CDM

Memoryless Machines

Klaus Sutner

Carnegie Mellon University

Spring 2023

1 Zero Space

2 Finite State Machines

3 DFA Decision Problems

So far, we have blissfully ignored physical limitations. For example, we pretend that our registers can hold arbitrarily large naturals, and increment/decrement them in one step.

We could avoid this by limiting the registers to k-bit numbers for some fixed k, say, $k = 64$. So we wind up with a particularly bad assembly language.

In our model, this has an unintended side-effect: there are only finitely many possible inputs (recall: the input has to be written into registers).

Alas, from the perspective of computability and even standard complexity, finite problems are utterly trivial.

Claim: Every finite decision problem is decidable in constant time (albeit for entirely the wrong reasons).

More precisely, we can simply hardwire a lookup table that lists the correct answer for each instance.

For example, this RM accepts $\{1, 3, 6, 7\}$.

```
//  X --> Y
    0:   dec X  1  9          5:   dec X  6  9
    1:   dec X  2  8          6:   dec X  7  8
    2:   dec X  3  9          7:   dec X  8  8
    3:   dec X  4  8          8:   inc Y  9
    4:   dec X  5  9          9:   halt
```

This is perfectly correct, but completely useless. We would very much like to distinguish obviously different levels of difficulty. Think about a finite set of instances I, say, all 1000 bit numbers.

- Y_1 all numbers in I divisible by 17.
- Y_2 all prime numbers in I.
- Y_2 all $x \in I$ such that $\{x\}() \downarrow$.

Clearly, these problem are listed in order of increasing complexity, the first one is nearly trivial, the second requires a bit of work and the last one is a hot mess. We will return to this problem when we talk about Kolmogorov-Chaitin complexity.

Convention: For the time being, we will insists that all our problems have infinite instance sets.

We could get around the problem of not being able to write arbitrarily large numbers into a k-bit register by reading the input in k-bit blocks, say, starting with the least significant digits. This is perfectly reasonable from an algorithmic point of view.

Unfortunately, this means we have to increase the instruction set of our machine: we need some read instruction and we need to be able to test whether the input is complete.

This is doable, but not really worth the effort. It is better to switch to Turing machines where this particular type of skulduggery is a bit more natural.

Once we think about the input as a sequence of letters, it makes little sense to insist that the rest of the machine should be based on arithmetic.

> The only thing that really matters is that a k-bit register machine can only assume finitely many internal states.

Recall our configurations $(p, \boldsymbol{z})$ where $p < n$ and the z_i are all k-bit. Clearly there are only finitely many possibilities.

How does this machine compute? Given a state, it reads another letter off the input and changes to a new state. That's all. To describe it, we can use a lookup table for each internal state and input letter.

So our new machines take infinitely many inputs, but they have no internal memory besides the state they are in. For example, they cannot remember the sequence of states that occurs during a computation.

We could relax this requirement and allow the machine to use some amount of memory depending on the size of the input. Say, on input of size n we are allowed to use $s(n)$ extra memory (s for space). This is exactly what real algorithms do and leads to much more powerful types of computation than our devices.

Note that $s(n) = O(n)$ is usually fine, but $s(n) = O(n^2)$ can already get quite tricky (as opposed to running time).

For $s(n) = 0$ we get Zero Space.

This is one of the occasions where Turing machines are better: historically, finite state machines were defined in terms of highly restricted Turing machines—register machines just don't work as well in this situation.

So how would we dumb down a Turing machine to make sure it performs only highly feasible computations?

The central problem with general Turing machines is that we have no way of predicting the amount of tape used during a computation—which could be used to also obtain a bound on the length of the computation, albeit it an exponential one.

So how about simply imposing a bound on the amount of tape that the machine may use? If the machine attempts to use more tape, the computation simply fails.

A fairly natural restriction would be to allow only as much tape as the input takes up originally: think of two special end markers

$$\# x_1 x_2 \ldots x_{n-1} x_n \#$$

where the head is originally positioned at the first symbol of x. The head is not allowed to move beyond the two cells marked $\#$.

