictions on the productions define different types of grammars and corresponding languages:
Type 0, **phrase structure grammar**: NO restrictions
Type 1, **context sensitive**: $|L| \leq |R|$, (exception: S -> ε is allowed if S never occurs on any right-hand side)
Type 2, **context free**: $L \in V$
Type 3, **regular**: $L \in V$, R = a or R = aX, where $a \in A$, $X \in V$

Exercise: Define a grammar for some tiny subset of natural language, and illustrate it with examples. Try this for a subset of English and the semantically similar subset of another language, and show that different natural languages have different grammars.

1.5 Markov algorithms: A universal model of computation

A.A. Markov (1903-1979) developed his "Theory of algorithms" around 1951. Markov algorithms can be interpreted as computer architecture for memory with sequential access only: Computation is modeled as a **deterministic** transformation of an input string into an output string, e.g. '1+2' => '3', according to productions that are scanned sequentially. The most comprehensive treatment in English is:
A.A. Markov, N.M. Nagorny: The Theory of Algorithms, (English transl), Kluwer Academic Publishers, 1988.

Alphabet A = {0, 1, ..}, our examples use A = {0, 1}. Functions A* -> A*. Marker alphabet M = { α, β, ..}.
Sequence (ordered) P = P1, P2, .. of **rewriting rules** (productions), which are of 2 types:
Pi = x -> y (continue) or Pi = x ¬ y (terminate), where x, y ∈ (A ∪ M)*.
Execution: use the **first rule** that applies to the data string, apply it at the **leftmost pattern match**.
A terminal rule stops the process.

Examples

1) Change all 0s to 1s. P1: 0 -> 1. No terminal rule is needed, algorithm stops when no rule applies.

2) Generate 0s forever: P1: ε -> 0. "ε" is the nullstring, it always matches to the left of any input string.

3) Append prefix '101': P1: ε ¬ 101. Terminal rule stops the rewriting process.

4) Append suffix '101'. Need marker, M = {α}. Careful when sequencing the rewrite rules!
P1: α 0 -> 0 α, P2: α1 -> 1 α, P3: α ¬ 101, P4: ε -> α
Rule P4 is executed first and generates the marker α. P4 appears last in the production sequence P, where it is protected by P3 from ever being executed again. The top priority rules P1 and P2 move α from the beginnig to the end of the data string. When α reaches its destination, the terminal rule P3 converts it to the suffix 101.

Notation

Usually we sequence the rules implicitly by writing them on separate lines. In example 4, the order in which the two rules P1: α 0 -> 0 α, P2: α1 -> 1 α appear, i.e. P1 P2 or P2 P1, is irrelevant. In order to emphasize that a subset of the rules may be permuted arbitrarily among themselves, we may write these on the same line:

4a) α 0 -> 0 α, α1 -> 1 α
α ¬ 101
ε -> α

Moreover, P1 and P2 have a similar structure. With a large alphabet A = { 0, 1, 2, ..} we need 2 |A| rules of the type α B -> B α, where the variable B ranges over A. In order to abbrievate the text, we write the algorithm as :

4b) α B -> B α
α ¬ 101
ε -> α

This is called a **production schema**, a meta notation which implies or generates the actual rule sequence 4a). A schema rule α B -> B α implies that the relative order of the individual rules generated is irrelevant.

Algorithm design

How do you invent Markov algorithms for specific tasks? Is there a "software engineering discipline" for Markov algorithms that leads you to compose a complex algorithm from standard building blocks by following general rules? Yes, there are recurrent ideas to be discovered by designing lots of Markov programs. Recurrent issues and problems in Markov programming:

a) Since the data string can only be accessed sequentially, i.e. you can't assign an "address" to some data of interest, you must identify the places in the string "where things happen" by placing markers. We call this initial placement of markers "the initialization or opening phase".

b) After the initial placement of markers, each operation on the data string is usually preceded by a scan from the beginning of the string to a specific marker. Let's call the substring consisting only of symbols from the alphabet A, with markers removed, the "restricted data string". We call the repeated scans for markers, followed by operations on the restricted data string, the "operating phase".

c) At verious times during execution, at the latest when the operations on the data string (restricted to the alphabet A) are done, now useless or harmful markers clutter the data string. “Cleaning up”, i.e. removing unneeded markers, is non-trivial: if you place $\alpha \rightarrow \varepsilon$ too early in the production sequence, marker α may be wiped out as soon as it is created!

d) Many simple Markov algorithms can be designed as executing in three phases: initialization, operating phase, clean-up phase. More complex algorithms, of course, may involve repeated creation and removal of markers, with various operating phases in between.

e) Once you have a plan for a Markov algorithm, it is often easy to see what type of productions are needed. Determining and verifying the correct order of these productions, however, is usually difficult. The presence or absence of specific markers in the data string helps to prove that certain productions cannot apply at a particular moment; thus, they may be ignored when reasoning about what happens during certain time intervals.

f) The crucial step is to invent invariants that are maintained at each iteration. This is by far the most important technique for understanding how an algorithm works, and proving it to be correct!

g) If you see a Markov algorithm without explanation of its logic, it is very difficult to ”reverse engineer” the designer’s ideas. Therefore we will only look at algorithms that are presented with arguments in support of how and why they work.

The following **examples** illustrate frequently useful techniques.

5) Generate a palindrome, $f(s) = s\, s^{reverse}$, $A = \{0, 1\}$

We postulate the following invariant for the operating phase. Let $s = s_L\, s_R$ be a partition of the data string s into a prefix s_L and a suffix s_R . Ex: s = a b c d e , s_L = a b, s_R = c d e. At all times, the current string s has the form $s = s_L\, \gamma\, s_L^{reverse}\, \alpha\, s_R$. In the example above: s = a b γ b a α c d e. The marker γ marks the center of the palindrome built so far from the prefix s_L; α marks the boundary between the palindrome already built and the suffix s_R yet to be processed. If we now remove the first letter of s_R and copy it twice, to the left and right of γ , the invariant is reestablished, with s_R shrunk and s_L expanded: a b c γ c b a α d e .

$0\, \beta_0 \rightarrow \beta_0\, 0,\ 1\, \beta_0 \rightarrow \beta_0\, 1, 0\, \beta_1 \rightarrow \beta_1\, 0,\ 1\, \beta_1 \rightarrow \beta_1\, 1$	the carry β_0 or β_1 moves left
$\gamma\ \beta_0 \rightarrow 0\, \gamma\, 0,\ \gamma\ \beta_1 \rightarrow 1\, \gamma\, 1$	carry has reached the center, produces a bit and its mirror image
$\alpha\, 0 \rightarrow \beta_0\, \alpha\ , \alpha\, 1 \rightarrow \beta_1\, \alpha$	remove the leftmost bit of s_R and save its value in markers β_0 or β_1
$\alpha \rightarrow \varepsilon\ , \gamma \neg \varepsilon$	at the end, remove the markers; notice terminating production!
$\varepsilon \rightarrow \gamma\ \alpha$	at the beginning, create the markers at the far left

Reversing a string is the key part of constructing a palindrome. We can easily modify Algorithm 5 to merely reverse a string by changing the invariant $s = s_L\, \gamma\, s_L^{reverse}\, \alpha\, s_R$ to $s = \gamma\, s_L^{reverse}\, \alpha\, s_R$, and the rules $\gamma\ \beta_0 \rightarrow 0\, \gamma\, 0,\ \gamma\ \beta_1 \rightarrow 1\, \gamma\, 1$ to $\gamma\ \beta_0 \rightarrow \gamma\, 0,\ \gamma\ \beta_1 \rightarrow \gamma\, 1$.
The following example illustrates a technique that uses a single marker for different purposes at different times, and thus, is be a bit hard to fathom.

6) Reverse the input string s: $f(s) = s^{reverse}$

P1	$\alpha\ \alpha\ \alpha \rightarrow \alpha\ \alpha$	a double α that merges with a single α gets trimmed back
P2	$\alpha\ \alpha$ B -> B $\alpha\ \alpha$	double α’s function as a pseudo marker to wipe out single α’s !
P3	$\alpha\ \alpha \neg\ \varepsilon$	
P4	α B’ B” -> B” α B’	this schema stands for 4 rules of the type $\alpha\ 1\ 0 \rightarrow 0\ \alpha\ 1$
P5	$\varepsilon \rightarrow \alpha$	gets executed repeatedly, but P2 and P3 stop P5 from creating more than 2 consecutive α’s

This algorithm is best understood as executing in two phases.
Phase 1, when productions P4 and P5 are active, mixes repeated initial placements of the marker α with a operations that reverse the restricted data string. Invariant of Phase 1: there are never 2 consecutive α’s in the data string, hence productions P1, P2, P3 do not apply.

After phase 1, there is a brief interlude where production P5 creates 2 α's at the head of the string.
Phase 2: the cleanup phase removes all markers by activating productions P1 and P2. Invariant of Phase 2: the data string always contains a pair of consecutive α's. Thanks to the "pseudo marker" , one of P1, P2, P3 always applies, and P4 and P5 no longer get executed. At the very end, the terminating production P3 wipes out the last markers.

7) Double the input string s: f(s) = ss

Introduce markers α, γ to delimit parts of the current string with distinct roles. Let $s = s_L\ s_R$ be a partition of s into a prefix s_L and a suffix s_R . In general, the current string has the form $s_L\ \alpha\ s_R\ \gamma\ s_R$. By removing the rightmost bit of s_L and copying it twice at the head of each string s_R , we maintain this invariant with s_L shrunk and s_R expanded. Some stages of the current string: s, $s\ \alpha\ \gamma$, $s_L\ \alpha\ s_R\ \gamma\ s_R$, $\alpha\ s\ \gamma\ s$, ss.

$\beta_0\ \gamma$ -> $\gamma\ 0$, $\beta_1\ \gamma$ -> $\gamma\ 1$	the carry β_0 or β_1 converts back to 0 or 1
$\beta_0\ 0$ -> $0\ \beta_0$, $\beta_0\ 1$ ->$1\ \beta_0$, $\beta_1\ 0$ -> $0\ \beta_1$, $\beta_1\ 1$ -> $1\ \beta_1$	the carry β_0 or β_1 moves right
$\alpha\ \gamma\ 0$ -> $0\ \alpha\ \gamma$, $\alpha\ \gamma\ 1$ -> $1\ \alpha\ \gamma$	the markers travel to the far right
$0\ \alpha$ -> $\alpha\ 0\ \beta_0$, $1\ \alpha$ -> $\alpha\ 1\ \beta_1$	move one bit of s_L to expand the first s_R and generate a carry β
α -> ε , γ ¬ ε	at the end, remove the markers; notice terminating production!
ε -> $\alpha\ \gamma$	at the beginning, create the markers at the far left

8) Serial binary adder

Given 2 n-bit binary integers $x = x_n .. x_1\ x_0$, $y = y_n .. y_1\ y_0$, $n \geq 0$, compute their sum $z = z_{n+1}\ z_n .. z_1\ z_0$, $A = \{0, 1, +, =\}$, $M = \{ \beta_0, \beta_1 \}$. β_0, β_1 store and transport a single bit.
Coding: Input string: x+y=0 , Output string: =z.
Idea. The input string x+y=0 gets transformed into intermediate strings of the form
$x_L + y_L = z_R$, where x_L is $x_n .. x_{i+1}\ x_i$, y_L is $y_n .. y_{i+1}\ y_i$, z_R is $z_i z_{i-1} .. z_1\ z_0$.
As the bit pair x_i, y_i is being cut off from the tail of x_L and y_L, a sum bit z_i is appended to the front of z_R .
Invariant I: $z_R = x_R + y_R$, where x_R , y_R are the tails cut off from x, y, respectively. I holds initially.
The algorithm is built from the following components:

Full adder: The addition logic is represented by 8 productions that can be written in any order:

$0\beta_0$=0 -> =00	$0\beta_1$=0 -> =01	$1\beta_0$=0 -> =01	$1\beta_1$=0 -> =10
$0\beta_0$=1 -> =01	$0\beta_1$=1 -> =10	$1\beta_0$=1 -> =10	$1\beta_1$=1 -> =11

Save the least significant bit of x_L: 0+ -> $+\beta_0$ 1+ -> $+\beta_1$

Transport this bit β next to the least significant bit of y_L:

$\beta_0 0$ -> $0\beta_0$	$\beta_0 1$ -> $1\beta_0$	$\beta_1 0$ -> $0\beta_1$	$\beta_1 1$ -> $1\beta_1$

These building blocks are sequenced such that, at any time during execution, the string contains at most one β:

Full adder
Transport β
Save the least significant bit of x_L
Remove "+": += -> =

Markov algorithms are one of several universal models of computation: you can program an algorithm to compute anything "computable". E.g.. there is a Markov algorithm that simulates a universal Turing machine. We have seen simple and data movement operations usefle for string matching and copying. In order to make this assertion of universality plausible, we now program simple control structures.

9) Control logic: f(s) ≡ if s = x then y else z; s, x, y, z ∈ A*.

Because x, y, and z are parameters that may assume arbitrary string values, we must expand our notation. We introduce an informal meta-language consisting of new symbols. For example, the "production" x ¬ y is

really a **production schema** that stands for an arbitrary specific instance, e.g for 00 ¬ 111. Production schemata also let us write an algorithm more concisely (particularly when the alphabet A is large). Let B be a symbol that stands for 0 or 1.The schema Bα -> α stands for the pair 0α -> α, 1α -> α.
The following algorithm distinguishes 3 cases:
1) s = x, 2) x is a proper substring of s, 3) s ≠ x and x isn't a substring of s.

Bα -> α	"eraser" α wipes out 0s and 1s to the left
αB -> α	α wipes out 0s and 1s to the right
α ¬ z	if s ≠ x then z. How do we know s ≠ x? That was checked when generating α
xB -> α	if s contains x as a proper substring ..
Bx -> α	.. generate α
x ¬ y	if s = x then y
ε -> α	if s ≠ x then generate α

Hw: Consider the context-free grammar G: E -> x | E¬ | E E ∧ | E E ∨ that generates the language Ls of Boolean suffix expressions over a single literal x and three Boolean operators Not, And, Or. Write a Markov algorithm that distinguishes syntactically correct expressions from incorrect ones.

Hw: Learn to use the software package Exorciser demonstrated in class, and solve the exercises on Markov algorithms contained therein.

1.6 Random access machines (RAM), the ultimate RISC

The model of computation called **random access machine,** RAM, is used most often in algorithm analysis. It is significantly more "powerful", in the sense of efficiency, than either Markov algorithms or Turing machine because its memory is not a tape, but an array with random access. Provided the programmer knows where an item currently needed is stored, a RAM can access it in a single memory reference, avoiding the sequential access tape searching that a Turing machine or a Markov algorithm need.

A RAM is essentially a *random access memory*, also abbreviated as RAM, of unbounded capacity, as suggested in the Fig. below. The memory consists of an infinite array of cells, addressed 0, 1, 2, … . To make things simple we assume that each cell can hold a number, say an integer, of arbitrary size, as the arrow pointing to the right suggests. A further assumption is that an arithmetic operation (+ , – , · , /) takes unit time, regardless of the size of the numbers involved. This assumption is unrealistic in a computation where numbers may grow very large, but is often useful. As is the case with all models, the responsibility for using them properly lies with the user.

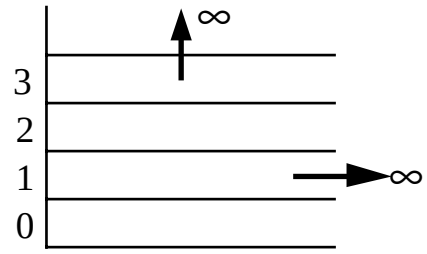


Fig.: RAM - unbounded address space, unbounded cell size

The ultimate RISC.

RISC stands for *Reduced Instruction Set Computer*, a machine that has only a few types of instructions built into hardware. What is the minimum number of instructions a computer needs in order to be universal? One! Consider a stored-program computer of the "von Neumann type" where data and program are stored in the same memory (John von Neumann, 1903 – 1957). Let the random access memory (RAM) be "doubly infinite": There is a *countable infinity* of memory cells addressed 0, 1, … , each of which can hold an integer of arbitrary size, or an instruction. We assume that the constant 1 is hardwired into memory cell 1; from 1 any other integer can be constructed. There is a single type of "three-address instruction" which we call "subtract, test and jump", abbreviated as

STJ x, y, z

where x, y, and z are addresses. Its semantics is equivalent to

STJ x, y, z ⇔ x := x – y; if x ≤ 0 then goto z;

x, y, and z refer to cells Cx, Cy, and Cz. The contents of Cx and Cy are treated as data (an integer); the contents of Cz, as an instruction.

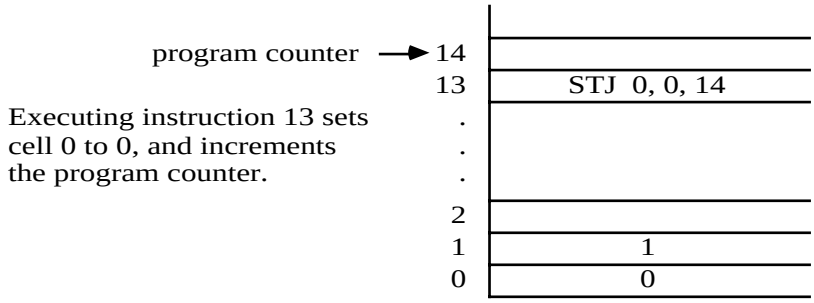


Fig.: Stored program computer: Data and instructions share the memory.

Since this RISC has just one type of instruction, we waste no space on an op-code field. But an instruction contains three addresses, each of which is an unbounded integer. In theory, three unbounded integers can be packed into the same space required for a single unbounded integer. This simple idea leads to a well-known technique introduced into mathematical logic by Kurt Gödel (1906 – 1978).

Exercise: Gödel numbering
a) Motel Infinity has a countable infinity of rooms numbered 0, 1, 2, … . Every room is occupied, so the sign claims "No Vacancy". Convince the manager that there is room for one more person.
b) Assume that a memory cell in our RISC stores an integer as a sign bit followed by a sequence d0, d1, d2, … of decimal digits, least significant first. Devise a scheme for storing three addresses in one cell.
c) Show how a sequence of positive integers i1, i2, , in of arbitrary length n can be encoded in a single natural number j: Given j, the sequence can be uniquely reconstructed. Gödel's solution: $2^{i1}\ 3^{i2}\ 5^{i3}$ ...(n-th prime)^{in} .

Basic RISC program fragments

To convince ourselves that this computer is universal we begin by writing program fragments to implement simple operations, such as arithmetic and assignment operator. Programming these fragments naturally leads us to introduce basic concepts of assembly language, in particular symbolic and relative addressing.

Set the content of cell 0 to 0: STJ 0, 0, .+1

Whatever the current content of cell 0, subtract it from itself to obtain the integer 0. This instruction resides at some address in memory, which we abbreviate as '.', read as "the current value of the program counter". '.+1' is the next address, so regardless of the outcome of the test, control flows to the next instruction in memory.
a := b, where a and b are symbolic addresses. Use a temporary variable t:

```
STJ  t, t, .+1        { t := 0 }
STJ  t, b, .+1        { t := –b }
STJ  a, a, .+1        { a := 0 }
STJ  a, t, .+1        { a := –t, so now a = b }
```

Exercise: A program library
(a) Write RISC programs for
a := b + c, a := b · c, a := b div c, a := b mod c, a := |b|, a : = min(b, c), a := gcd(b, c).
(b) Show how this RISC can compute with rational numbers represented by a pair [a, b] of integers denoting numerator and denominator.
(c) (Advanced) Show that this RISC is universal, in the sense that it can simulate any computation done by any other computer.

The exercise of building up a RISC program library for elementary functions provides the same experience as the equivalent exercise for Turing machines, but leads to the goal much faster, since the primitive STJ does a lot more work in a single operation than the primitives of a Turing machine.

1.7 Programming languages, [un-]decidability

Having learned to program an extremely primitive computer, it is obvious that any of the conventional programming languages is a universal model of computation. The power of universal computation comes at a

conceptual cost: many reasonable questions about the behavior of a universal computing model are **undecidable**: that is, they cannot be answered by following any effective decision process. The **halting problem** is the standard example. We present it here in the words of Christopher Strachey, as written in a letter to the editor of The Computer Journal ( ):

To the Editor, The Computer Journal

An impossible program

Sir,
A well-known piece of folk-lore 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 a railway carriage on the way to a Conference at the NPL 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 thew wrong value, and this contradiction shows that the function T cannot exist.

Yours faithfully, C. Strachey
Churchill College, Cambridge

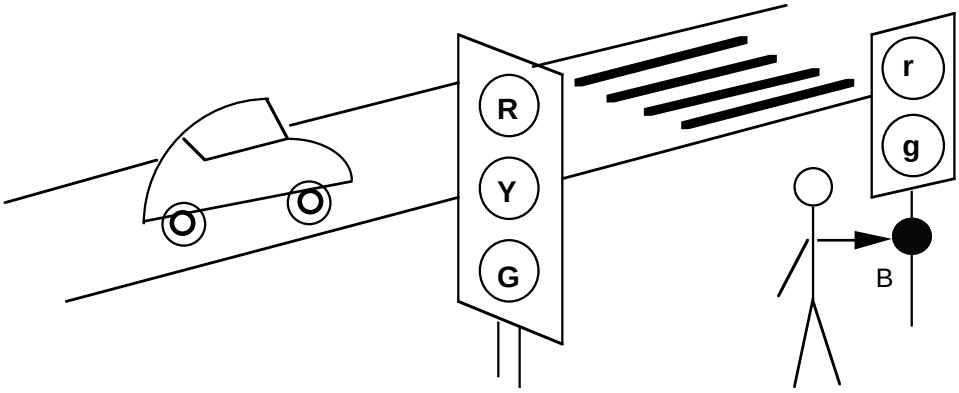
End of Ch1

2. Finite state machines (fsm, sequential machines): examples and applications

Goal of this chapter: fsm's are everywhere in our technical world! Learn how to work with them.

2.1 Example: Design a finite state controler to synchronize traffic lights

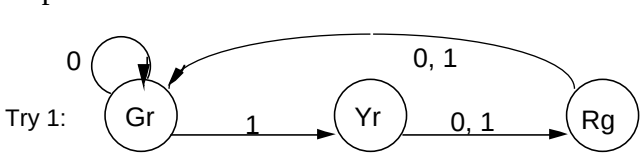
Finite state machines are the most common controlers of machines we use in daily life. In the example illustrated by the figure, the intersection of a main road with a pedestrian crossing is controled by two traffic lights. The traffic light for cars has 3 possible values: red R, yellow Y, and green G. The traffic light for pedestrians has 2 possible values: red r and green g. Your task is to design a finite state controler that turns the various lights on and off in a reasonable and safe manner.

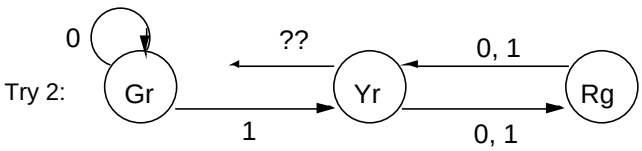


The controler reacts to an input signal B that has 2 possible values, 1 (pressed) and 0 (not pressed), depending on whether or not a pedestrian has pressed the button B in the preceding time interval. This signal is sampled at regular time intervals of, say, 10 seconds, and the controler executes a transition once every 10 seconds. The time interval of 10 seconds is assumed to be sufficient to cross the road.

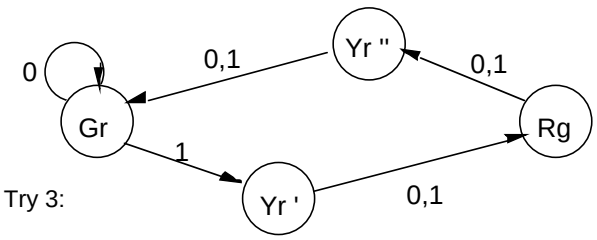
Exercise:
a) List all the states that this traffic light system is potentially capable of signaling to an observer, and comment on the desirability and role of each such visible state.
b) Draw the state diagram of a finite state machine to control both traffic lights. (Notice that the state space of the controler is not necessarily identical with the visible states of the lights).
c) Comment on the behavior, advantages and disadvantages of your design: how safe is it? does it give precedence to pedestrians or to cars? There is no unique best solution to this problem - the clarity of your design and analysis counts!

One possible solution:



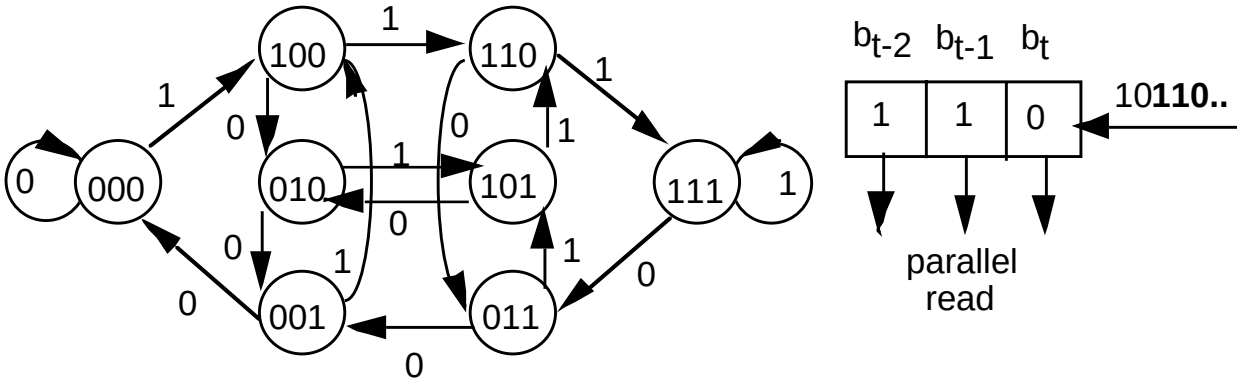


The solution at left has the problem that it relies on the courtesy of drivers to delay take-off until all pedestrians complete their crossing. If we try to use the delay state Yr in both phases, the transition from Gr to Rg and also from Rg to Gr, we run into a technical problem: both transitions out of Yr have been set to lead to Rg. The solution below recognizes the fact that the total state of the system is not necessarily limited to to the visible state. When the lights show Yr, we need an additional bit of memory to remember whether the previous state was Gr or Rg. Thus, we introduce 2 delay states, which look identical to an observer, but are distinguished internally.

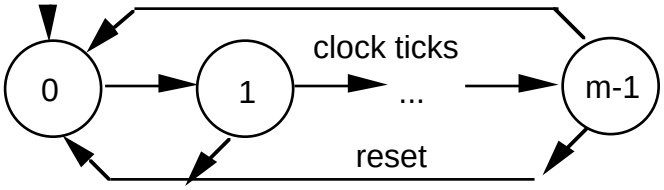


2.2 Simple finite state machines encountered in light switches, wrist watches, ticket vending machines, computer user interfaces, etc

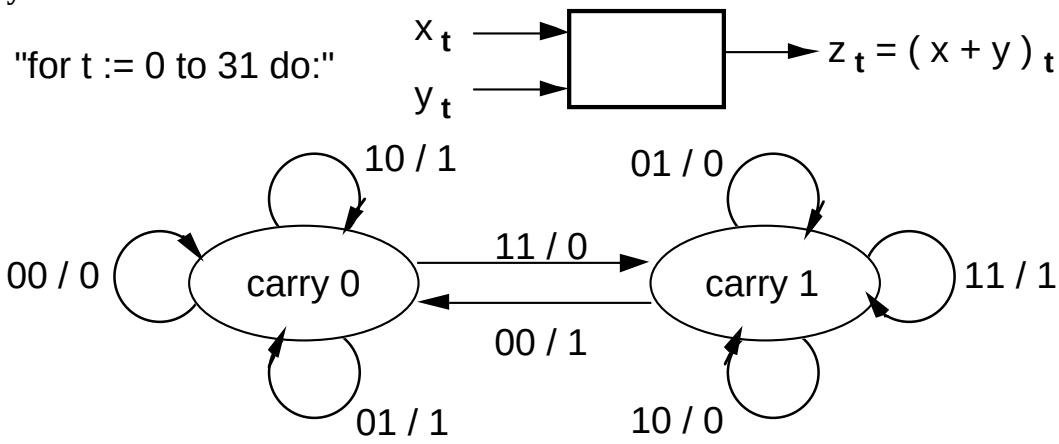
Shift register: save the last 3 bits of a data stream



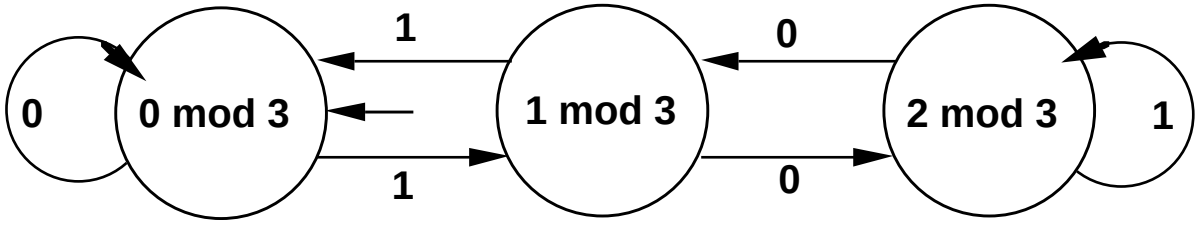
Counter mod m with reset (e.g. clock):



Serial binary adder:



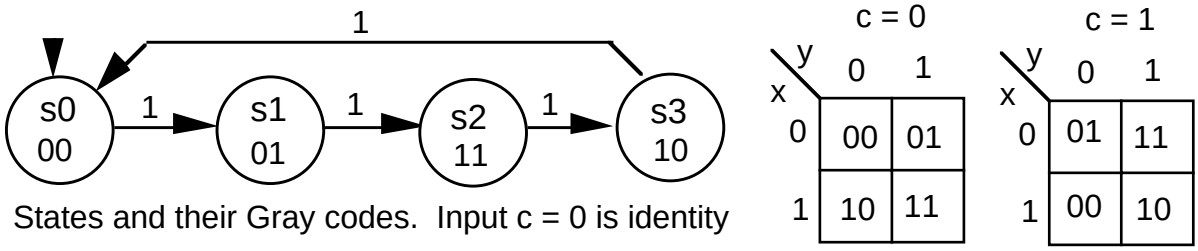
Mod 3 divider: read a binary integer "left to right", i.e. most significant bit first



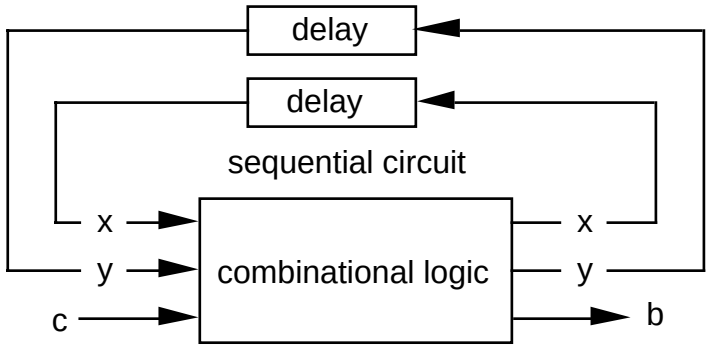
Reason for this construction. Consider an integer with the binary representation L b, where L is a bitstring and b is a single bit. Let |L| denote the integer represented by L, and similarly for |L b| . Assume L has been read, most significant bit first, and, by of example, the fsm has found that |L| mod 3 = 1, recorded by the fact that it is now in state '1 mod 3'. Then L 0 represents the integer |L 0| = 2 |L|, where |L 0| mod 3 = 2. Similarly, L 1 represents |L 1| = 2 |L| + 1, where |L 1| mod 3 = 0. This justifies the two transitions out of state '1 mod 3'.

Sequential circuit design

Logic design for a mod 4 counter. State assignment using Gray codes (each transition changes 1 single bit).

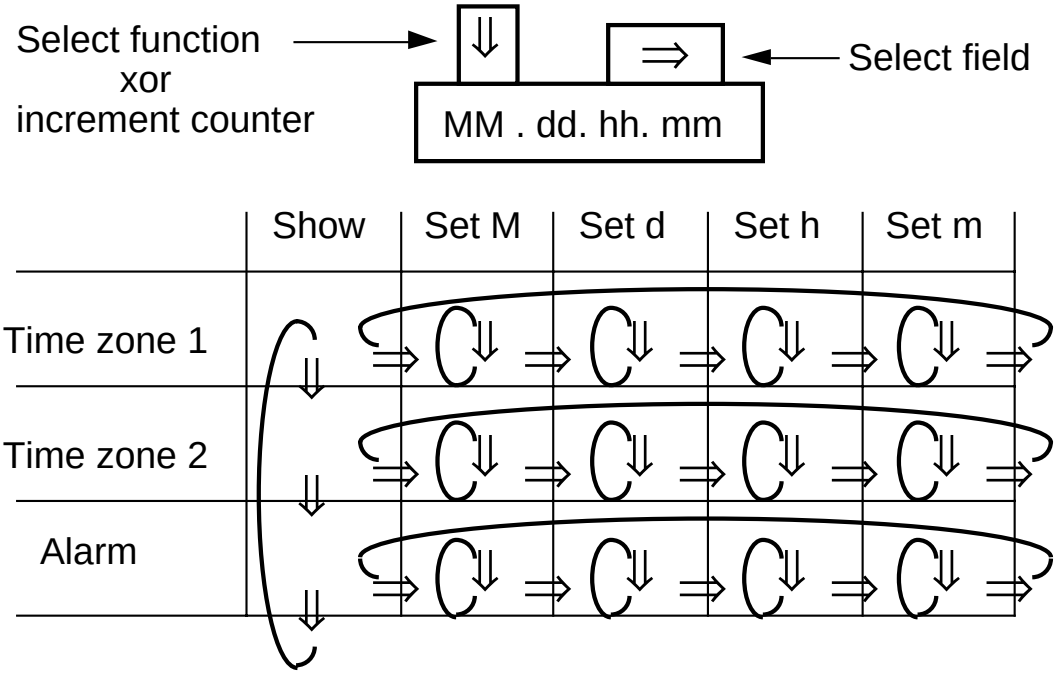


State assignement: 2 boolean variables x, y code the states: s0: 00, s1: 01, s2: 11, s3: 10.
Input c: count the number of 1's, ignore the 0s. Ouput b: b = 1 in s3, b = 0 otherwise.
Combinational logic: $x := \neg c \wedge x \vee c \wedge y$, $y := \neg c \wedge y \vee c \wedge \neg x$, $b := x \wedge \neg y$



2.3 Design a digital watch interface: State space as a Cartesian product

Goal: 2 buttons suffice, no operating manual is needed.

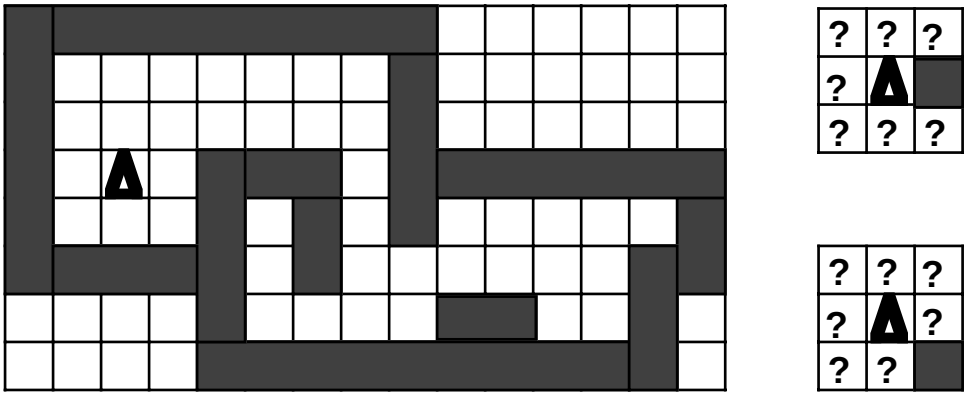


This is the state space (that should be) visible to the user, e.g. by displaying the function (e.g. "TZ2") and by flashing the active field. The machine's internal state space of course is much more complex. Each field is a counter: mod 12, mod 31, mod 24, mod 60.

Design principle: the state space visible to the user is a homomorphic image of the internal state space

2.4 The wall-hugging robot

Program a toy robot to follow a wall. The figure below illustrates the assumptions. In a simplified version of "turtle geometry" [Papert], the robot's world is a grid, any square is either free or occupied by a "wall". The robot, shown as a triangle, is placed on a free square facing in one of four possible directions. The robot has two binary sensors: **h** ("head", a touch sensitive bumper) signals whether the square in front of the robot is free (0) or occupied by a wall (1); **r** (right) provides the same information for the square to the right of the robot. The robot is capable of two primitive actions: **R** turns right by 90° while remaining on the same square; **F** (forward) advances one square in the robot's current direction.



a) The city wall and b) the predicate "wall to the rear-right"

The robot must be programmed to find a piece of wall, then endlessly cycle along the inside of the wall, hugging it with his right side. The following finite state machine, presented in tabular form, solves the problem.

Seek: precondition = true, postcondition = "wall to the rear-right"

r	**h**	**Actions**	**Next state**
1	-		Track
-	1	RRR	Track
0	0	F	Seek

Track: precondition = postcondition = "wall to the rear-right"

r	**h**	**Actions**	**Next state**
0	-	RF	Track
1	0	F	Track
1	1	RRR	Track

The robot starts in state Seek, not knowing anything about its position, as expressed by the vacuous assertion "precondition = true". The function of this state is to bring about the postcondition "wall to the rear-right", illustrated in Fig.b: There is a piece of wall to the right of the robot, or diagonally to the rear-right, or both. As long as the robot has not sensed any piece of wall (**r** = 0, **h** = 0) it moves forward with F. If it senses a wall, either ahead or to the right, it positions itself so as to fulfill the postcondition "wall to the rear-right". With the "don't care" notation "-" we follow the design principle "specify behavior only as far as needed" and make the robot non-deterministic.

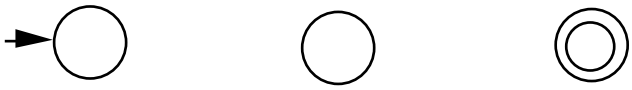
The mathematically inclined reader may find it instructive to prove the program correct by checking that the state Track maintains the invariant "wall to the rear-right", and that, in each transition, the robot progresses in his march along the wall. Here we merely point out how several fundamental ideas of the theory of computation, of algorithms and programs can be illustrated in a simple, playful setting whose rules are quickly understood.

2.5 Varieties of finite state machines and automata: definitions

Notation: Alphabet A = {a, b, ..} or A = {0, 1, ..}. A* = { w, w', ...}. Nullstring ε. Empty set {} or Ø. "Language" L ⊆ A*. Set S = {s, s', .., s0, s1, ..}. Cardinality |S|. Power set $\mathbf{2^S}$. "¬" not or complement.

Deterministic Finite Automaton (FA, DFA), **Finite State Machine** (FSM): M = (S, A, f, s0, ...)
Set of states S, alphabet A, transition function f: S x A -> S, initial state s0.
Other components of M, indicated above by dots "...", vary according to the purpose of M.

Acceptor (the standard model in theory)**:** M = (S, A, f, s0, F), where F ⊆ S is a set of accepting or final states.



Notation: starting state (arrow), non-accepting state, accepting state

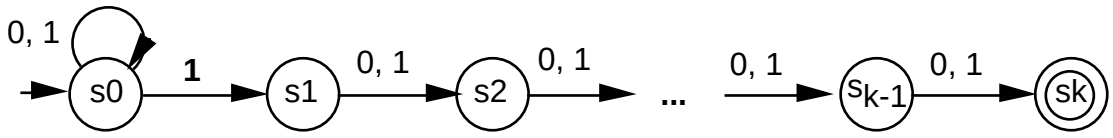
Extend f from S x A -> S to f: S x A* -> S as follows: f(s, ε) = s, f(s, wa) = f(f(s, w), a) for w ∈ A*
Df: **M accepts w** ∈ A* iff f(s0, w) ∈ F. Set L ⊆ A* accepted by M: **L(M)** = { w | f(s0, w) ∈ F}.

Transducer (fsm's used in applications)**:** M = (S, A, f, g, s0), with a function g that produces an output string over an alphabet **B**: g: S -> B (Moore machine), h: S x A -> B (Mealy machine)
An acceptor is the special case of a transducer where F(s) = 1 for s ∈ F, F(s) = 0 for s ∉ F.

Non-deterministic Finite Automaton (NFA) **with ε-transitions: f: S x (A ∪ {ε}) -> $\mathbf{2^S}$**.
Special case: NFA without ε-transitions: **f: S x A -> $\mathbf{2^S}$.**
Variation: **Probabilistic FA**: An NFA whose transitions are assigned probabilities.

Extend f: S x A* -> 2^S: f(s, ε) = ε-hull of s = all states reachable from s via ε-transitions (including s);
f(s, wa) = ∪ f(s', a) for s' ∈ f(s, w).
Extend f further f: 2^S x A* -> 2^S as follows: f(s1, .., sk, a) = ∪ f(si, a) for i = 1, .., k.
Df: **M accepts w** ∈ A* iff f(s0, w) ∩ F ≠ {}. Notice: w is accepted iff ∃ some w-path from s0 to F.
Set L ⊆ A* accepted by M: L(M) = { w | f(s0, w) ∩ F ≠ {}}.

A non-deterministic machine spawns multiple copies of itself, each one tracing its own root-to-leaf path of the tree of all possible choices. Non-determinism yields an **exponential increase in computing power**!
Ex 1: L = (0 ∪ 1)* 1 (0 ∪ 1)$^{k-1}$ i.e. all strings whose k-th last bit is 1. A DFA that accepts L must contain a shift register k bits long, with 2^k states as shown in 2.2 for k=3. A NFA accepts L using only k+1 states, by "guessing" where the tail-end k bits start. This example shows that simulating a NFA by a DFA may require an **exponential increase in the size of the state space.**



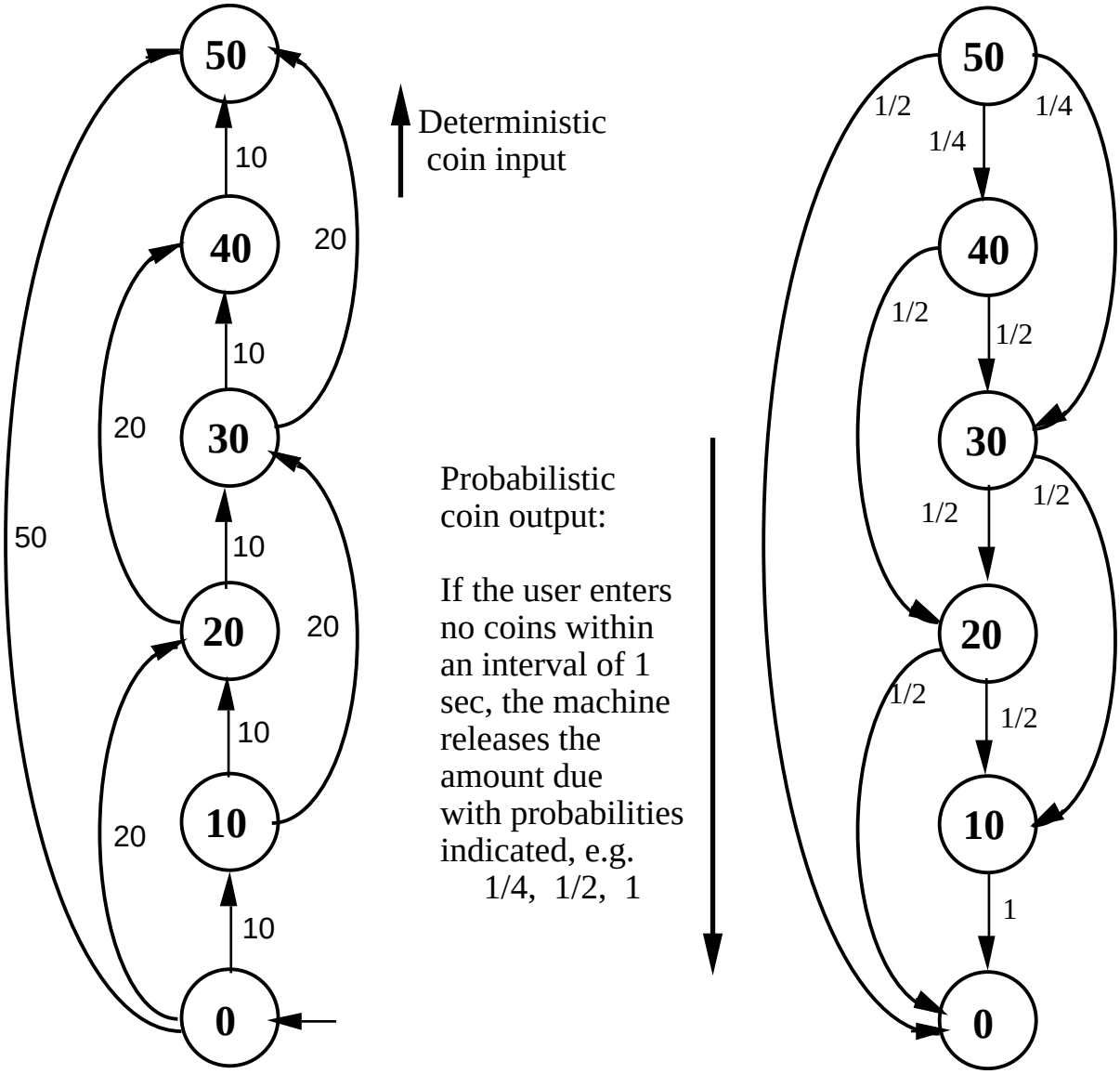
NFA accepts the language of strings whose k-th last bit is 1 - by "guessing"

2.6 Probabilistic coin changer

Any utility machine (vending or ticket machine, watch, video cassette recorder. etc.) is controled by an fsm. Understanding their behavior (user interface) frequently requires a manual, which we usually don't have at hand. A diagram of the fsm, perhaps animated, would often help the novice user to trace the machine's behavior. As an example, imagine a coin changer modeled after gambling machines, whose display looks as shown in the following figure.