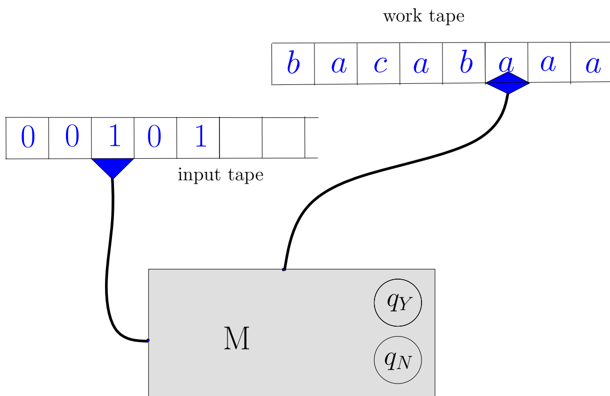
This leads to an important class of machines: linear bounded automata (LBA) and the corresponding class $\mathrm{SPACE}(n)$ of problems solvable in linear space; introduced in their deterministic form in 1960 by Myhill, and in their full nondeterministic form in 1964 by Kuroda.

Unfortunately, LBAs are still much too powerful and complicated. E.g., nondeterministic LBAs can accept every context-sensitive language.

Ominously, the question whether the languages of nondeterministic LBAs are closed under complement was open for three decades before being answered affirmatively by Immerman and Szelepcsényi independently.

We need a more stringent condition, something more restrictive than just linear space.

We can get better control by separating the input from the workspace.



A Turing machine acceptor with separate, read-only input tape.

Suppose the input has length n, some arbitrarily large number.

We can then impose a bound on the size of the worktape, some function $s(n)$ that limits the amount of memory available for the computation.

Popular choices for $s(n)$ are n and $\log n$. Sublinear space complexity is quite interesting, as opposed to sublinear time complexity.

We will be a bit more radical, we go for $s(n) = 0$: Zero Space.

One might suspect that we get less and less compute power as we decrease the memory-size function $s(n)$, say, to $\log n$, $\log\log n$, $\log\log\log n$, and so on.

Here is a major surprise:

Theorem (Hartmanis, Lewis, Stearns 1965)

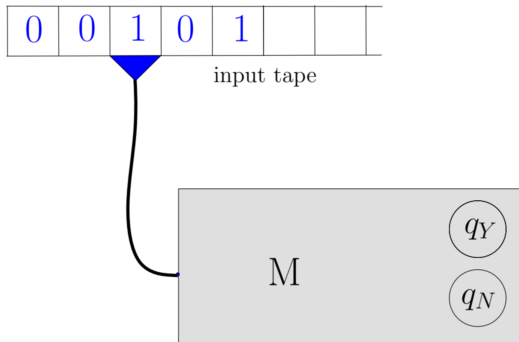
Suppose some decision problem is not solvable in constant space. Then every Turing machine solving the problem requires space $\Omega(\log\log n)$ infinitely often.

Hence, once we get to $s(n) = o(\log\log n)$ we might as well have a worktape of fixed size. The proof is somewhat complicated, see Hartmanis Eal.

But allowing only a constant amount of work tape is already equivalent to allowing no work tape at all: there are only finitely many possible work tape contents and head positions.

We can simply inflate the number of states and think of these finitely many worktape configurations as part of the finite state control, something like $Q \times \Gamma^k$.

In other words, we simply hide the fixed size tape inside the control unit of the Turing machine and, voila, we get Zero Space.



A zero space machine.

One might think that allowing the read-head to move left and right (so that parts of the input can be read repeatedly) would improve the performance of our devices.

As it turns out, one can assume without loss of generality that the read head only moves from left to right: at each step one symbol is scanned and then the head moves right and never returns.

Theorem (Rabin/Scott, Shepherdson)

Every decision problem solved by a constant space two-way machine can already be solved by a constant space one-way machine.

The proof of this result is quite messy, we won't go into details. Unsurprisingly, the one-way version has more internal states. See Rabin/Scott 59.

Note that configurations for these restricted Turing machines are simpler than in the general case, all we need is

$$p\,x \qquad p \in Q, x \in \Sigma^\star$$

where x is the part of the tape to the right of the head: there is no need to keep track of the left part of the tape, we can never go back there.