The states correspond to the amount of money the machine owes you, and the current state lights up. As long as you enter coins rapidly, the machine accumulates them up to the total of 50 cents. If you pause for a clock interval, the machine starts emitting coins randomly, to the correct total. After a few tries you are likely to get useful change: either breaking a big coin into smaller ones, or vice- versa.

:



Exercise: Mathematically analyze, or estimate by simulation, the average number of tries (times you need to enter a coin) in order to obtain a desired change. E.g., to change a 50 cent coin into five 10 cent coins.

Exercise: Logic design for the fsm "binary integer mod 3" shown in 2.2.

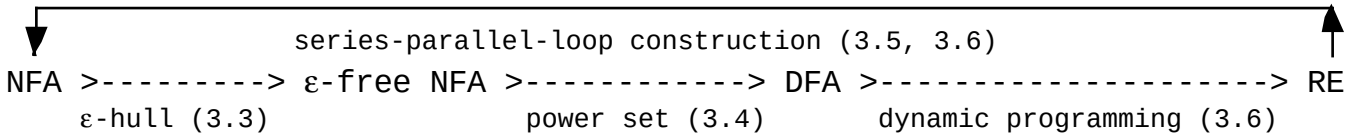
Hw2.1: Design an fsm that solves the problem "binary integer mod 3" when the integer is read least significant bit first (as opposed to most significant bit first as shown in 2.2.

Hw2.2: Analyze the behavior of the ticket machines used by Zurich's system of public transportation, VBZ. Draw a state diagram that explains the machines' behavior. Evaluate the design from the user's point of view.

End of Ch2

3. Finite automata and regular languages: theory

Goal of this chapter: equivalence of DFA, NFA, regular expressions



3.1 Varieties of automata: concepts, definitions, terminology

Notation: Alphabet A , e.g. A = {0, 1, ..}. Kleene star A* = { w | w = a1 a2 ... am, m ≥ 0, ai ∈ A}.
A^+ = AA*. Nullstring ε. Empty set {} or Ø.
"Language" L ⊆ A*. Set S = {s, s', .., s0, s1, ..}. Cardinality |S|. Power set $\mathbf{2^S}$. "¬" not or complement.

Deterministic Finite Automaton (FA, DFA) M = (S, A, f, s0, F)
Set of states S, alphabet A, transition function f: S x A -> S, initial state s0, accepting or final states F ⊆ S
Extend f from S x A -> S to f: S x A* -> S as follows: f(s, ε) = s, f(s, wa) = f(f(s, w), a) for w ∈ A*
Df: **M accepts w** ∈ A* iff f(s0, w) ∈ F. Set L ⊆ A* accepted by M: **L(M)** = { w | f(s0, w) ∈ F}.

Non-deterministic Finite Automaton (NFA) **with ε-transitions: f: S x (A ∪ {ε}) -> 2^S**.
Special case: NFA without ε-transitions: **f: S x A -> 2^S.**

Extend f: S x A* -> 2^S: f(s, ε) = ε-hull of s = all states reachable from s via ε-transitions (including s);
f(s, wa) = ∪ f(s', a) for s' ∈ f(s, w).
Extend f further f: 2^S x A* -> 2^S as follows: f(s1, .., sk, a) = ∪ f(si, a) for i = 1, .., k.
Df: **M accepts w** ∈ A* iff f(s0, w) ∩ F ≠ {}. Notice: w is accepted iff ∃ some w-path from s0 fo F!
Set L ⊆ A* accepted by M: L(M) = { w | f(s0, w) ∩ F ≠ {}}.

The operation of a non-deterministic machine can be interpreted in 2 equivalent ways:
- Oracle: For any given word w, if there is any sequence of transitions that leads to an accepting state, the machine will magically find it, like a sleep walker, avoiding all the paths that lead to a rejecting state.
- Concurrent computation: The machine spawns multiple copies of itself, each one tracing its own root-to-leaf path of the tree of all possible choices. If any copy reaches an accepting state, it broadcasts success.
Non-determinism yields an **exponential increase in computing performance**!

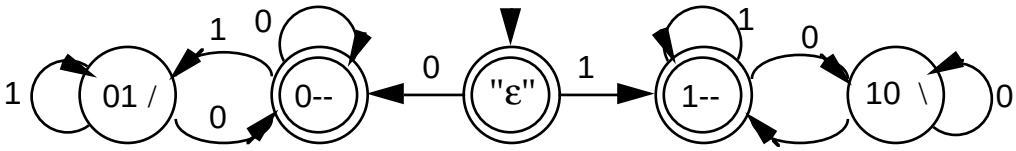
Df: Two FAs (of any type) are **equivalent** iff they accept the same language.

HW 3.1:
Design a) an NFA and b) a DFA that accepts the language L = (0 ∪ 1)*01 of all strings that terminate in 01.
Given an arbitrary string w over {0, 1}, describe a general rule for constructing
a) an NFA N(w) and b) a DFA M(w) that accepts the language L(w) = (0 ∪ 1)*w.

3.2 Examples

Ex 1: **FAs "can't count"**: No FA can recognize L = {0^k1^k | k > 0}. By way of contradiction, assume ∃ FA M that accepts L, |M| = n. In the course of accepting w = 0^n1^n, as M reads the prefix 0^n , it goes thru n+1 states s0, s1, .., sn. By the "pigeon hole principle", some 2 states si, sj in this sequence must be equal, si = sj, i < j. Thus, M cannot distinguish the prefixes 0^i and 0^j, and hence also accepts, incorrectly, w' = $0^{n-(j-i)}1^n$, and many other ill-formed strings. Contradiction, QED.

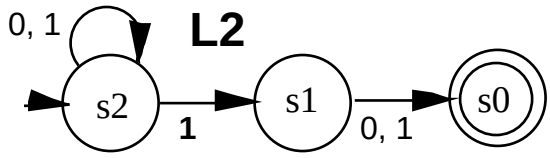
Ex 2: **Careful about "can't count"!** Let L = {w ∈ {0 ∪ 1}* | #(01) = #(10) }. In any w, "ups" = 01 and "downs" = 10 alternate, so #(01) = #(10) ± 1.Solution: L = {w | first character = last character }.

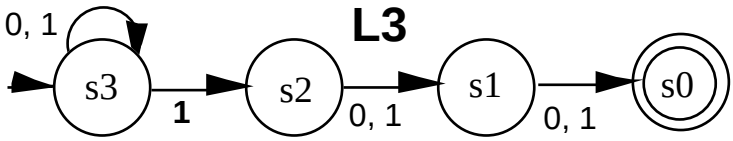


Exercise: We saw a 2-state fsm serial adder. Show that there is no fsm multiplier for numbers of arbitrary size.

Ex 3: NFA's clairvoyance yields an exponential reduction of the size of the state space as compared to DFAs.

Consider $L_k = (0 \cup 1)^* \ 1 \ (0 \cup 1)^{k-1}$ i.e. all strings whose k-th last bit is 1. A NFA accepts L_k using only k+1 states (as shown for k = 2 and k = 3) by "guessing" where the tail-end k bits start.

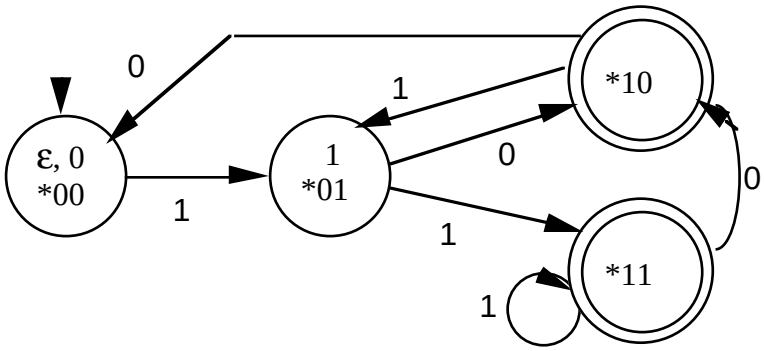




A DFA that accepts L_k must contain a shift register k bits long, hence has at least 2^k states. This shows that, in general, simulating a NFA by a DFA requires an **exponential increase in the size of the state space.**

The following DFA $M(L_2)$ has a state for each of the 2-bit suffixes 00, 01, 10, 11. Each state s corresponds to a "language", i.e. a set of words that lead M from its initial state to s. The short strings ε , 0, 1 can be associated with some set of "long" strings with the following semantics:

ε, 0, *00: no "1" has been seen that might be useful as the next-to-last bit
1, *01: the current bit is a "1", if this turns out to be the next-to-last bit, we must accept
*10: accept if this is the end of the input; if not, go to state *00 or *01 depending on the next bit read
*11: accept if this is the end of the input; if not, go to state *10 or *11 depending on the next bit read

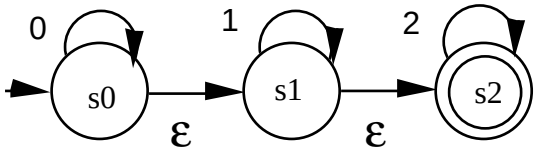


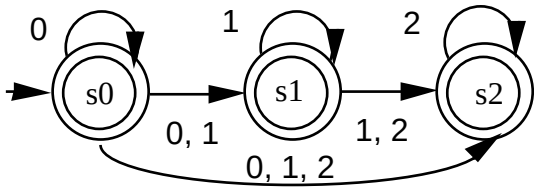
This DFA $M(L_2)$ is a subset of of a DFA $M(L_2)$ with 8 states that is obtained from the general construction of section 3.4 to simulate a NFA N by some DFA M.

3.3 Spontaneous transitions

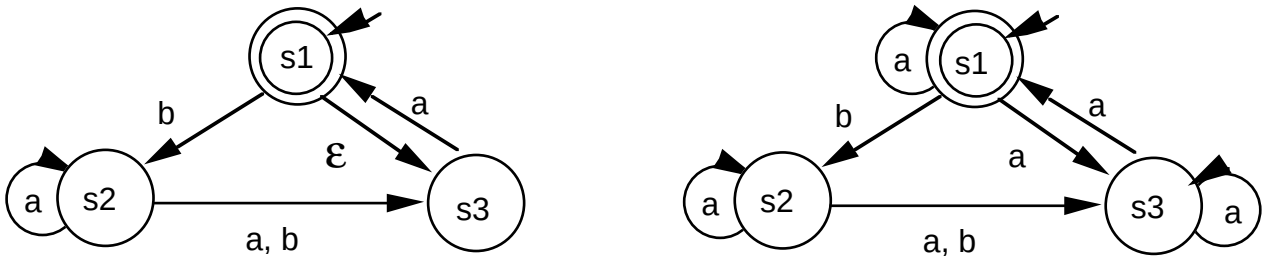
Lemma (ε-transitions):
Any NFA N with ε-transitions can be converted to an equivalent NFA N' without ε-transitions.
Pf: Let an ε-path be any cycle-free sequence of transitions labeled ε (this includes paths of length 0). If an ε-path leads from state s to state r, this means that anything that can be done from r can already be done from s. Before removing ε-transitions we must introduce a) new accepting states, and b) new transitions.
a) For any ε-path from state s to state r, where r is accepting but s is not, make s accepting
b) For any path of the form E1 a E2 that leads from s to r, where E1 and E2 are ε-paths and $a \in A$, add the transition f(s,a) = r.
There are only finitely many cycle-free paths, thus this terminates. Thereafter, drop all ε-transitions. QED

Ex 4: L = 0*1*2*. This language is typical of the structure of communication protocols. A message consists of a prefix, a body, and a suffix, in this order. If any part may be of arbitrary length, including zero length, the language of legal messages has the structure of L.





Ex 5: NFA converted to an ε-free NFA, and later to a DFA. L = (a ∪ ba* (a ∪ b) a)*

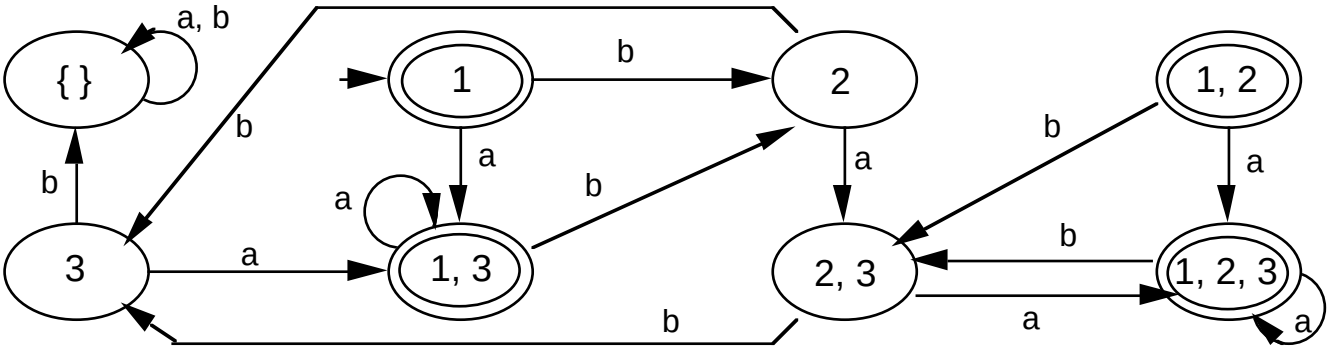


3.4 DFA simulating NFA: the power set construction.

Thm (equivalence NFA-DFA): Any NFA N can be converted into an equivalent DFA M'.

Pf: Thanks to Lemma on ε-transitions, assume without loss of generality that N = (S, A, f, s0, F) has no ε-transitions. Define M' = (2^S, A, f', {s0}, F') as follows. 2^S is the power set of S, {s0}the initial state. F' consists of all those subsets R ⊆ S that contain some final state of N i.e. R ∩ F ≠ {}. f': 2^S x A-> 2^S is defined as:
for R ∈ 2^S and a ∈ A, f'(R, a) = { s ∈ S | s ∈ f(r, a) for some r ∈ R}.
N and M' are equivalent due the following invariant: for all x ∈ A*, f'({s0}, x) = f(s0, x). QED

Ex (modified from Sipser p57-58). Convert the NFA of 3.3 Ex 5 at right to an equivalent DFA.



The power set construction tends to introduce unreachable states. These can be eliminated using a transitive closure algorithm. As an alternative, we generate only states of M' corresponding to subsets of S as they are being reached, effectively combining transitive closure with the construction of the state space.

3.5 Closure of the class of regular sets under union, catenation, and Kleene star

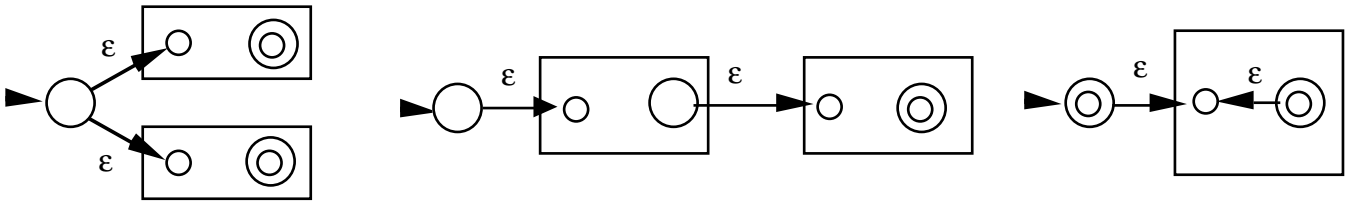
Df: **Set (or language) L ⊆ A* is** called **regular iff L is accepted by some FA.**
It turns out that all FAs (DFA or NFA, with or without ε-transitions) are equivalent w.r.t. "accepting power".

Given L, L' ⊆ A*, define union L ∪ L', catenation L ∘ L' = { v = ww' | w ∈ L, w' ∈ L'}.

Define L^0 = { ε }, L^k = L ∘ L^{k-1} for k > 0. Kleene star: L* = ∪ (k = 0 to ∞) L^k .

Thm (closure under the regular operations**)**:

If L, L' ⊆ A* are regular sets, L ∪ L', L ∘ L' and L* are regular sets.
Pf: Given FAs that accept L, L' respectively, construct NFAs to accept L ∪ L', L' ∘ L' and L* as shown. The given FAs are represented as boxes with starting state at left (small) and one accepting state (representative of all others) at right (large). In each case we add a new starting state and some ε-transitions as shown.



In addition to closure under the regular operations union, catenation, and Kleene star, we have:

Thm: if L is regular, the complement ¬L is also regular.
Pf: Take a **D**FA M = (S, A, f, s0, F) that accepts L. M' = (S, A, f, s0, S-F) accepts ¬L. QED

Thm: If L, L' ⊆ A* are regular, the intersection L ∩ L' is also regular. Pf: L ∩ L' = ¬(¬L ∪ ¬L'). QED

Thus, the class of regular languages over an alphabet A forms a Boolean algebra.

3.6 Regular expressions

Df: Given an alphabet A, the class R(A) of regular expressions over A is obtained as follows:
Primitive expressions: a for ever a ∈ A, ε (nullstring), Ø (empty set).
Compound expressions: if R, R' are regular expressions, (R ∪ R'), (R ◦ R'), (R*) are regular expressions.
Convention on operator priority: * > ◦ > ∪. Use parentheses as needed to define structure.
A regular expression denotes a regular set by associating the expression operators *, ◦ , ∪ with the set operations Kleene star, catenation, and union, respectively.

Thm: A set L is regular iff L is described by some regular expression.

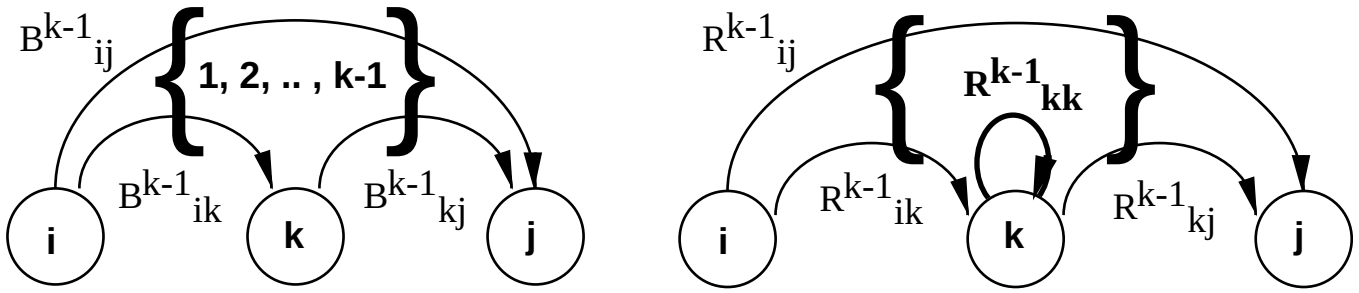
Pf <=: Convert a given regular expression R into an NFA N. Use trivial NFAs for the primitive expressions, the constructions used in the Closure Thm for the compound expressions. QED

Pf =>(McNaughton, Yamada 1960. Compare: Warshall's transitive closure algorithm, 1962):
Let DFA M = (S, A, f, s1, F) accept L. S = { s1, .. , sn}. Define R^k_{ij} = the set of all strings w that lead M from state si to state sj **without passing thru any state with label > k**.
Initialize: R^0_{ij} = { a | f(si, a) = sj } for i ≠ j. $R^0_{i\,i}$ = { a | f(si, a) = si ∪ { ε }.
Induction step: $R^k_{ij} = R^{k-1}_{ij} \cup R^{k-1}_{ik} \, (R^{k-1}_{kk})^* \, R^{k-1}_{kj}$
Termination: R^n_{ij} = the set of all strings w that lead M from state si to state sj without any restriction.
L(M) = ∪ R^n_{1j} for all sj ∈ F. The right hand side is a regular expression that denotes L(M). QED

Intuitive verification. Remember Warshall's transitive closure and Floyd's "all distances" algorithms.

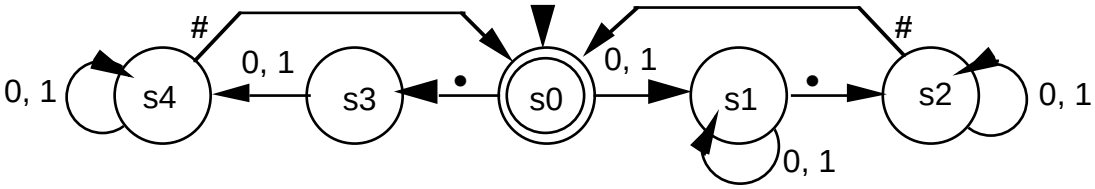
Warshall	Floyd
$B^0_{i\,j} = A_{i\,j}$ adjacency matrix, $B^0_{i\,i}$ = true	$B^0_{i\,j} = A_{i\,j}$ edge length matrix, $B^0_{i\,i} = 0$
$B^k_{i\,j} = B^{k-1}_{i\,j}$ or (B^{k-1}_{ik} and B^{k-1}_{kj})	$B^k_{i\,j} = \min (B^{k-1}_{i\,j} , B^{k-1}_{ik} + B^{k-1}_{kj})$
$B^n_{i\,j} = C_{i\,j}$ connectivity matrix	$B^n_{i\,j} = D_{i\,j}$ distance matrix

In Warshall's and Floyd's algorithms, cycles are irrelevant for the issue of connectedness and harmful for computing distances. Regular expressions, on the other hand, describe **all** paths in a graph (state space), in particular the infinitely many cyclic paths. Thus, we add a loop R^{k-1}_{kk} in the Fig. at right, and insert the regular expression $(R^{k-1}_{kk})^*$ between R^{k-1}_{ik} and R^{k-1}_{kj}.



3.7 DFAs and right invariant equivalence relations. State minimization.

Ex "real constants": A = {0, 1, •, #}. L = **(** **(** $(0 \cup 1)^+ \bullet (0 \cup 1)^*$ $\cup$ $(0 \cup 1)^* \bullet (0 \cup 1)^+$ **)** # **)***
Interpret a word in L as a sequence of real constants with a mandatory binary point •, e.g 0•1, 11•, •1.
A constant must have at least one bit 0 or 1, the binary point alone is excluded. Constants are separated by #.
To get a DFA, imagine that the transitions not shown in the figure all lead to a non-accepting trap state s5.



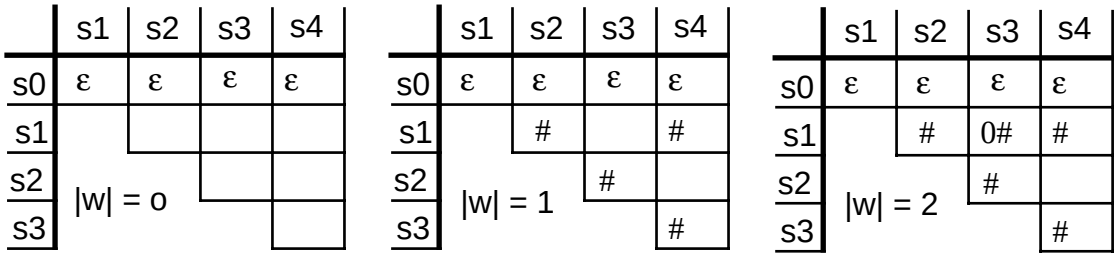
State identification, equivalent states: Given the state diagram of a DFA M, devise an experiment to
1) determine the current state of M, or 2) to distinguish two given states r, s.

Ex: In order to identify s0, feed ε into M - no other state is accepting. '•#' uniquely identifies s1.
'#' distinguishes between s3 and s4. No experiment distinguishes s2 from s4: s2 and s4 are equivalent.
Equivalent states can be merged to obtain a smaller FA M' equivalent to M.

Df: States r and s of M are equivalent (indistinguishable) iff for all w ∈ A*, f(r, w) ∈ F ⇔ f(s, w) ∈ F.
It turns out that in order to prove 2 states equivalent, it suffices to test all words w of length |w| ≤ n = |S|.

Before proving this result, consider a dynamic programming algorithm to identify non-equivalent state pairs. We start with the observation that all states might be equivalent. As pairs of non-equivalent states are gradually being identified, we record for each such pair s, r a shortest witness that distinguishes s and r. We illustrate this algorithm using the example of the FA "real constants" above.

At left in the figure below, all state pairs si ≠ sj are marked that can be distinguished by some word of length 0. This distinguishes accepting states from non-accepting states, and ε is a shortest witness. Unmarked slots identify pairs that have not yet been proven distinguishable. For each of these unmarked pairs r, s, and all a ∈ A, check whether the pair f(r, a), f(s, a) has been marked distinguishable. If so, mark r, s distinguishable with a shortest witness w = aw', where w' is inherited from f(r, a), f(s, a). When computing the entry for s1, s3 at right, for example, notice that f(s1, B) = s1, f(s3, B) = s4. Since s1, s4 have already been proven distinguishable by w' = #, s1, s3 are distinguishable by w = B#.
Checking the last unmarked pair s2, s4 at right yields no new distinguishable pair: f(s2,#) = f(s4,#) = s0; f(s2, •) = f(s4, •) = trap state s5; f(s2,B) = s2, f(s4,B) = s4, but s2, s4 have not yet been proven distinguishable. This terminates the process with the information that s2, s4 are equivalent and can be merged. Distinguishable states obviously cannot be merged -> this is a state minimization algorithm.



Hw 3.2.: Invent another interesting example of a DFA M with equivalent states and apply this dynamic programming algorithm to obtain an equivalent M' with the minimum number of states.
Hw 3.3: Analyze the complexity of this dynamic programming algorithm in terms of |S| = n and |A|.
Hw 3.4: Prove the following Thm: If states r, s are indistinguishable by words w of length $|w| \le n = |S|$, r and s are equivalent. Hint: use the concepts and notations below, and prove the lemmas.

Df "r, s are indistinguishable by words of length ≤ k":
$r \sim_k s$ for $k \ge 0$ iff for all $w \in A^*$, $|w| \le k$: $f(r, w) \in F \Leftrightarrow f(s, w) \in F$
Important properties of of the equivalence relations $\sim_k$:
Lemma (inductive contruction): $r \sim_k s$ iff $r \sim_{k-1} s$ and for all a: $f(r, a) \sim_{k-1} f(s, a)$
Lemma (termination): If $\sim_k = \sim_{k-1}$, $\sim_k = \sim_m$ for all $m > k$.
Thm: If r, s are indistinguishable by words w of length $|w| \le n = |S|$, r and s are equivalent.

An algebraic approach to state minimization

Given any $L \subseteq A^*$, define the equivalence relation (reflexive, symmetric, transitive) R_L over A^*:
$x R_L y$ iff All $z \in A^*$, $xz \in L \Leftrightarrow yz \in L$. I.e., either xz and yz both in L, or xz and yz both in ¬L
Notice: R_L is "right invariant": $x R_L y \Rightarrow$ All $z \in A^*$, $xz R_L yz$.
Intuition: $x R_L y$ iff the prefixes x and y cause all pairs xz, yz to share [non-]membership status w.r.t. L.

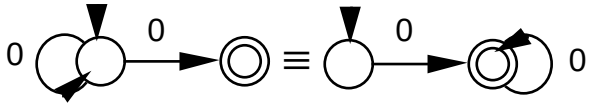
Given DFA M, define equivalence relation R_M over A^*: $x R_M y$ iff $f(s0, x) = f(s0, y)$. R_M is right invariant.

Df: index of equivalence relation R = # of equivalence classes of R.

Thm (regular sets and equivalence relations of finite index). The following 3 statements are equivalent:
1) $L \subseteq A^*$ is accepted by some DFA M
2) L is the union of some of the equivalence classes of a right invariant equivalence relation of finite index
3) R(L) is of finite index.

Thm: The minimum state DFA accepting L is unique up to isomorphism (renaming states).

In contrast, minimum state **N**FAs are not necessarily unique. Ex $A = \{0\}$, $L = 0^+$:



3.8 Odds and ends about regular languages and FAs

The "pumping lemma" (iteration lemma):
For any regular $L \subseteq A^*$ there is an integer $n > 0$ (the "pumping length") with the following property:
any $w \in L$ of length $|w| \ge n$ can be sliced into 3 parts w = xyz satisfying the following conditions:
1) $|xy| \le n$, 2) $|y| > 0$, **3) for all $i \ge 0$, $x y^i z \in L$.**

Pf: Consider any DFA M that accepts L, e.g. the minimum state DFA M(L). Choose $n = |S|$ as the pumping length. Feed any $w \in L$ of length $|w| \ge n$ into M. On its way from s0 to some accepting state, M goes through $|w| + 1 \ge n + 1$ states. Among the first n+1 of these states, s0, s1, .., sn, .., there must be a duplicate state si = sj for some i < j, with a loop labeled y leading from si back to si. Thus, xz, xyz, xyyz, ... are all in L. QED

The pumping lemma is used to prove, by contradiction, that some language is **not** regular.
Ex: $L = \{ 0^i 1^j \mid i < j\}$ is **not** regular. Assume L is regular. Let n be L's pumping length. Consider $w = 0^n 1^{n+1}$, $w \in L$. Even if we don't know how to slice w, we know $|xy| \le n$ and hence $y = 0^k$ for some $k > 0$. But then $0^n 1^{n+1}$, $0^{n+k} 1^{n+1}$, $0^{n+2k} 1^{n+1}$, .. are all $\in L$, contradicting the definition $L = \{ 0^i 1^j \mid i < j\}$. L is not regular, QED.

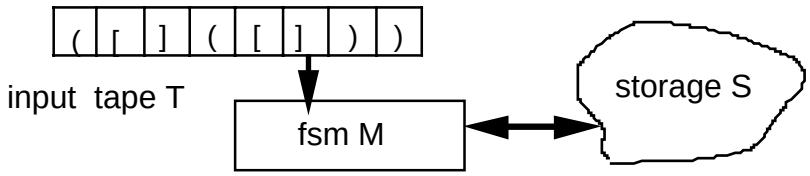
End of Ch3

4. Finite automata with external storage. Pushdown automata (PDA) jn 03.05.05

Concepts of this chapter: Finite automata with external storage of bounded or unbounded capacity, subject to various access restrictions. Counter automata (CA), pushdown automata (PDA), linear bounded automata (LBA). Non-equivalence of deterministic and non-deterministic PDAs.

4.1 Finite automata with external storage - components, details, notation

M: FSM controls access to an input tape (if any) and a storage device. General form of a transition:
(current state of M, currently scanned symbol on T, currently scanned symbol on S)
-> (new state of M, newly written symbol on S, motion of the read/write head on S)

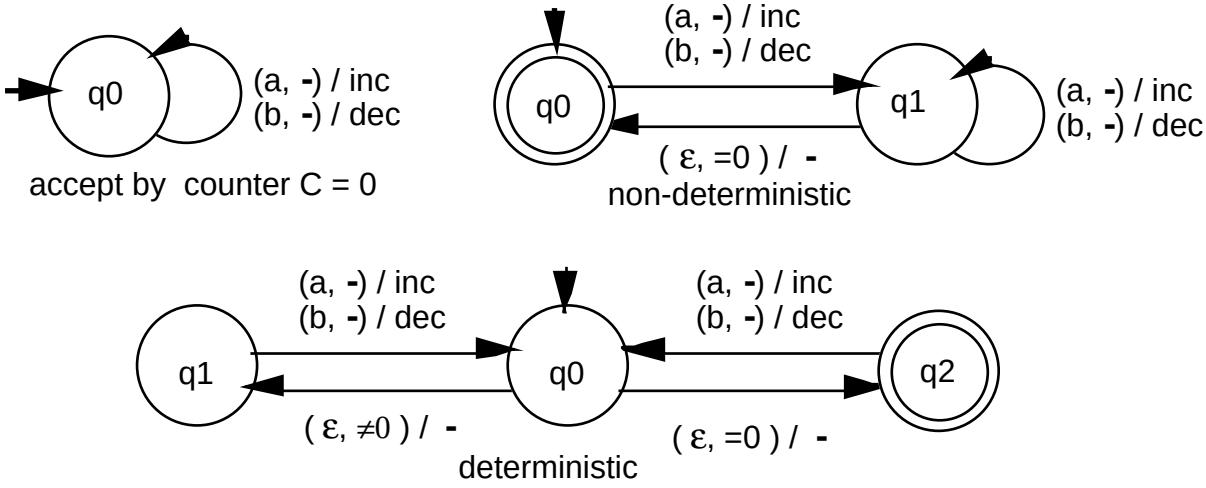


The most important parameter: size of storage device (usually unbounded) and types of access operations. Details may vary greatly, e.g:
There may be no input tape T. The input to be processed is written in a designated area of the storage device.
Structure of the storage device, e.g: one or more tapes (single-ended or double-ended), 2-d grid, etc.
Acceptance: by accepting state or by designated content of the storage device (e.g. empty).
Tape T and storage S may have separate alphabets, which we call A, B.
We will use different conventions chosen to make each example simple.

4.2 Counter automata

A counter automaton consists of a finite automaton M augmented by a register (counter) C that can hold a single integer of arbitrary size. C is initialized to zero. M can increment and decrement the counter, and test it for 0. Thus, M's memory is unbounded, and arguments of the type "FAs can't count", as formalized in the pumping lemma of Ch 3, don't apply. Restrictions on M's computational power are caused by access limited to increment, decrement, and test for 0. This makes it impossible for M to code 2 integers a, b into a single integer, e.g. $2^a 3^b$. Due to these access restrictions, counter automata "can't do much more than count".

As an example, each of the the following counter automata accepts the language of all strings over A = {a, b} with an equal number of a's and b's:



A transition is activated by a pair (input symbol, state of counter) and triggers an action on the counter. Testable states of the counter are '=0' and '≠0', and counter actions are increment 'inc' and decrement 'dec'. Instead of reading an input symbol, M can also act based on the counter state alone, which we denote as 'reading the nullstring ε". M may ignore the state of the counter, which we denote by '-' = "don't care". Finally, M may choose to execute no action on the counter C, which we denote by '-' = "don't act".

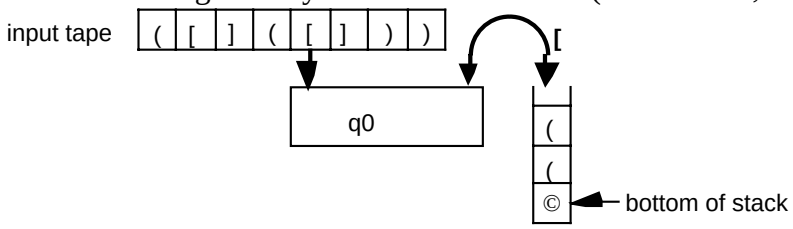
HW 4.1: Parenthesis expressions. Polish or parenthesis-free notation for arithmetic expressions.

a) Consider the language L1 of correct parenthesis expressions over the alphabet A = { (,)}. Examples of correct expressions: (( )), ( ) ( ), (( )) ( ), and the nullstring ε. Either: exhibit a counter automaton M1 that accepts the language L1 , or show that no such counter automaton exists.
b) Consider the language L2 of correct parenthesis expressions over the alphabet A = { (,), [,]}, involving two pairs of parentheses. Examples of correct expressions: ([ ]), ( ) [ ] ( ), (( ) []) ( ), and the nullstring ε. Example of an incorrect expression: ([ )]. Either: exhibit a counter automaton M2 that accepts the language L2 , or argue that no such counter automaton exists.
c) Polish or parenthesis-free notation for arithmetic expressions come in 2 versions: prefix and suffix notation. Consider operands designated by a single letter, say x, y, or z, and the 4 binary operators +, -, *, /. In prefix notation the operator is written before its 2 operands, in suffix notation after. x+y becomes +xy or xy+. Design a counter automaton P that recognizes correct prefix expressions, and a counter automaton S that recognizes correct suffix epressions.

4.3 Deterministic pushdown automata (DPDA)

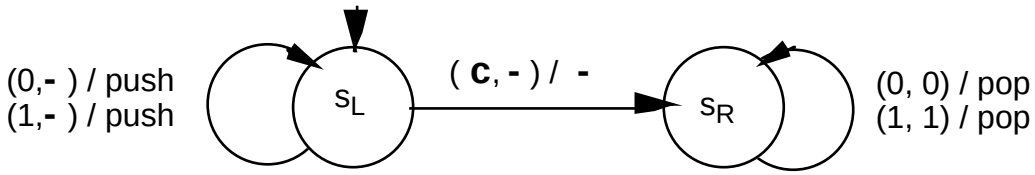
PDA: FSM controls a stack. Unbounded memory, limited last-in-first-out (LIFO) access.
Abbreviation: $A_\varepsilon = A \cup \{ \varepsilon \}$, $B_\varepsilon = B \cup \{ \varepsilon \}$
Df DPDA: M= (Q, A, B, f, q0, F), f: Q x A_ε x B_ε -> Q x B_ε
Transition: (q, a, b) -> (q', v), $a \in A_\varepsilon$, $b \in B_\varepsilon$, $v \in B^*$.
When M is shown in a diagram, this transition is drawn as an arrow from q to q' labeled a, b -> v.
There may be a special 'bottom-of-stack symbol' ¢

Ex: **Parenthesis expressions** are recognized by a 1-state fs controler (+ error state, error transitions)



Transition function f: **(**current state, input symbol, pop top of stack**)** -> **(**next state, push stack**)**
(q, (, ε**)** -> **(**q, (**)**, **(** q, [, ε**)** -> **(**q, [**)**, **(** q,), (**)** -> **(**q, ε **)**, **(** q,], [**)** -> **(**q, ε**)**
This PDA accepts whenever the stack is empty. Variations: final states, end-of-input & bottom-of-stack test.

Ex: M accepts (via empty stack) **palindromes with a center marker c**. L = {w c $w^{reverse}$ | w ∈ {0, 1}* }.



LIFO access makes PDAs computationally weak! No deterministic PDA can recognize palindromes **without** an explicit center marker. It doesn't know when to stop pushing input symbols onto the stack, and to start popping. Non-determinism adds some power, but doesn't remove the LIFO bottleneck.

4.4 Nondeterministic pushdown automata (NPDA)

Df NPDA: M= (Q, A, B, f, q0, F)
Q: set of states, q0: initial state, F ⊆ Q: final or accepting states
A: input alphabet; B: stack alphabet (convention: includes a for all a ∈ A , and bottom of stack symbol ©)
f transition function: f: Q x A_ε x B_ε -> 2^{QxB^*}
Reading a stack symbol implies 'pop'. If the stack is not consulted, read ε from the stack.
Variants: accept via empty stack and/or via F; push 1 or more symbols onto stack.

Notice: **non-deterministic PDAs are more powerful acceptors than deterministic PDAs.**

Ex: A 2-state NPDA M accepts (via empty stack) **even palindromes**, L_0 = {w $w^{reverse}$ | w ∈ {0, 1}* }.
(p, 0, ε) -> (p, 0), (p, 1, ε) -> (p, 1) p assumes M is still reading the first half of the input string

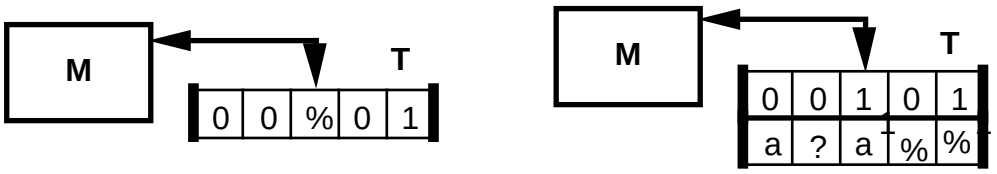
(p, ε, ε) -> (q, ε)	non-deterministic guess of the mipoint
(q, 0, 0) -> (q, ε), (q, 1, 1) -> (q, ε)	q assumes M is reading the second half of the input string.

1) $\forall$ $ww^{reverse}$, w $\in$ {0, 1}*, $\exists$ a sequence of transitions driven by $ww^{reverse}$ that results in an empty stack.
2) $\forall$ z $\neq$ $ww^{reverse}$, no sequence of transitions driven by z results in an empty stack.

Ex "LIFO bottleneck": In contrast to palindromes, it turns out that L = {ww | w $\in$ {0, 1}* } is **not** accepted by any PDA (as explained in the next chapter, this implies that L is not context-free CF).

4.5 Linear bounded automata (LBA)

A linear bounded automaton is a finite automaton with read/write access to a tape T of fixed length. The length of T is determined by the input string w: initially, T contains an input string over some alphabet A. M has a read/write head that can be moved left or right one square at a time, but cannot be moved off the tape, M has a working alphabet B which contains A as a subset, and typically has additional characters. In the example of the figure below at left, A = {0, 1}, B = {0, 1, %, ...}. A tape of length L has a storage capacity of L log |B| bits, which is a constant factor of log |B| / log |A| larger than the storage capacity required to store input strings of length L - thus, the term "linear bounded memory".
It may be convenient to think of M's tape as consisting of several parallel tracks, as shown in the figure at right. A designated track contains the input string to be processed, and might be read-only. Other tracks are read/write tracks used for scratch work. In this model, M's alphabet is a Cartesian product A x B' x B" x ..of the input alphabet A and other track alphabets B', B", ..



A deterministic LBA has the components: M = (Q, A, B, f: Q x B -> Q x B x {L, R, H,..}, q_0, F}.
Q: finite state space; A: input alphabet; B tape alphabet; f: transition function; q_0: initial state; F: final states.
{L, R, H, ..}: tape actions: L = move left, R = move right, H = halt. Optional: "stay put".

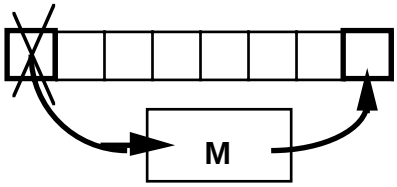
HW 4.2: The word problem for LBAs is decidable.
Describe a decision procedure which, given an LBA M and a word w, decides whether M accepts w.

4.6 Turing machines (see Ch 6) and related automata

A Turing machine (TM) is am FSM that controls an external storage device of unbounded size whose access operations are sufficiently versatile to make the machine "universal" in a sense that will be made precise. It is amazing how little it takes to turn some of the restricted automata discussed in this chapter into TMs!

Exercise: Consider a FSM that controls 2 stacks. Give a rigorous definition that captures this concept, and show that a "2-stack PDA" is equivalent to a FSM that controls an unbounded tape, i.e. a Turing machine.

Creative exercise: A Post machine (Emil Post 1897-1954) or Tag machine is a FSM with a single tape of unbounded length with FIFO access (first-in first-out, as opposed to the LIFO access of a stack). M reads and deletes the letter at the head of the FIFO queue and may append letters to the tail of the queue. Technical detail: there is a special symbol #, not part of the alphabet A of the input string, typically used as a delimiter. Perhaps surprisingly, Post machines are universal, i.e. equivalent in computational power to TMs.



End of Ch 4

5. Context free grammars (CFG) and languages (CFL)

Goals of this chapter: CFGs and CFLs as models of computation that define formal notations as used in programming languages. Properties, strengths and weaknesses of CFLs. Equivalence of CFGs and NPDAs. Non-equivalence of deterministic and non-deterministic PDAs. Parsing. Context sensitive grammars CSG.

5.1 Context free grammars and languages (CFG, CFL)

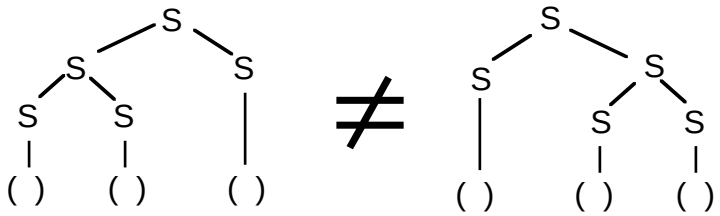
Algol 60 pioneered CFGs and CFLs to define the syntax of programming languages (Backus-Naur Form).
Ex: arithmetic expression E, term T, factor F, primary P, a-op A = {+, -}, m-op M = {., /}, exp-op = *.
E -> T | EAT | AT, T -> F | TMF, F -> P | F*P,
P -> unsigned number | variable | function designator | **(** E **)** [Notice the recursion: E ->* (E)]

Ex Recursive data structures and their traversals:
Binary tree T, leaf L, node N: T -> L | N T T (prefix) or T -> L | T N T (infix) or T -> L | T T N (suffix).
These definitions can be turned directly into recursive traversal procedures, e.g:
procedure traverse (p: ptr); begin if p ≠ nil then begin visit(p); traverse(p.left); traverse(p.right); end; end;

Df CFG: G = (V, A, P, S)
V: non-terminal symbols, "variables"; A: terminal symbols; S ∈ V: start symbol, "sentence";
P: set of productions or **rewriting rules** of the form X -> w, where X ∈ V, w ∈ (V ∪ A)*
Rewriting step: for u, v, x, y, y', z ∈ (V ∪ A)*: u -> v iff u = xyz, v = xy'z and y -> y' ∈ P.
Derivation: "->*" is the transitive, reflexive closure of "->", i.e.
u ->* v iff $\exists\ w_0, w_1, .., w_k$ with k ≥ 0 and u = w_0, w_{j-1} -> w_j, w_k = v.
L(G) context free language generated by G: L(G) = {w ∈ A* | S ->* w }.