One step in the computation is then given by a map δ, the so-called transition function, where

$$p\,ax \mathrel{\vert\!\frac{1}{M}}\; q\,x \iff \delta(p, a) = q$$

Example: Parity and Majority

Let's suppose the input is given as a bit sequence $x = x_1 x_2 \dots x_{n-1} x_n$. Here are two classical problems concerning these sequences:

- **Parity:** Is the number of 1-bits in x even?
- **Majority:** Are there more 1-bits than 0-bits in x?

Parity can easily be handled without memory: just add the bits in x modulo 2.

On the other hand, Majority seems to require an integer counter of unbounded size $\log n$ bits; we will see in a while that Majority indeed cannot be solved in zero space.

```
s = 0;

while( there is another input bit b )
       s = b xor s;

return s;
```

This really computes the exclusive-or of all the bits, which happens to be the right answer:

$$s = x_1 \oplus x_2 \oplus \ldots \oplus x_{n-1} \oplus x_n$$

So this is an extremely simple case of a streaming algorithm, then number of scans is just 1 and the memory is constant (as opposed to small number of scans, little memory).

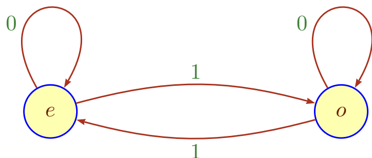
```
initialize;

while( there is another input letter x )
       process x;                  // state transition

return answer;
```

The point is that the state transition is extremely fast, typically using a lookup table, or evaluating a simple function.

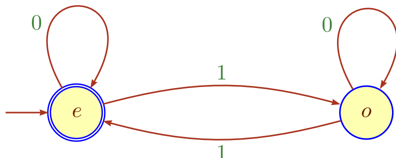
A most useful representation for our parity checker is a diagram:



The edges are labeled by the input bits, and the nodes indicate the internal state of the checker (called e and o for clarity, these are the two internal states).

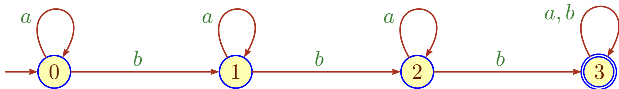
This pictures are very easy to read and interpret for humans (and useless as input to algorithms).

It is customary to indicate the initial state (where all computations start) by a sourceless arrow, and the so-called final states states (corresponding to answer Yes) by marking the nodes.



In this case state e is both initial and final.

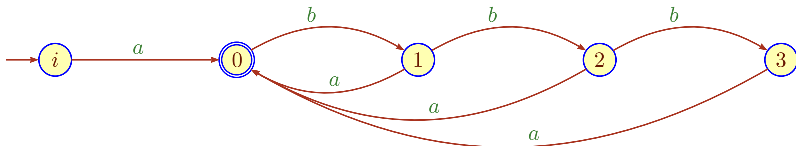
"Final state" is another example of bad terminology, something like "accepting state" would be better. Alas ...



There are 4 states $\{0, 1, 2, 3\}$. Input $x \in \{a, b\}^\star$ will take us from state 0 to state 3 if, and only if, it contains at least 3 letters b.

The "correctness proof" here consists of staring at the picture for a moment.

Consider all words over $\{a, b\}$ that start and end with a and have the property that all as are separated by 1, 2 or 3 bs.



We allow missing transitions: if the machine reads b in state i it simply "crashes" (see the formal definition of acceptance below). As a practical matter, partial transition functions are critical for efficiency.

Correctness is by diagram chasing. Note that the informal description above does not explain whether the first and last a need to be distinct. Deal with the other case.

A typical primality testing algorithm starts very modestly by making sure that the given candidate number x is not divisible by small primes, say, 2, 3, 5, 7, and 11 (actually, checking the first 100 or so primes seems to be more realistic in practice).

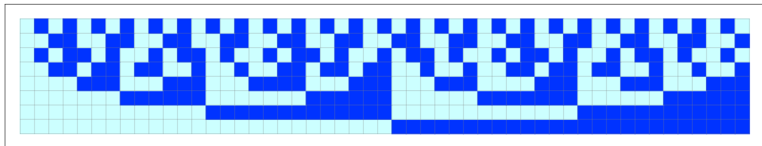
Assume n has 1000 bits. Using standard large integer library to do the tests is not really a good idea, we want a very fast method to eliminate lots of bad candidates quickly.