Ex Symmetric structures: L = { $0^n\ 1^n$ | n ≥ 0 }, or even palindromes L_0 = { w $w^{reversed}$ | w ∈ {0, 1}* }
G(L) = ({S}, {0, 1}, { S -> 0S1, S -> ε}, S); G(L_0) = ({S}, {0, 1}, { S -> 0S0, S -> 1S1, S -> ε}, S)
Palindromes (length even or odd): L_1 = {w | w = $w^{reversed}$ }. G(L_1): add the rules: S -> 0, S -> 1 to G(L_0).

Ex Parenthesis expressions: V = {S}, T = { (,), [,] }, P = { S -> ε, S -> (S), S -> [S], S -> SS }
Sample derivation: S -> SS -> SSS ->* ()[S][] -> ()[SS][] ->* ()[()[]][]
The rule S -> SS makes this grammar **ambiguous** -> useless in practice, since **structure** is important.



Ex Ambiguous structures in natural languages:
"Time flies like an arrow" vs. "Fruit flies like a banana".
"Der Gefangene floh" vs. "Der gefangene Floh".

Bad news: There exist CFLs that are **inherently ambiguous**, i.e. every grammar for them is ambiguous (see Exercise). Moreover, the problem of deciding whether a given CFG G is ambiguous or not, is **undecidable**.
Good news: For practical purposes it is **easy to design unambiguous CFG's.**

Exercise:
a) For the Algol 60 grammar G (simple arithmetic expressions) above, explain the purpose of the rule E -> AT and show examples of its use. Prove or disprove: G is unambiguous.
b) Construct an unambiguous grammar for the language of parenthesis expressions above.
c) The ambiguity of the "dangling else". Several programming languages (e.g. Pascal) assign to nested if-then[-else] statements an ambiguous structure. It is then left to the semantics of the language to disambiguate. Let E denote Boolean expression, S statement, and consider the 2 rules:
S -> **if** E **then** S, and S -> **if** E **then** S **else** S. Discuss the trouble with this grammar, and fix it.
d) Give a CFG for L = { $0^i\ 1^j\ 2^k$ | i = j or j = k }. Try to prove: L is inherently ambiguous.

5.2 Equivalence of CFGs and NPDAs

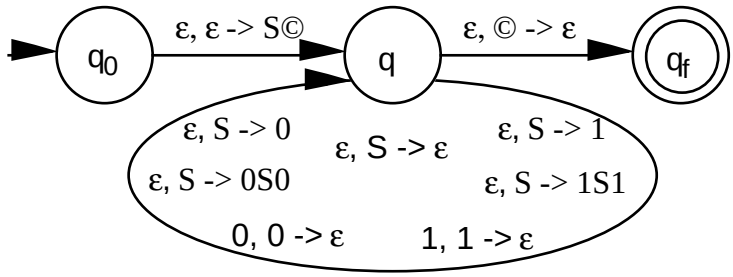
Thm (CFG~ NPDA): **L ⊆ A* is CF iff ∃ NPDA M that accepts L.**

Pf ->: Given CFL L, consider any grammar G(L) for L. Construct NPDA M that simulates all possible derivations of G. M is essentially a single-state FSM, with a state q that applies one of G's rules at a time. The start state q0 initializes the stack with the content S ¢, where S is the start symbol of G, and ¢ is the bottom of stack symbol. This initial stack content means that M aims to read an input that is an instance of S. In general, the current stack content is a sequence of symbols that represent tasks to be accomplished in the characteristic LIFO order (last-in first-out). The task on top of the stack, say a non-terminal X, calls for the next characters of the imput string to be an instance of X. When these characters have been read and verified to be an instance of X, X is popped from the stack, and the new task on top of the stack is started. When ¢ is on top of the stack, i.e. the stack is empty, all tasks generated by the first instance of S have been successfully met, i.e. the input string read so far is an instance of S. M moves to the accept state and stops.

The following transitions lead from q to q:
1) ε, X-> w for each rule X -> w. When X is on top of the stack, replace X by a right-hand side for X.
2) a, a -> ε for each terminal a ∈ A. When terminal a is on top of the stack, read a from the input if possible.

M can be considered to be a non-deterministic parser for G. A formal proof that M accepts precisely L can be done by induction on the length of the derivation of any w ∈ L. QED

Ex L = palindromes: G(L) = ({S}, {0, 1}, { S -> 0S0, S -> 1S1, S -> 0, S -> 1, S -> ε}, S)



When q, q' are shown explicitly, the transition:
(q, a, b) -> (q', v), v ∈ B*
is abbreviated as:
a, b -> v

Pf <- (sketch): Given NPDA M, construct CFG G that generates L(M).
For simplicity's sake, first transform M to have the following features: 1) a single accept state, 2) empty stack before accepting, and 3) each transition either pushes a single symbol, or pops a single symbol, but not both. For each pair of states p, q ∈ Q, introduce non-terminal V_{pq}. **Invariant: V_{pq} generates all strings w that take M from p with an empty stack to q with an empty stack.** The idea is to relate all V_{pq} to each other in a way that reflects how labeled paths and subpaths through M's state space relate to each other. LIFO stack access implies: any w ∈ V_{pq} will lead M from p to q regardless of the stack content at p, and leave the stack at q in the same condition as it was at p. Different w's ∈ L(V_{pq}) may do this in different ways, which leads to different rules of G:
1) The stack may be empty only in p and in q, never in between. If so, w = a v b, for some a, b ∈ A, v ∈ A*. And M includes the transitions (p, a, ε) -> (r, t) and (s,b, t) -> (q, ε). Add the rules: V_{pq} -> a V_{rs} b
2) The stack may be empty at some point between p and in q, in state r.
For each triple p, q, r ∈ Q, add the rules: V_{pq} -> V_{pr} V_{rq}.
3) For each p ∈ Q, add the rule V_{pp} -> ε .

5.3 Normal forms

When trying to prove that all objects in some class C have a given property, it is often useful to first prove that each object O in C can be transformed to some equivalent object O' in some subclass C' of C. Thereafter, the argument can be limited to the restrictions that define the subclass C'.

Any CFG can be transformed into a number of "normal forms" (NF) that are (almost!) equivalent.

Chomsky normal form (right-hand sides are short):

All rules are of the form X -> YZ or X -> a, for some non-terminals X, Y, Z ∈ V and terminal a ∈ A

Thm: Every CFG G can be transformed into a Chomsky NF G' such that L(G') = L(G) - {ε}.
Pf idea: repeatedly replace a rule X -> v w, |v| ≥ 1, |w| ≥ 2 by X -> YZ, Y -> v, Z -> w, where Y and Z are new non-terminals used only in these new rules. The right-hand-sides v and w are shorter than the original v w.

Greibach normal form (at every step, produce 1 terminal symbol at the far left - useful for parsing!):
All rules are of the form X -> a w, for some terminal a ∈ A, and some w ∈ V*

Thm: Every CFG G can be transformed into a Greibach NF G' such that L(G') = L(G) - {ε}.
Pf idea: for a rule X -> Y w, ask whether Y can ever produce a terminal at the far left, i.e. Y ->* a v. If so, replace X -> Y w by rules such as X -> a v w. If not, X -> Y w can be omitted, as it will never lead to a terminating derivation.

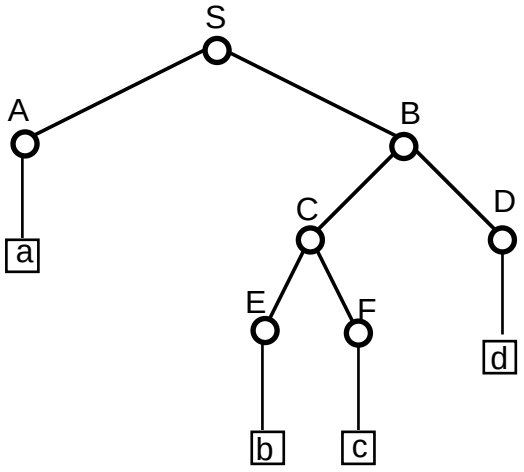
5.4 The pumping lemma for CFLs

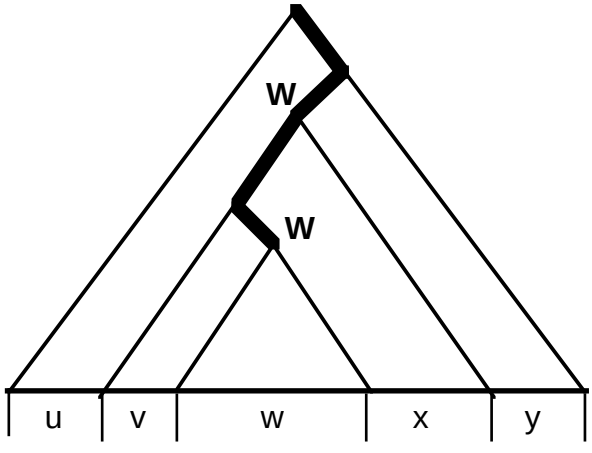
Recall the pumping lemma for regular languages, a mathematically precise statement of the intuitive notion "a FSM can count at most up to some constant n". It says that for any regular language L, any sufficiently long word w in L can be split into 3 parts, w = x y z, such that all strings $x\ y^k\ z$,, for any k ≥ 0, are also in L.

PDAs, which correspond to CFGs, can count arbitrarily high. But the LIFO access limitation implies that the stack can essentially be used only as one single counter. The pumping lemma for CFLs is a precise statement of this limitation.

Thm: For every CFL L there is a constant n such that every z ∈ L of length |z| ≥ n can be written as z = u v w x y such that the following holds:
1) vx ≠ ε , 2) |vwx| ≤ n, and 3) $\mathbf{u\ v^k\ w\ x^k\ y} \in$ **L for all k ≥ 0.**

Pf: Given CFL L, choose any G = G(L) in Chomsky NF. This implies that the parse tree of any z ∈ L is a binary tree, as shown in the figure below at left. The length n of the string at the leaves and the height h of a binary tree are related by h ≥ log n, i.e. a long string requires a tall parse tree. By choosing the critical length $\mathbf{n = 2^{|V|+1}}$ we force the height of the parse trees considered to be **h ≥ |V| + 1**. On a root-to-leaf path of length ≥ |V| + 1 we encounter at least |V| + 1 nodes labeled by non-terminals. Since G has only |V| distinct non-terminals, this implies that on some long root-to-leaf path we must encounter 2 nodes labeled with the same non-terminal, say W, as shown at right.





For two such occurrences of W (in particular, the two lowest ones), and for some u, v, y, x, w ∈ A*, we have: S ->* u W y, W ->* v W x and W ->* w. But then we also have W ->* v^2 W x^2, and in general, W ->* v^k W x^k, and S ->* u v^k W x^k y and S ->* u v^k w x^k y for all k ≥ 0, **QED**.

How to use the pumping lemma to prove that PDAs are poor counters!
Thm: L = { $0^k\ 1^k\ 2^k$ / k ≥ 0 } is **not context free**.

Pf: Assume L is CF, let n be the constant asserted by the pumping lemma.
Consider z = $0^n 1^n 2^n$ = u v w x y. Although we don't know where vwx is positioned within z, the assertion $|vwx| \leq n$ implies that vwx contains at most two distinct letters among 0, 1, 2. In other words, one or two of the three letters 0, 1, 2 is missing in vwx. Now consider u v^2 w x^2 y. By the pumping lemma, it must be in L. The assertion $|vx| \geq 1$ implies that u v^2 w x^2 y is longer than uvwxy. But uvwxy had an equal number of 0s, 1s, and 2s, whereas u v^2 w x^2 y cannot, since only one or two of the three distinct letters increased in number. This contradiction proves the thm.

5.5 Closure properties of the class of CFLs

Thm (CFL closure properties): The class of CFLs over an alphabet A is closed under the regular operations union, catenation, and Kleene star.

Pf: Given CFLs L, L' $\subseteq$ A*, consider any grammars G, G' that generate L and L', respectively. Combine G and G' appropriately to obtain grammars for L $\cup$ L', L L', and L*. E.g, if G = (V, A, P, S), we obtain G(L*) = (V $\cup$ {S_0}, A, P $\cup$ { S_0 -> S S_0 , S_0 -> ε }, S_0).

For regular languages, we proved closure under complement by appealing to **deterministic** FAs as acceptors. For these, changing all accepting states to non-accepting, and vice versa, yields the complement of the language accepted. This reasoning fails for CFL's, because deterministic PDAs accept only a subclass of CFLs. For non-deterministic PDAs, changing accepting states to non-accepting, and vice versa, does not produce the complement of the language accepted. Indeed, closure under complement does not hold for CFLs.

Thm : The class of CFLs over an alphabet A is **not** closed under intersection and complement.

Pf: Consider CFLs L_0 = { $0^m 1^m 2^n$ | m, n ≥ 1 } and L_1 = { $0^m 1^n 2^n$ | m, n ≥ 1 }.
$L_0 \cap L_1$ = { $0^k 1^k 2^k$ | k ≥ 1 } is **not** CF, as we proved in the previous section using the pumping lemma.

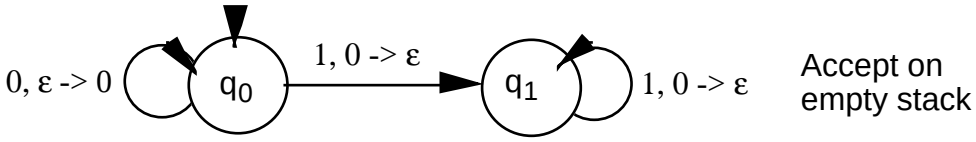
If the class of CFLs were closed under complement, it would also be closed under intersection, because of the identity: L $\cap$ L' = ¬(¬L $\cup$ ¬L'). QED

5.6 The "word problem". CFL parsing in time $O(n^3)$ by means of dynamic programming

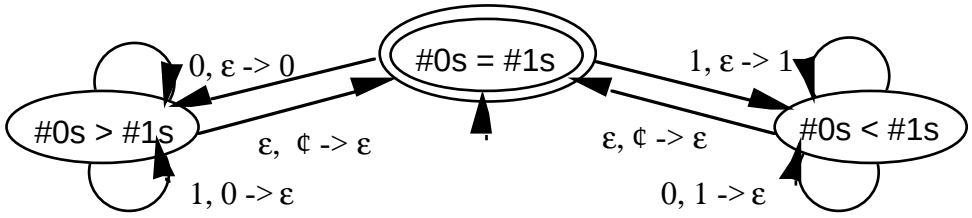
The word problem: given G and w ∈ A*, decide whether w ∈ L(G). More precisely: is there an algorithm that applies to any grammar G in some given class of grammars, and any w ∈ A*, to decide whether w ∈ L(G)?

Many algorithms solve the word problem for CFGs, e.g: a) construct an NPDA that accepts L(G), or b) convert G to Greibach NF and enumerate all derivations of length ≤ |w| to see whether any of them generates w.

Ex1: L = { $0^k 1^k$ | k ≥ 1 }. G: S -> 01 | 0 S1. Use "0" as a stack symbol to count the number of 0s.



Ex2: L = {w ∈ {0, 1}* | #0s = #1s }. G: S -> ε | 0 Y' | 1 Z', Y' -> 1 S | 0 Y' Y', Z' -> 0 S | 1 Z' Z'
Invariant: Y' generates any string with an extra 1, Z' generates any string with an extra 0.
The production Z' -> 0 S | 1 Z' Z' means that Z' has two ways to meet its goal: either produce a 0 now and follow up with a string in S, i.e with an equal number of 0s and 1s; or produce a 1 but create two new tasks Z'.



For CFGs there is a “bottom up” algorithm (Cocke, Younger, Kasami) that systematically computes all possible parse trees of all contiguous substrings of the string w to be parsed, and works in time O($|w|^3$). We illustrate the idea of the CYK algorithm using the following example:

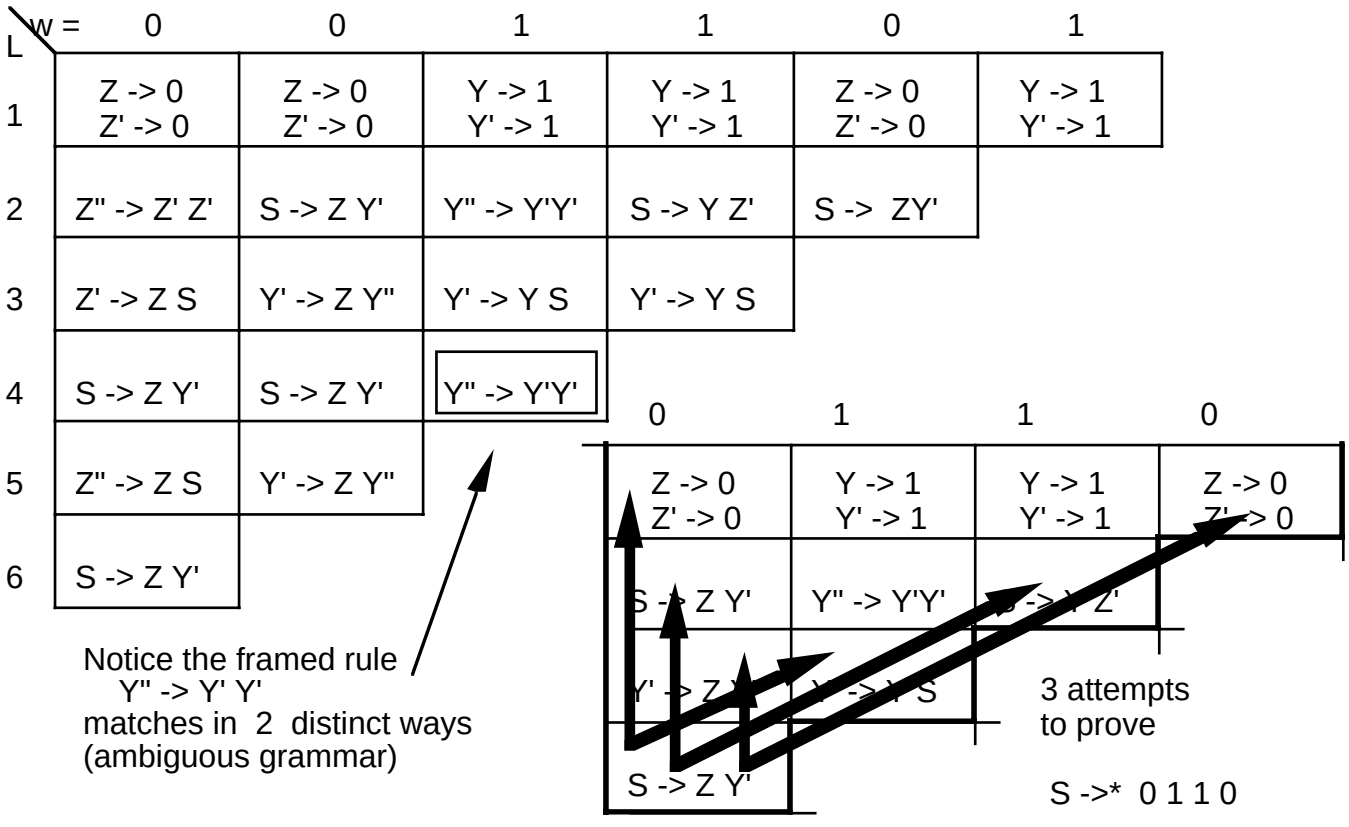
Ex2a: L = {w ∈ {0, 1 }$^+$ | #0s = #1s }. G: S -> 0 Y’ | 1 Z’, Y’ -> 1S | 0 Y’ Y’, Z’ -> 0 S | 1 Z’ Z’

Convert G to Chomsky NF. For the sake of formality, introduce Y that generates a single 1, similarly for Z and 0. Shorten the right hand side 0 Z’ Z’ by introducing a non terminal Z” -> Z’ Z’, and similarly Y” -> Y’Y’. Every w ∈ Z” can be written as w = u v, u ∈ Z’, v ∈ Z’. As we read w from left to write, there comes an index k where #1s = #0s +1, and that prefix of w can be taken as u. The remainder v has again #1s = #0s +1.

The grammar below maintains the invariants: Y generates a single “1”; Y’ generates any string with an extra “1”; Y” generates any string with 2 extra “1”. Analogously for Z, Z’, Z” and “0”.

S -> Z Y’ | Y Z’	start with a 0 and remember to generate an extra 1, or start with a 1 and ...
Z -> 0, Y -> 1	Z and Y are mere formalities
Z’ -> 0 | Z S | Y Z”	produce an extra 0 now, or produce a 1 and remember to generate 2 extra 0s
Y’ -> 1 | Y S | Z Y”	produce an extra 1 now, or produce a 0 and remember to generate 2 extra 1s
Z” -> Z’ Z’, Y” -> Y’Y’	split the job of generating 2 extra 0s or 2 extra 1s

The following table parses a word w = 001101 with |w| = n. Each of the n (n+1)/2 entries corresponds to a substring of w. Entry (L, i) records all the parse trees of the substring of length L that begins at index i. The entries for L = 1 correspond to rules that produce a single terminal, the other entries to rules that produce 2 non-terminals.



The picture at the lower right shows that for each entry at level L, we must try (L-1) distinct ways of splitting that entry’s substring into 2 parts. Since (L-1) < n and there are n (n+1)/2 entries to compute, the CYK parser works in time O(n^3).

Useful CFLs, such as parts of programming languages, should be designed in such a way that they admit more efficient parsers, preferably parsers that work in linear time. LR(k) grammars and languages are a subset of CFGs and CFLs that can be parsed in a single scan from left to right, with a look-ahead of k symbols.

5.7 Context sensitive grammars and languages

The rewriting rules B -> w of a CFG imply that a non-terminal B can be replaced by a word $w \in (V \cup A)^*$ "in any context". In contrast, a context sensitive grammar (CSG) has rules of the form
u B v -> u w v, where $u, v, w \in (V \cup A)^*$, implying that B can be replaced by w only in the context "u on the left, v on the right".

It turns out that this definition is equivalent (apart from the nullstring ε) to requiring that any CSG rule be of the form v -> w, where $v, w \in (V \cup A)^*$, and $|v| \leq |w|$. This monotonicity property (in any derivation, the current string never gets shorter) implies that the word problem for CSLs: "given CSG G and given w, is $w \in L(G)$?" is decidable. An exhaustive enumeration of all derivations up to the length |w| settles the issue.

As an example of the greater power of CSGs over CFGs, consider:

Thm: $L = \{ 0^k 1^k 2^k / k \geq 1 \}$ is context sensitive
The following CSG generates L. Function of the non-terminals V = {S, B, C, Y, Z}: each Y and Z generates a 1 or a 0 at the proper time; B initially marks the beginning (left end) of the string, and later converts the Zs into 0s; C is a counter that ensures an equal number of 0s, 1s, 2s are generated. Non-terminals play a similar role as markers in Markov algorithms. Whereas the latter have a deterministic control structure, grammars are non-deterministic.

S -> B K 2	at the last step in any derivation, B K generates 01, balancing this '2'
K -> Z Y K 2	counter K generates $(ZY)^k 2^k$
K -> C	when k has been fixed, C may start converting Ys into 1s
Y Z -> Z Y	Zs may move towards the left, Ys towards the right at any time
B Z -> 0 B	B may convert a Z into a 0 and shift it left at any time
Y C -> C 1	C may convert a Y into a 1 and shift it right at any time
B C -> 01	when B and C meet, all permutations, shifts and conversions have been done

Hw 5.2: Prove that $L = \{ww \mid w \in \{0, 1\}^* \}$ is **not** context free. Optional: show L is context sensitive.

Ex: Recall the grammar G1 of arithmetic expressions, e.g. in the simplified form:
E -> T | EAT, T -> F | TMF, F -> N | V | (E) , A -> + | - , M -> * | /
For simplicity's sake, we limit numbers to single bits, i.e. N -> 0 | 1, and use only 3 variables, V -> x | y | z

a) Extend G1 to a grammar G2 that includes function terms, such as f(x) and g(1 - x/y)
Use only 3 function symbols defined in a new production H -> f | g | h

b) Extend G2 to a grammar G3 that includes integration terms, such as S [a, b] f(x) dx, a linearized form of "integral from a to b of f(x) dx".

c) Discuss the strengths and weaknesses of CFGs as tools to solve the tasks a) and b).

Ex: Let $L = \{ww \mid w \in \{0, 1\}^* \}$
a) Prove that L is **not** context free, and b) prove that L is context sensitive.

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”)

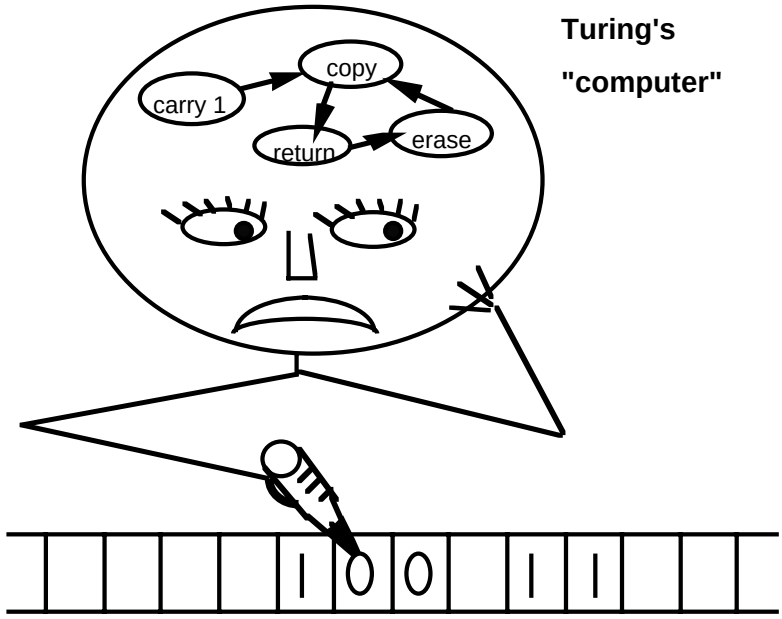
David Hilbert’s 10-th problem (1900): “10. Entscheidung der Lösbarkeit einer diophantischen Gleichung. Eine diophantischen Gleichung mit irgendwelchen Unbekannten und mit ganzen rationalen Zahlkoeffizienten sei vorgelegt; **man soll ein Verfahren angeben**, nach welchem sich mittels einer endlichen Anzahl 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: $x^2 + y^2 + 1$ 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, $P(x) = a0 + a1\,x + a2\,x^2 + \ldots + an\,x^n$, such a decision process was known, based on formulas that provably bound the absolute value of any root x0, e.g. $|x0| \le 1 + \max |ai / an|$, 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.



“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"

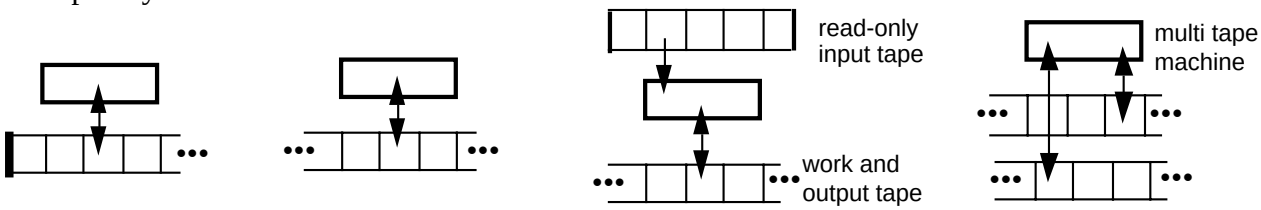
The question of effective computability was suddenly raised in the 30s and investigated by several logicians using different formalisms because of the crisis in the foundation of mathematics produced by

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:

$$M = (Q, A, f: Q \times A \to Q \times A \times \{L, R, -, H\}, q_0, F\}.$$

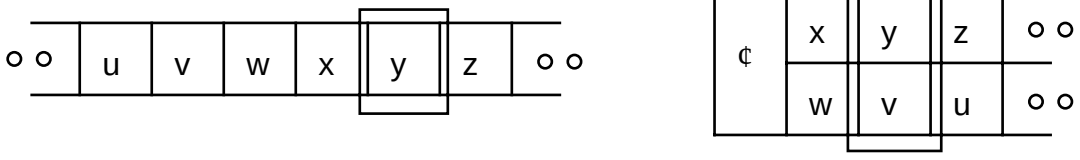
Q: finite state space; A: finite alphabet; f: transition function; q_0: initial state; $F \subseteq A$ accepting states.
{L, R, -, H}: tape actions: L = move left, R = move right, - stay put, optional H = halt.
If a tape action calls for the read/write head to drop off the fixed end of the tape, this action is ignored.

Non-deterministic TM: replace $f: Q \times A \to Q \times A \times \{L, R, ..\}$ by $f: Q \times A \to 2^{Q \times A \times \{L, R, ..\}}$.
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, q_0) 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, q_0), construct M1 = (Q', A', f ', q_0') 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 q_0' 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.

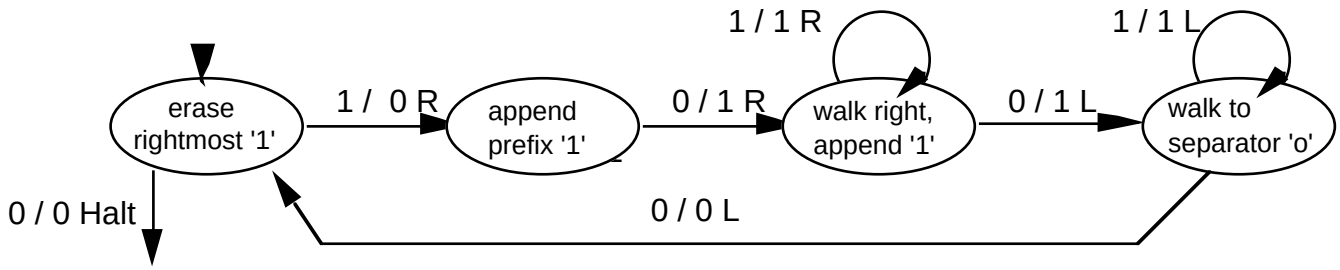
Universal TM, UTM: A universal TM U is an interpreter that reads the description <M> of any arbitrary TM M and faithfully executes operations on data D precisely as M does. For single-tape TMs, imagine that <M> is written at the beginning of the tape, followed by D.

"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

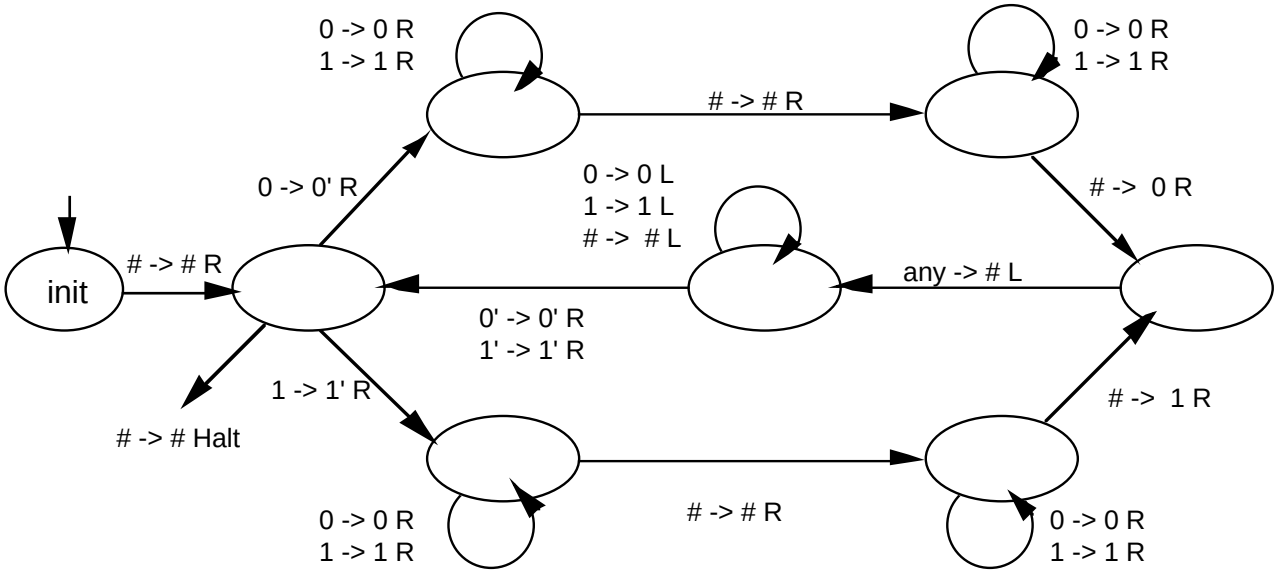
Ex1: f(x) = 2x in unary notation ($x \geq 1$): double the length of a string of contiguous 1's
Configurations. A **bold** symbol indicates it is currently being scanned by the read-write head.

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

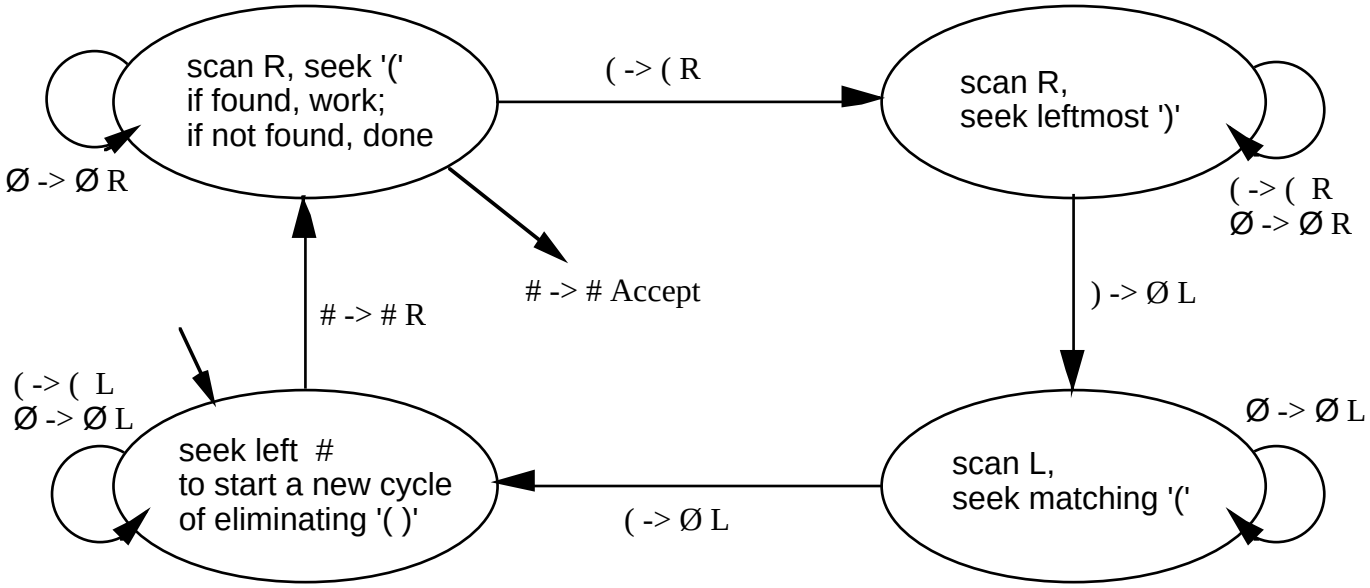
Initial:	**#** 1 0 1 1 # #
Intermediate:	# 1' 0' **1** 1 # 1 0 # (some bits have been replaced by 1' or 0' and have been copied)
Final:	**#** 1' 0' 1' 1' **#** 1 0 1 1 # (should be followed by a clean up phase where 1' is changed to 1)



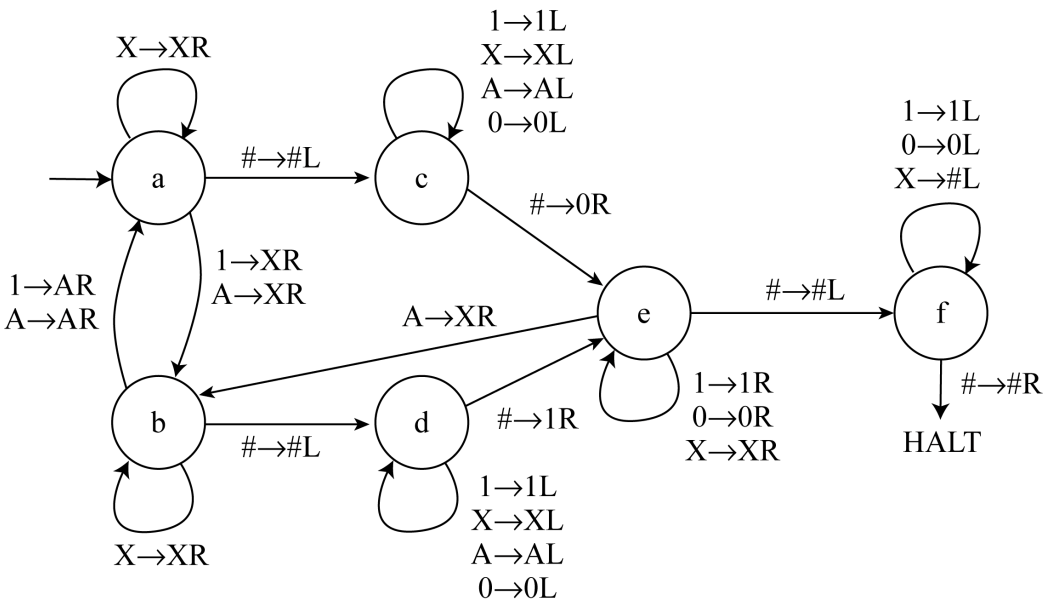
Ex3 parentheses expressions: For a simple task such as checking the syntactic correctness of parentheses expressions, the special purpose device PDA, with its push-pop access to a stack, is more convenient. A PDA throws away a pair of matching parentheses as soon as it has been discovered and never looks 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

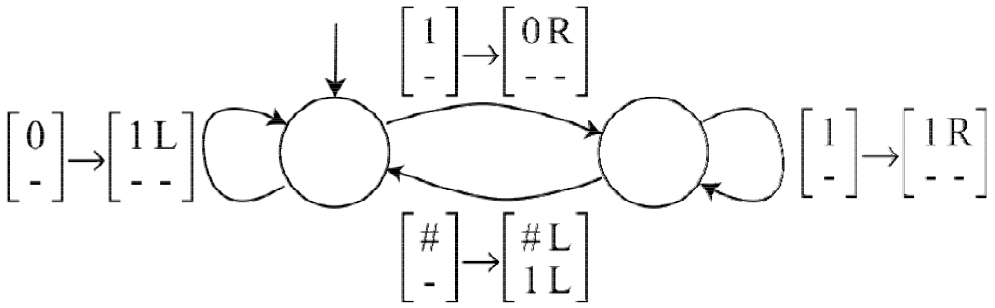


Example

```
..####11111##..
      a
..####XAXAX##..
           b
..###1XAXAX##..
     e
..###1XXXAX##..
           a
..##01XXXAX##..
    e
..##01XXXXX##..
           b
..#101XXXXX##..
   e
..#101XXXXX##..
           f
..#101#######..
```

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):

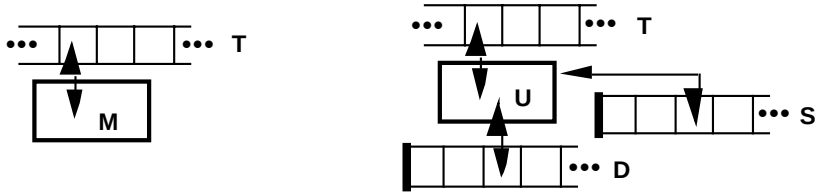
Configurations. A **bold** symbol indicates it is currently being scanned by the read-write head.
Initial: . ..0 **1** 1 1 X 1 1 0 ..
Intermediate: ..0 0 0 1 X 1 1 1" 1" 1" 0 ..
Intermediate: ..0 0 0 0 X 1 1 1" 1" 1" 1" 1" 1" 0 ..
Final: ..0 0 0 0 X 1 1 1 1 1 1 1 0 ..
The multiplier x = 3 is to the left 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.

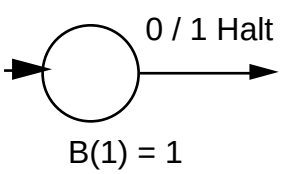
Georg Cantor (1845-1918, founder of set theory) showed that the set of functions from natural numbers to natural numbers is uncountable. Cantor's diagonalization technique is a standard tool used to prove undecidability results. By way of contradiction, assume that all functions from natural numbers 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, ...

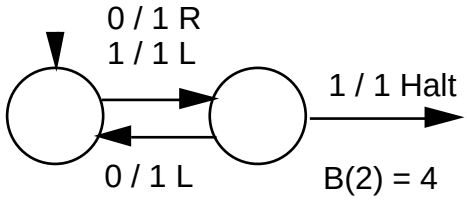
f1:	**f1(1)**	f1(2)	f1(3)	f1(4)	...	f1:	**f1(1) + 1**	f1(2)	f1(3)	f1(4)	...
f2:	f2(1)	**f2(2)**	f2(3)	f2(4)	...	f2:	f2(1)	**f2(2) + 1**	f2(3)	f2(4)	...
f3:	f3(1)	f3(2)	**f3(3)**	f3(4)	...	f3:	f3(1)	f3(2)	**f3(3) + 1**	f3(4)	.
..											

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 a modified version of M that produces f(x) in unary notation might 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 $f(x) \le B(\lceil \log x \rceil + c + k)$. 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), 10^{9999} } 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) ≤ 10^{9999}, 10^{9999}, ... A huge but conceptually trivial TM can store the first m function values and the constant 10^{9999}, 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 10^{9999} 1s; we could never be sure whether we missed a stronger competitor Mc with c < k states that prints at least 10^{9999} 1s. Among the TMs with c states, there might have been some that printed more than 10^{9999} 1s, but we dismissed them as not halting. And although it is possible to prove some specific TM to be 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}, q_0, q_a , q_r }.
Tape actions: L = move left, R = move right. Designated states: q_0 = start state, q_a = accept, q_r = reject.
A Turing machine acceptor has **3 possible answers: accept, reject, loop**.

Df: M accepts w ∈ A* iff M's computation on w ends in q_a . L(M) = { w ∈ A* / M accepts w }
Df: L ⊆ A* is **Turing recognizable** (recursively enumerable) 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 ⊆ 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

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

Ex: L_{NPDA} = { <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):
L_{TM} = { <M, w> | M is a TM , M accepts w } is **not** decidable.

Pf: Assume L_{TM} is decidable, and D is a decider for L_{TM} , 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 who 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 L_{TM} , 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". 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 L_{TM} .

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 H exists for L_{TM}. 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 L_{TM} , 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.
Thus, the existence of H implies the existence of R, a decider for the word problem L_{TM} , a problem we have just proven to be undecidable. Contradiction -> QED.

Hw 6.4: 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" L_{TM} = { <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 $\in$ 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, $n \geq 1$, 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" L_{TM} , 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 6.5: 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 $0 \le x \le 1$, 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 $k \ge 1$, prints the k-th digit bk of x and halts.

Hw 6.6: 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 $k \ge 1$, 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, $y \ne xk$ 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

7. Complexity: P & NP

Goals of this chapter: Given a model of computation and a measure of complexity of computations, it is possible to define the **inherent complexity of a class of problems**. This is a **lower bound on the complexity of any algorithm** that solves instances of the given problem class. The model of computation considered are Turing machines, the complexity measure is time as measured by the number of transitions executed sequentially (other complexity measures, e.g. space, are only mentioned). Understand the drastic difference between deterministic and non-deterministic computation, and concepts such as P, NP, NP-complete, NP-hard. Satisfiability and other NP-complete problems.

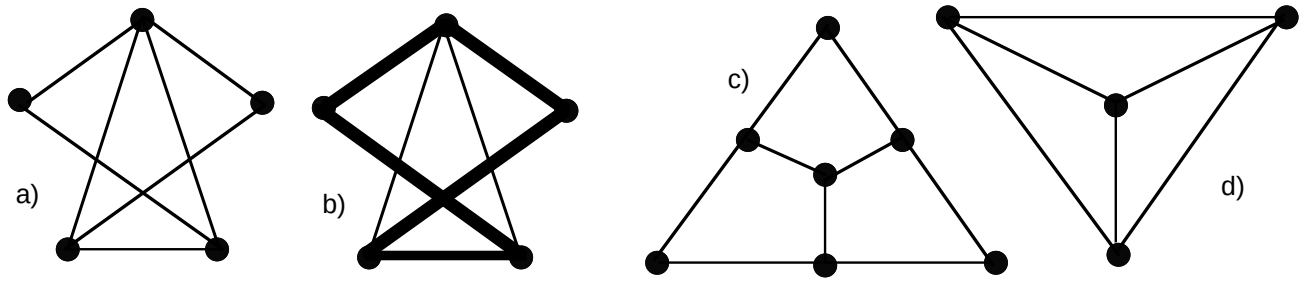
7.1 Decision problems, optimization problems: examples

Df: **Hamiltonian cycle** in a graph G = (V, E): a cycle that contains each of the vertices in V (exactly once)

Df: **Traveling salesman problem (TSP):**
Weighted graph G = (V, E, w: E -> Reals), V = {1, .., n}, E = { (i, j) | i < j } (often the complete graph Kn).
Weight (or length) of a path or cycle = sum of the weights of its edges.
Given G= (V, E, w) find a shortest Hamiltonian cycles, if any exist.

Df: A **clique** in a graph G = (V, E): a subset V' $\subseteq$ V such that for all u, v $\in$ V', (u, v) $\in$ E.
|V'| is the size of the clique. A clique of size k is called a k-clique.

Df: **Clique problems**: Given G= (V, E), answer questions about the existence of cliques, find maximum clique, enumerate all cliques.



Ex: the graph a) shown at left has exactly 1 Hamiltonian cycle, highlighted in b). The graph c) has none. The complete graph K4 has 3 Hamiltonian cycles. d) The complete graph Kn has (n-1)! / 2 Hamiltonian cycles
Ex: Graph a) has three maximum 3-cliques. The only cliques in graph c) are the vertices and the edges, i.e. the 1-cliques and the 2-cliques. Graph d) is a 4-clique, and every subset of its vertices is a clique.

As the examples above show, "basically the same" combinatorial problem often comes in different versions, some of which are optimization problems, others decision problems. The three problems: 1) "find a maximum clique", 2) "what is the size of a maximum clique", 3) "is there a k-clique for a given value k?" clearly call for similar computational techniques. Combinatorial problems in practice are usually concerned with optimization, i.e. we search for an object which is best according to a given objective function. The complexity theory of this chapter, on the other hand, is mostly concerned with decision problems. To appreciate its practical relevance it is important to understand that optimization problems and decision problems are often related: any information about one type of problem provides useful knowledge about the other. We distinguish 3 types of problems of seemingly increasing difficulty:

Decision problems:
• Given G, does G have a Hamiltonian cycle? Given G and k, does G have a k-clique?
Given G = (V, E, w) and a real number B, does G have a Hamiltonian cycle of length $\leq$ B?
Finding the answer to a decision problem is often hard, whereas verifying a positive answer is often easy: we are shown an object and merely have to verify that it meets the specifications (e.g. trace the cycle shown in b). Decision problems are naturally formalized in terms of machines accepting languages, as follows: problem instances (e.g. graphs) are coded as strings, and the code words of all instances that have the answer YES (e.g. have a Hamiltonian cycle) form the language to be accepted.

Optimization problems:
• Given G, construct a maximum clique.
• TSP: Given Kn = (V, E, w) find a Hamiltonian cycle of minimal total length.
Both problems, of finding the answer and verifying it, are usually hard. If I claim to show you a maximum clique, and it contains k vertices, how do you verify that I haven't missed a bigger one? Do you have to enumerate all the subsets of k+1 vertices to be sure that there is no (k+1)-clique? Nevertheless, verifying is usually easier than finding a solution, because the claimed solution provides a bound that eliminates many suboptimal candidates.

Enumeration problems:
• Given G, construct all Hamiltonian cycles, or all cliques, or all maximum cliques.
Enumeration problems are solved by exhaustive search techniques such as backtrack. They are time consuming but often conceptually simple, except when the objects must be enumerated in a prescribed order. Enumeration is the technique of last resort for solving decision problems or optimization problems that admit no efficient algorithms, or for which no efficient algorithm is known. It is an expensive technique, since the number of objects to be examined often grows exponentially with the length of their description.

The complexity theory of this chapter is formulated in terms of decision problems,
Whereas in practice, most problems of interest are optimization problems!

In practice, hardly any problem calls for a simple yes/no answer - almost all computations call for constructing a solution object of greater complexity than a single bit! It is therefore important to realize that the complexity theory of decision problems can be made to apply to optimization problems also. The way to do this is to associate with an optimization problem a related decision problem such that the complexity of the latter has implications for the complexity of the former. We distinguish 3 types of optimization problems:
a) **optimization problem** (in the strict sense): find an optimal solution
b) **evaluation problem**: determine the value of an optimal solution
c) **bound problem**: given a bound B, determine whether the value of an optimal solution is above or below B.

Intuitively, a solution to problem a) gives more information than b), which in turn contains more information than c). Indeed, for all "reasonable problems", an efficient solution to an optimization problem in the strict sense a) implies an efficient solution to the corresponding evaluation problem b) by simply computing the cost of this solution. Of course, one can construct pathological examples where the evaluation function is very complex, or even non-computable. In that case, given a solution s, we might not be able to compute its value v(s). But such cases don't seem to occur in practice.
In turn, an efficient solution to an evaluation problem b) implies an efficient solution to the corresponding bound problem c) - just by comparing 2 numbers.

The opposite direction is less obvious. Nevertheless, we show that a solution to the bound problem c) helps in finding a solution to the evaluation problem b), which in turn helps in finding a solution to the optimization problem a). There is a considerable cost involved, but this cost is "polynomial", and in the generous complexity measure of this chapter we ignore polynomial costs. Although c) contains less information than b), and b) usually less than a), it is not surprising that even a small amount of information can be exploited to speed up the solution to a harder problem, as follows.

To exploit c) in solving b), we use the fact that the possible values of a combinatorial problem are usually discrete and can be taken to be integers. Assume we can solve the bound problem c) within time T. For the corresponding evaluation problem b) one usually knows a priori that the value lies within a certain range [L, U] of integers. Using binary search, we solve the evaluation problem with log | U - L | calls to the bound problem c), and hence in time T log | U - L |.

Ex: TSP on weighted graph Kn = (V, E, w: E -> Reals), |V| = n, |E| = n-choose-2.
Use c) to solve b). A tour or Hamiltonian cycle in a graph of n vertices has exactly n edges. Thus, the sum S of the n longest edges is an upper bound for the length of the optimal tour. On the other hand, the sums of all m-choose-n subsets of n edges is a finite set of numbers, and the minimal non-zero difference d among two of these numbers defines the granularity of the tour lengths. Two distinct tours either have the same value, or their lengths differ by at least d. Thus, a binary search that computes log(S / d) bound problems determines the length (value) of an optimal tour.

To exploit b) in solving a), we use the fact that the solution of a combinatorial problem is a discrete structure D that consists of predefined elementary parts, such as vertices or edges in a graph, say P1, .., Pn. And that the evaluation function v(D) is often monotonic with respect to subsets of D = P1, .., Pn - if we drop some parts in a maximization problem, the value v(D') of the subset D' may be less than v(D). Assume that we can solve

the evaluation problem b) within time T, and that there are n elementary parts P1, .., Pn, each of which is a potential component of a solution. First, solve the evaluation problem for the given data D consisting of n elementary parts to obtain the target value v(D). Thereafter, for each part Pi, obtain the value v(D - Pi). If v(D - Pi) = v(D), part Pi is not an essential component of an opti,mal solution, and can be omitted. Whereas if v(D - Pi) < v(D), part Pi is an essential component of any solution object and must e retained. By applying this test of essentiality to each of the n parts Pi, we construct a solution object within time (n+1) T.

Ex: TSP on weighted graph Kn = (V, E, w: E -> Reals), |V| = n, |E| = n-choose-2.
Use b) to solve a). First, find the length L of an optimal tour. Thereafter, for each edge ej find the length Lj of an optimal tour when ej is prohibited; this is achieved by temporarily assigning to ej a huge weight, eg making ej longer than the sum of all other edges. In a non-degenerate configuration, when the optimal tour is unique, the set of n edges j for which Lj < L form exactly this unique optimal tour. For a degenerate configuration such as the equilateral triangle with its center shown in Fig d) above, which has 3 isomorphic optimal tours, the rule is a bit more complicated (no single edge is essential).

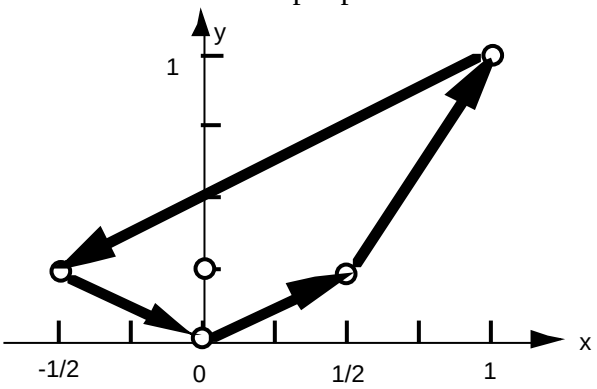
7.2 Problem reduction, classes of equivalent problems

Scientific advances often follow from a successful attempt to establish similarities and relationships among seemingly different problems. In mathematics, such relationships are formalized in two ways:
- problem A can be reduced to problem B, $A \leq B$, implies that if we can solve B, we can also solve A
- problems A and B are equivalent, $A \leq B$ and $B \leq A$, implies that if we can solve either problem, we can also solve the other, roughly with an equivalent investment of resources. Whereas here we use "≤" in an intuitive sense, section 7.5 presents a precise definition of polynomial reducibility, denoted by "≤p".
This chapter is all about reducing some problems to others, and to characterizing some prominent classes of equivalent problems. Consider some examples.

Sorting as a key data management operation. Answering almost any query about an unstructured set D of data items requires, at least, looking at each data item. If the set D is organized according to some useful structure, on the other hand, many important operations can be done faster than in linear time. Sorting D according to some useful total order, for example, enables binary search to find, in logarithmic time, any query item given by its key. As a data management operation of major importance, sorting has been studied extensively. In the RAM model of computation, the worst case complexity of sorting is $\Theta(n \log n)$. This fact can be used in two ways to bound the complexity of many other problems: 1) to show that some other problem P can be solved in time $O(n \log n)$, because P reduces to sorting, and 2) to show that some other problem Q requires time $\Omega(n \log n)$, because sorting reduces to Q. Two examples:

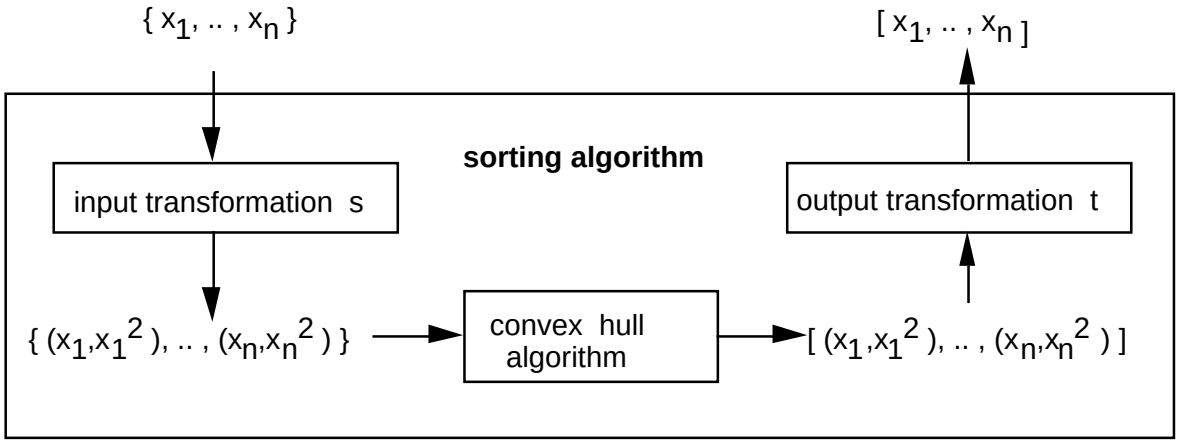
1) Finding the median of n numbers x1, .., xn can be done in time $O(n \log n)$. After sorting x1, .., xn in an array, we find the median in constant time as x [n div 2]. A more sophisticated analysis shows that the median can be determined in linear time, but this does not detract from the fact that we can quickly and easily establish an upper bound of $O(n \log n)$.

2) Finding the convex hull of n points in the plane requires time $\Omega(n \log n)$. An algorithm for computing the convex hull of n points in the plane takes as input a set $\{ (x_1, y_1), .. , (x_n, y_n) \}$ of coordinates and delivers as ouput a **sequence** $[(x_{i1}, y_{i1}), .. , (x_{ik}, y_{ik})]$ of extremal points, i.e. those on the convex hull, ordered clockwise or counter-clockwise. The figure shows 5 unordered input points and the ordered convex hull.

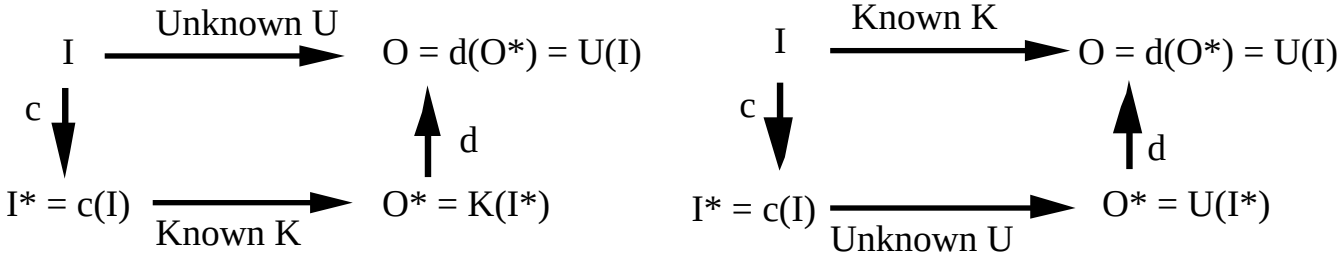


We now reduce the sorting problem of known complexity $\Omega(n \log n)$ to the convex hull problem as shown in the next figure. Given a set $\{ x_1, .. , x_n \}$ of reals, we generate the set $\{ (x_i, x_i^2), .. , (x_n, y_n^2) \}$ of point coordinates. Due to the convexity of the functiuon x^2, all points lie on the convex hull. Thus, a convex hull

algorithm returns the **sequence** of points ordered by increasing x-coordinates. After dropping the irrelevant y-coordinates, we obtain the sorted sequence $[x_1, .. , x_n]$ .Thus, a convex hull algorithm can be used to obtain a ridiculously complicated sorting algorithm, but that is not our aim. The aim is to show that the convex hull problem is at least as complex as sorting - if it was significantly easier, then sorting would be easier, too.



The following figure explains problem reduction in general. Given a problem U of unknown complexity, and a problem and algorithm K of known complexity. A coding function c transforms a given input I for one problem to input I* for the other, and the decoding function d transforms output O* into output O. This type of problem reduction is of interest only when the complexity of c and of d is no greater than the complexity of K, and when the input transformation I* = c(I) leaves the size n = | I | of the input data asymptotically unchanged: $| I^* | \in \Theta (n)$. | This is what we assume in the following.



Deriving an **upper bound**. At left we construct a solution to the problem U of unknown complexity by the detour U(I) = d(K(c(I))). Letting | c |, | K | and | d | stand for the time required for c, K and d, respectively, this detour proves the upper bound $| U | \leq | c | + | K | + | d |$. Under the usual assumption that $| c | \leq | K |$ and $| d | \leq | K |$, this bound asymptotically reduces to $| U | \leq | K |$. More explicitly: if we know that K(n), c(n), d(n) are all in O(f(n)), then also $U(n) \in O(f(n))$.

Deriving a **lower bound**. At right we argue that a problem U of unknown complexity | U | is asymptotically at least as complex as a problem K for which we know a lower bound, $K(n) \in \Omega(f(n))$. Counter-intuitively, we reduce the known problem K to the unknown problem U, obtaining the inequality $| K | \leq | c | + | U | + | d |$ and $| U | \geq | K | - | c | - | d |$. If $K(n) \in \Omega(f(n))$ and c and d are strictly less complex than K, i.e. $c(n) \in o(f(n))$, $d(n) \in o(f(n))$, then we obtain the lower bound $U(n) \in \Omega(f(n))$.

7.3 The class P of problems solvable in polynomial time

Practically all standard combinatorial algorithms presented in a course on Algorithms and Data Structures run in polynomial time. They are sequential algorithms that terminate after a number of computational steps that is bounded by some polynomial p(n) in the size n of the input data, as measured by the number of data items that define the problem. A computational step is any operation that takes constant time, i.e. time independent of n.

In practical algorithm analysis there is a fair amount of leeway in the definition of "computational step" and "data item". For example, an integer may be considered a single data item, regardless of its magnitude, and any arithmetic operation on integers as a single step. This is reasonable when we know a priori that all numbers generated are bounded by some integer "maxint", and is unreasonable for computations that generate numbers of unbounded magnitude.

In complexity theory based on Turing machines the definition is clear: a computational step is a transition executed, a data item is a character of the alphabet, read or written on a square of tape. The alphabet is usually chosen to be {0, 1} and the size of data is measured in bits. When studying the class **P** of problems solvable in polynomial time, we only consider **deterministic TMs that halt on all inputs**.

Let t_M: A* -> Integers be the number of steps executed by M on input $x \in A^*$.

This chapter deals with TMs whose running time is bounded by some polynomial in the length of the input.

Df: TM M is or runs in polynomial time iff $\exists$ polynomial p such $\forall x \in A^*$: $t_M(x) \leq p(|x|)$

Df: **P** = { $L \subseteq A^*$ | $\exists$ TM M, $\exists$ polynomial p such that L = L(M) and $\forall x \in A^*$: $t_M(x) \leq p(|x|)$ }

Notice that we do **not** specify the precise version of TM to be used, in particular the number of tapes of M is left open. This may be surprising in view of the fact that a multi-tape TM is much faster than a single-tape TM. A detailed analysis shows that "much faster" is polynomially bounded: a single-tape TM S can simulate any multi-tape TM M with at most a polynomial slow-down: for any multi-tape TM M there is a single-tape TM S and a polynomial p such that for all $x \in A^*$, $t_S(x) \leq p(t_M(x))$. This simulation property, and the fact that a polynomial of a polynomial is again a polynomial, makes the definition of the class P extremely robust.

The question arises whether the generous accounting that ignores polynomial speed-ups or slow-downs is of practical relevance. After all, these are much greater differences than ignoring constant factors as one does routinely in asymptotics. The answer is a definite YES, based on several considerations:

1) Practical computing uses random access memory, not sequential storage such as tapes. Thus, the issue of how many tapes are used does not even arise. The theory is formulated in terms of TMs, rather than more realistic models of computation such as conventional programming languages, for the sake of mathematical precision. And it turns out that the slowness of tape manipulation gets absorbed, in comparison with a RAM model, by the polynomial transformations we ignore so generously.

2) Most practical algorithms are of low degree, such as $O(n)$, $O(n \log n)$, $O(n^2)$, or $O(n^3)$. Low-degree polynomials grow slowly enough that the corresponding algorithms are computationally feasible for many values of n that occur in practice. E.g. for $n = 1000$, $n^3 = 10^9$ is a number of moderate size when compared to processor clock rates of 1 GHz and memories of 1 GByte. Polynomial growth rates are exceedingly slow compared to exponential growth (consider 2^{1000}).

3) This complexity theory, like any theory at all, is a model that mirrors some aspects of reality well, and others poorly. It is the responsibility of the programmer or algorithm designer to determine in each specific case, whether or not an algoritm "in **P**" is practical or not.

Examples of problems (perhaps?) in **P**:

1) Every context-free language is in **P**. In Ch5 we saw an $O(n^3)$ parsing algorithm that solves the word problem for CFLs, where n is the length of the input string.

2) The complexity of problems that involve integers depends on the representation. Fortunately, with the usual radix representation, the choice of radix $r > 1$ is immaterial (why?). But if we choose an exotic notation, everything might change. For example, if integers were given as a list of their prime factors, many arithmetic problems would become easier. Paradoxically, if integers were given in the unwieldy unary notation, some things might also become "easier" according to the measure of this chapter. This is because the length of the unary representation $< k >_1$ of k is exponential in the length of the radix $r \geq 2$ representation $< k >_r$. Given an exponentially longer input, a polynomial-time TM is allowed to use an exponentially longer computation time as compared to the "same" problem given in the form of a concise input.

The following example illustrates the fact that the complexity of arithmetic problems depends on the number representation chosen. The assumed difficulty of factoring an integer lies at the core of modern cryptography. Factoring algorithms known today require, in the worst case, time exponential in the length, i.e. number of bits, of the radix representation of the number to be factored. But there is no proof that factoring is NP-hard - according to today's knowledge, there might exist polynomial-time factoring algorithms. This possibility gained plausibility when it was proven that primality, i.e. the problem of determining whether a natural number (represented in radix notation) is prime or composite, is in P (M. Agrawal, N. Kayal, N. Saxena: PRIMES is in

P, Indian Institute of Technology Kanpur, August 2002). If we represent numbers in unary notation, on the other hand, then it is straightforward to design a polynomial-time factoring algorithm (why?).

7.4 The class NP of problems solvable in non-deterministic polynomial time

"NP" stands for "non-deterministic polynomial". It is instructive to introduce two different but equivalent definitions of NP, because each definition highlights a different key aspect of NP. The original definition explicitly introduces non-deterministic TMs:

Df 1: **NP** = { $L \subseteq A^*$ | $\exists$ **N**TM N, $\exists$ polynomial p such that L = L(N) and $\forall x \in A^*$: $t_N(x) \leq p(|x|)$ }

Notice that this differs from the definition of **P** only in the single letter "**N**" in the phrase "$\exists$ **N**TM N ..", indicating that we mean non-deterministic Turing machines. We had seen that deterministic and non-deterministic TMs are equally powerful in the presence of unbounded resources of time and memory. But there is a huge difference in terms of the **time** they take for certain computations. A NTM pursues simultaneously a number of computation paths that can grow exponentially with the length of the computation.

Because of many computation paths pursued simultaneously we must redefine the function t_N: A* -> Integers that measures the number of steps executed. An input $x \in A^*$ may be accepted along a short path as well as along a long path, and some other paths may not terminate. Therefore we define **$t_N(x)$ as the minimum number of steps executed along any accepting path for x**.

Whereas the original definition of NP in terms of non-deterministic TMs has the intuitive interpretation via parallel computation, an equivalent definition based on deterministic TMs is perhaps technically simpler to handle. The fact that these two definitions are equivalent provides two different ways of looking at **NP**.

The motivation for this second definition of **NP** comes from the observation that it may be difficult to decide whether a string meeting certain specifications **exists**; but that it is often easier to decide whether or not a **given string** meets the specifications. In other words, finding a solution, or merely determining whether a solution exists, is harder than checking a proposed solution's correctness. The fact that this intuitive argument leads to a rigorous definition of **NP** is surprising and useful!

In the following definition, the language L describes a problem class, e.g. all graphs with a desired property ; the string w describes a problem instance, e.g. a specific graph G; the string c = c(w), called a "certificate for w" or a witness, plays the role of a key that "unlocks w": the pair w, c is easier to check than w alone!

Example: for the problems of Hamiltonian cycles and cliques introduced in 7.1, w = <G> is the representation of graph G, and the certificate c is the representation of a cycle or a clique, respectively. Given c, it is easy to verify that c represents a cycle or a clique in G.

Df: a verifier for $L \subseteq A^*$ is a **deterministic** TM V with the property that
$L = \{ w \mid \exists\ c(w) \in A^*$ such that V accepts $< w, c > \}$

The string c = c(w) is called a certificate for w's membership in L. <w, c> denotes a representation of the pair (w, c) as a single string, e.g. w#c, where a reserved symbol # separates w from its certificate. The idea behind this concept of certificate is that it is easy to verify $w \in L$ if you are given w's certificate c. If not, you would have to try all strings in the hope of finding the right vertificate, a process that may not terminate. We formalize the phrase "easy to verify" by requiring that a verifier V is (or runs in) polynomial-time.

Df 2: NP is the class of languages that have deterministic polynomial-time verifiers.

Definitions Df1 and Df2 provide two different interpretations of the same class NP: Df1 in terms of **parallel computation**, Df2 in terms of **sequential verification**. A technical advantage of Df2 is that the theory of NP can be developed using only deterministic TMs, which are simpler than NTMs. We now present an intuitive argument that the two definitions are equivalent.

Df1 -> Df2 ("simulate and check"): If L is in **NP** according to Df1, there is a NDTM N with the following property: every $w \in L$ has an accepting path p of length polynomial in |w|. It is tempting to merely remove the transitions of N that are not executed along the path p, but this does not turn N into a deterministic TM, because different w's may use different transitions. Instead, we construct a DTM M that reads as input a string <w, N, P> that represents the following information: the word w to be accepted, the NTM N, the

accepting path p. M is a universal TM that interprets the description of N: at every step along the path p, M checks that this step is one of the possible transitions of N on input w. In this construction, the pair <N, p> is w's certificate.

Df2 -> Df1 ("guess and check": We present the idea using as an example the problem of Hamiltonian cycle. According to Df2, there is a DTM M that verifies <G, c> in polynomial time, where the certificate c is a Hamiltonian cycle of G. Construct a NTM N that first generates in parallel lists of n numbers v1, v2, .., vn, vi $\in$ 1 .. n. Each vi can be generated one bit at a time in logarithmic time, the entire list in time n log n. Each list is checked for syntactic correctness in the sense that there are no repetitions of vertex numbers, and that any 2 consecutive vertices, including vn v1, are edges of G. This checking is identical to the verification done by the deterministic TM M, but the non-deterministic N does it on all lists simultaneously. If any one list is verified as being a Hamiltonian cycle, then N accepts <G>.

From both definitions Df1 and Df2 it is evident that $\mathbf{P} \subseteq \mathbf{NP}$. From Df1 because a DTM is a special case of a NTM. From Df2 because P is the class of languages L that require no certificate, so we can take the nullstring as a trivial certificate for all words of L.

P = NP ? is one of the outstanding questions of the theory of computations, so far unanswered. All experience and intuition suggests that $\mathbf{P \neq NP}$. Non-determinism provides computing power that grows exponentially with the length of the sequential computation. In the time that a DTM performs t steps, a NTM can perform at least $\mathbf{2^t}$ steps. Whereas a DTM must backtrack to explore alternatives, a NTM explores all alternatives at the same time. The belief that $P \neq NP$ is so strong that today many researchers prove theorems that end with the caveat "..unless P = NP". By this they mean to imply that the theorem is to be considered empirically true, i.e. as true as the conjecture $P \neq NP$. But mathematics has its own strict rules as to what constitutes a theorem and a proof, and today there is no proof in sight to give the conjecture $P \neq NP$ the status of a theorem.

7.5. Polynomial time reducibility, NP-hard, NP-complete

We add some formal definitions to the general concept of problem reduction of 7.2.

Df: A function f: A* -> A* is polynomial-time computable iff there is a polynomial p and a DTM M which, when started with w on its tape, halts with f(w) on its tape, and $t_M(w) \leq p(|w|)$.

Df: L is polynomial-time reducible to L', denoted by $L \leq_p L'$ iff
there is polynomial-time computable f: A* -> A* such that $\forall w \in A^*$, $w \in L$ iff $f(w) \in L'$.

In other words, the question $w \in L$? can be answered by deciding $f(w) \in L'$. Thus, the complexity of deciding membership in L is at most the complexity of evaluating f plus the complexity of deciding membership in L'. Since we generously ignore polynomial times, this justifies the notation $L \leq_p L'$.

A remarkable fact makes the theory of NP interesting and rich. With respect to the class **NP** and the notion of polynomial-time reducibility, there are "hardest problems" in the sense that all decision problems in **NP** are reducible to any one of these"hardest problems".

Df: L' is **NP-hard** iff $\forall L \in NP$, $L \leq_p L'$
Df: L' is **NP-complete** iff $L' \in NP$ and L' is NP-hard

7.6 Satisfiability of Boolean expressions (SAT) is NP-complete

SAT = satisfiability of Boolean expressions: given an arbitrary Boolean expression E over variables
x1, x2, .. , x_d, is there an assignment of truth values to x1, x2, .. that makes E true?

It does not matter what Boolean operators occur, conventionally one considers And $\wedge$, Or $\vee$, Not $\neg$.

SAT is the prototypical "hard problem". I.e. the problem that is generally proven to be NP-complete "from scratch", whereafter all other problems to be proven NP-complete are reduced to SAT. The theory of NP-completeness began with the key theorem **(Cook 1971): SAT is NP-complete.**

Given this central role of SAT it is useful to develop an intuitive understanding of the nature of the problem, including details of measurement and "easy" versions of SAT.

1) It does not matter what Boolean operators occur, conventionally one considers And $\wedge$, Or $\vee$, Not $\neg$. Special forms, eg CNF

2) The length n of E can be measured by the number of characters of E. It is more convenient, however, to first eliminate "Not-chains" $\neg\neg\neg$ .. using the identity $\neg \neg x = x$, and to measure the length n of E by the number n of **occurences** of variables (d denotes the number of **distinct** variables $x1, x2, .. x_d$ in E, $d \leq n$). It is convenient to avoid mentioning the unary operator $\neg$ explicitly by introducing, for each variable x, its negation $\neg x$ as a dependent variable. A variable and its negation are called **literals**. Thus, an expression E with d variables has 2d distinct literals that may occur in E.

3) Any Boolean expression E can be evaluated in linear time. Given truth values for $x1, x2, .. x_d$, the n occurrences of literals are the leaves of a binary tree with n -1 internal nodes, each of which represents a binary Boolean operator. The bottom-up evaluation of this tree requires n-1 operations.

4) Satisfiability of expressions over a constant number d of distinct variables can be decided in linear time. By trying all 2^d assignments of truth values to the variables, SAT can be decided in time $O(2^d n)$, a bound that is linear in n and exponential in d. If we consider d constant, 2^d is also a constant - hence this version of the satisfiability problem can be solved in linear time! This argument shows that, if SAT turns out to be a difficult problem, this is due to an unbounded growth of the number of distinct variables. Indeed, if d grows proportionately to n, the argument above yields an exponential upper bound of $O(2^n)$.

5) In order to express SAT as a language, choose a suitable alphabet A and a coding scheme that assigns to any expression E a word code(E) $\in$ A*, and define: SAT = { code(E) | E is a satisfiable Boolean expression }

Thm (Cook 1971): SAT is NP-complete - the key theorem at the origin of all other NP-complete results.

1) SAT is in **NP**. Given E and an assignment of truth values as a certificate of satisfiability, the truth value of E can be computed and verified in linear time.

2) Idea of proof that SAT is NP-hard. Let L be any language $\in$ **NP** , we have to prove $L \leq_p$ SAT. $L \in$ **NP** means there is a deterministic polynomial-time verifier M with L = L(M). We construct in polynomial time a Boolean expression E = E(M, w) with the property that $w \in L$ iff E is satisfiable. The intuition is that E(M, w) simulates, i.e. describes, the computation of M on w step by step. Although E is "huge", a detailed analysis shows that the length |E| of E is polynomially bounded in the length |w|, and moreover, that E can be computed in time polynomial in |w|.

The following idea guides the construction of E(M, w). The computation of M on any word w is a sequence of configurations, where a configuration contains the state of the tape, the state of M, and the location of M's read/write head on the tape. This sequence can be described, or modeled, by a (huge) set of Boolean variables. For example, we introduce a variable s(99, 1) which we want to be true iff after 99 steps M is in state 1. The transition from step 99 to step 100 can be modeled by some expression that relates all the variables involved, such as t(99, 0), which we want to be true iff after 99 steps M is reading symbol 0 on its tape. Thus, the behavior of any TM M on w can be modeled by a huge expression which is true iff M accepts w. QED.

Satisfiability remains NP-complete even if we restrict the class of Boolean expressions in various ways. A standard and convenient normal form, that still allows us to express any given Boolean function, is the **conjunctive normal form, CNF** .

Df: a CNF expression E is the conjunction of clauses Ci, where each clause is the disjunction of literals Lj, and a literal is a single variable or the negation of a single variable, e.g. x:
$E = C1 \wedge C2 \wedge C3 \wedge ...$ where $Ci = (L1 \vee L2 \vee L3 \vee ..)$ and $Lk = x$ or $Lk = \neg x$.

Any Boolean expression E can be transformed into an equivalent expression in CNF using the identities:
- de Morgan's law: $\neg (x \wedge y) = \neg x \vee \neg y$, $\neg (x \vee y) = \neg x \wedge \neg y$
- distributive law: $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$

Let CNF be the problem of deciding the satisfiability of CNF expressions. Since SAT is NP-complete and any Boolean expression E can be transformed into an equivalent expression F in CNF, it is obvious that SAT is

reducible to CNF, and it is plausible to conjecture that SAT ≤p CNF, and hence that CNF is NP-complete. The conjecture is correct, but the plausibility is too simple minded. What is wrong with the following "pseudo-proof"? In order to prove SAT ≤p CNF, transform any given Boolean expression E in arbitrary form into an equivalent CNF expression F; E is satisfiable iff F is satisfiable. "QED"

The error in the argument above is hidden in the letter 'p' in '≤p', which we defined as a **polynomial-time reduction**. The transformation of an arbitrary Boolean expression E into an equivalent CNF expression F may cause F to be **exponentially longer** than E, and thus to require time exponential in the length of E merely to write down the result! Witness the following formula E in disjunctive normal form (DNF):

$$E = x11 \wedge x12 \vee x21 \wedge x22 \vee x31 \wedge x32 \vee \ldots \vee xn1 \wedge xn2$$

It consists of 2n literals each of which appears only once, thus it has length 2n as measured by the number of occurrences of literals. repeated application of the distributive laws yields

$$F = \bigwedge_{(j1 \,..\, jn \,=\, 1,2)} (x1j_1 \vee x2j_2 \vee x3j_3 \vee \ldots \vee x\,nj_n)$$

F is an AND of 2^n clauses, where each clause has exactly one variable xj for j = 1 .. n, with the second index chosen independently of the choice of all other second indices. Thus, the length of F is $n\,2^n$, a transformation that cannot be done in polynomial time. (Exercise: prove that E and F are equivalent).

We will now provide a correct proof that CNF is NP-complete, as a consequence of the stronger assertion that 3-CNF is NP-complete.

Df: A 3-CNF expression is the special case of a CNF expression where each clause has 3 literals ("exactly 3" or "at most 3" is the same - why?).

Because of its restricted form, 3-CNF is conceptually simpler than SAT, and so it is easier to reduce it to new problems that we aim to prove NP-complete. We aim to show that 3-CNF is NP-complete by reducing SAT to 3-CNF, i.e. SAT ≤p 3-CNF. In contrast to the argument used in the wrong proof above, which failed because it ignores exponential lengthening, the following construction keeps the length of F short by using the trick that E and F need not be equivalent as Boolean expressions!

Thm: 3-CNF SAT is NP-complete

Pf idea: reduce SAT to 3-CNF SAT, SAT ≤p 3-CNF SAT. To any Boolean expression E we assign in polynomial time a 3-CNF expression F that is **equivalent in the weak sense** that either both E and F are satisfiable, or neither is. Notice that E and F need not be equivalent as Boolean expressions, i.e. they need not represent the same function! They merely behave the same w.r.t. satisfiability.

Given E, we construct F in 4 steps, illustrated using the example $E = \neg (\neg x \wedge (y \vee z))$

1) Use de Morgan's law to push negations to the leaves of the expression tree:
   $E1 = x \vee \neg (y \vee z) = x \vee (\neg y \wedge \neg z)$

2) Assign a new Boolean variable to each internal node of the expression tree, i.e. to each occurrence of an operator, and use the Boolean operator 'equivalence' ⇔ to state the fact that this variable must be the result of the corresponding operation: $u \Leftrightarrow (\neg y \wedge \neg z)$, $w \Leftrightarrow x \vee u$

3) Construct an expression E2 that states that the root of the expression tree must be true, traces the evaluation of the entire tree, node by node, and combines all these assertions using ANDs:
   $E2 = w \wedge (w \Leftrightarrow (x \vee u)) \wedge (u \Leftrightarrow (\neg y \wedge \neg z))$.

   E2 and E are equivalent in the weak sense of simultaneous satisfiability. If E is satisfiable, then E2 is also, by simply assigning to the new variables u and w the result of the corresponding operation. Conversely, if E2 is satisfiable, then E is also, using the same values of the original variables x, y, z as appear in E2.
   Notice that E2 is in conjunctive form at the outermost level, but its subexpressions are not, so we need a last transformation step.

4) Recall the Boolean identity for implication: $a \Rightarrow b = \neg a \vee b$ to derive the identities:
   $a \Leftrightarrow (b \vee c) = (a \vee \neg b) \wedge (a \vee \neg c) \wedge (\neg a \vee b \vee c)$
   $a \Leftrightarrow (b \wedge c) = (\neg a \vee b) \wedge (\neg a \vee c) \wedge (a \vee \neg b \vee \neg c)$
   Using these identities on the subexpressions, E2 gets transformed into F in 3-CNF:
   $F = w \wedge (w \vee \neg x) \wedge (w \vee \neg u) \wedge (\neg w \vee x \vee u) \wedge (\neg u \vee x) \wedge (\neg u \vee z) \wedge (u \vee y \vee z)$

Each of the four transformation steps can be done in linear time and lengthens the expression by at most a constant factor. Thus, the reduction of a Boolean expression E in general form to one, F, in 3-CNF can be done in polynomial time. QED

Notice the critical role of the integer '3'in 3-CNF: we need to express the result of a binary operator, such as $w \Leftrightarrow x \vee u$, which naturally involves three literals. Thus, it is no surprise that the technique used in the proof above fails to work for 2-CNF. Indeed, **2-CNF is in P** (Exercise: look up the proof in some textbook).

In analogy to CNF we define the **disjunctive normal form DNF** as an OR of terms, each of which is an AND of literals: $E = T1 \vee T2 \vee T3 \vee ...$ where $Ti = (L1 \wedge L2 \wedge L3 \wedge ..)$

Notice: **DNF-SAT** is in **P**, in fact DNF-SAT can be decided in linear time (consider $xy'z \vee x\ x' \vee ...$).
Does this mean that the NP-completeness of SAT is merely a matter of representation? No!
Exercise: Show that the **Falsifiability of DNF-SAT is NP-complete**.
This relationship between CNF and DNF implies that the transformation DNF <-> CNF must be hard.

7.7 Hundreds of well-known problems are NP-complete

3-CNF SAT is the starting point of chains of problem reduction theorems to show that hundreds of other well known decision problems are NP complete. The problem CLIQUE is an example: Given a graph G and an integer k, does G contain a clique of size $\geq k$?

Thm: CLIQUE is NP-complete

Pf: Show that 3-CNF $\leq_p$ CLIQUE. Given a 3-CNF expression F, construct a graph G = (V, E) and an integer k such that F is satisfiable iff G has a k-clique.

Let $F = (z11 \vee z12 \vee z13) \wedge (z21 \vee z22 \vee z23) \wedge ... \wedge (zm1 \vee zm2 \vee zm3)$, where each zij is a literal.
To each occurrence of a literal we assign a vertex, i.e. V = { (1,1), (1,2), (1,3), ... , (m, 1), (m, 2), (m, 3) }
We introduce an edge ((i, j) (p, q)) iff $i \neq p$ (the two literals are in different clauses)
and $zij \neq \neg zpq$ (the 2 literals do not clash, i.e. both can be made true under the same assignment).
Finally, let k, the desired clique size, be = m, the number of clauses.
With this construction of G we observe that F is satisfiable via an assignment A
iff 1) each clause contains a literal that is true under A, say $z_{1,\ j1},\ z_{2,\ j2}, ...,\ z_{m,\ jm}$
iff 2) there are literals $z_{1,\ j1},\ z_{2,\ j2}, ...,\ z_{m,\ jm}$ no 2 of which are negations of each other
iff 3) there are vertices (1, j1), (2, j2), ... , (m, jm) that are pairwise connected by an edge
iff 4) G has a k-clique.

7.8 The P versus NP Problem

The Clay Mathematics Institute of Cambridge, Massachusetts (CMI, www.claymath.org, www.claymath.org/Millennium_Prize_Problems/P_vs_NP) has named seven "Millennium Prize Problems." The Scientific Advisory Board of CMI selected these problems, focusing on important classic questions that have resisted solution over the years. The Board of Directors of CMI designated a $7 million prize fund for the solution to these problems, with $1 million allocated to each. The first in the list is the "P versus NP Problem", deescribed as follows:

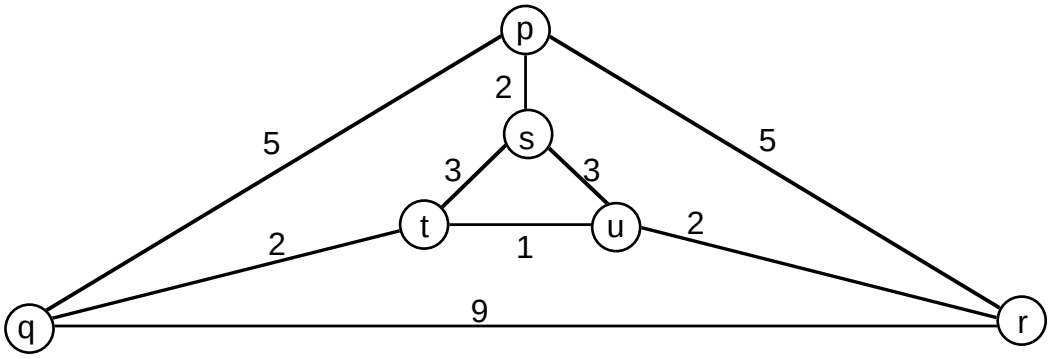
"Suppose that you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP-problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair from taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be

quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971. ”

Hw 7.1: Polynomal-time problem reduction: HC ≤p TSP

A Hamiltonian cycle in a graph G(V, E) is a simple closed path that touches every vertex exactly once. The Hamiltonian Cycle Problem HC asks you to decide, given a Graph G, whether or not G has a Hamiltonian cycle.We consider the Traveling Salesman Problem TSP in the form of a decision problem. Given a weighted graph G(V, E, w), where w: E -> Reals defines edge weights, and given a real number B: is there a traveling salesman tour (i.e. a Hamiltonian cycle in G) of total weight ≤ B?

a) Decide whether the weighted graph shown here has a tour of length ≤ 21. If yes, show it, if not, say why.



Complexity theory represents a class D of decision problems by coding each problem instance d by a string c(d) over some alphabet A. c(d) is a code that uniquely defines d. The class D and the code c partition A* into 3 subsets: 1) the set of strings that are not codewords c(d) for any problem instance d ("syntax error"), 2) the set of codewords c(d) for instances d with a negative answer, and 3) the set L(D) of codewords c(d) for instances d with a positive answer. Solving the decision problem means constructing a Turing machine that always halts and accepts the language L(D).

b) Define a coding scheme to represent weighted graphs and show the codeword that your scheme assigns to the example of TSP-problem b).

c) Recall the definition of “L1 ≤p L2”, i.e. language L1 is polynomial-time reducible to language L2. Show in general how HC is polynomial-time reducible to TSP, i.e. HC ≤p TSP, or more precisely, L(HC) ≤p L(TSP). Hint: Given a graph G as input to HC, construct an appropriate weighted graph G’and an appropriate bound B as input to TSP.

Hw 7.2: CNF ≤p 3-CNF

Show that the satisfiability problem CNF can be reduced in polynomial time to its special case 3-CNF.
Hint: Given a CNF expression E that contains long clauses, any clause C with > 3 literals,

$C = (x1 \vee x2 \vee x3 \vee .. \vee xk)$

can be broken into a number of shorter clauses by repeated application of the following transformation that introduces additional variables, z:

$C' = (x1 \vee x2 \vee z) \wedge (x3 \vee .. \vee \neg z)$

Show 1) that E and the 3-CNF expression F that results from repeated application of this transformation are equivalent in the weak sense that E is satisfiable iff F is satisfiable; 2) that F can be constructed from E in polynomial time; and 3), work out an interesting example.

Hw 7.3: Set cover is NP-complete

Set cover problem: Given a finite set U (“universe”), n subsets S1, S2, .., Sn ⊆ U, and an integer k ≤ n, does there exist a selection of k subsets Sj_1, Sj_2, .., Sj_k (among the given subsets) whose union is U?

Your task: search the library or the web to find a proof of the theorem ‘Set cover is NP-complete’, understand it, an be prepared to present this proof to your assistant. If you don’t succeed with ‘Set cover’, do the same for some other NP-complete problem of your choice.

End of chapter