One could hardwire the division algorithm for a small divisor d but even that's still clumsy.

Can we use one of our memoryless machines?

8-bit binary numbers that are divisible by 5 (written here in columns, LSD on top).

There is some regularity in the bit patterns, but it's elusive.



We need a machine that accepts these bit pattern, but rejects all others. And, of course, works for an arbitrary number of bits.

Write $\nu(x)$ for the numerical value of bit-sequence x, assuming the MSD is read first.

Then

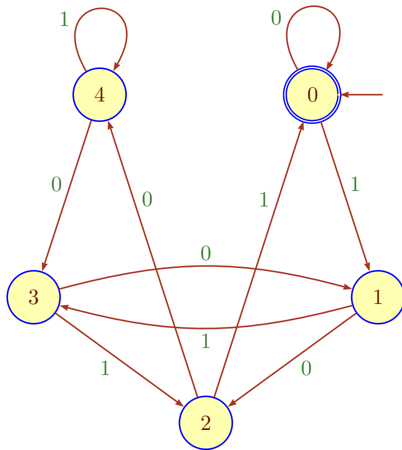
$$\nu(x0) = 2 \cdot \nu(x)$$
$$\nu(x1) = 2 \cdot \nu(x) + 1$$

So if we are interested in divisibility by, say, $d = 5$ we have

$$\nu(xa) = 2 \cdot \nu(x) + a \pmod 5$$

Since we only need to keep track of remainders modulo 5 there are only 5 values, corresponding to 5 internal states of the loop body.

Remainders Mod 5



Lower bound arguments are often tricky, but this really is the fastest possible algorithm for divisibility by 5 as can be seen by an adversary argument.

Suppose there is an algorithm that takes less than n steps.

Then this algorithm cannot look at all the bits in the input, so it will not notice a single bit change in at least one particular place.

But that cannot possibly work, every single bit change in a binary number affects divisibility by 5:

$$x \pm 2^k \neq x \pmod 5$$

for any $k \geq 0$.

1 Zero Space

2 **Finite State Machines**

3 DFA Decision Problems

We can think of our devices as consisting of two parts:

- a transition system, and
- an acceptance condition.

The transition system includes the states and the alphabet and can be construed as a labeled digraph.

Definition

A transition system or semi-automaton (SA) is a structure

$$\langle Q, \Sigma, \delta \rangle$$

where Q and Σ are finite sets and $\delta \subseteq Q \times \Sigma \times Q$.

The elements of δ are transitions and often written suggestively as $p \xrightarrow{a} q$.

It is customary to refer to the input sequences as words or strings.

Given an alphabet Σ one writes $\Sigma^\star$ for the collection of all words over Σ, and Σ^+ for the collection of all non-empty words.

In practice, the alphabet is usually one of

- $\mathbf{2} = \{0, 1\}$ (2)
- $\{0, 1, \ldots, 9\}$ (10)
- $\{0, 1, \ldots, 9, A, \ldots, F\}$ (16)
- lowercase letters (26)
- ASCII (128 or 256)
- UTF-8 (1,112,064)

but it is better to keep the definition general. Very large alphabets cause interesting algorithmic problems.

Suppose $\mathcal{A}$ is some semi-automaton. Given a word $u = a_1 a_2 \dots a_m$ over the alphabet of $\mathcal{A}$ a run of the automaton on u is an alternating sequence of states and letters

$$p_0, a_1, p_1, a_2, p_2, \dots, p_{m-1}, a_m, p_m$$

such that $p_{i-1} \xrightarrow{a_i} p_i$ is a valid transition for all i. p_0 is the source of the run and p_m its target, and $m \geq 0$ its length. So a run is just a path in a labeled digraph.

Sometimes we will abuse notation and also refer to the corresponding sequence of states alone as a run:

$$p_0, p_1, \dots, p_{m-1}, p_m$$

Given a run

$$\pi = p_0, a_1, p_1, a_2, p_2, \dots, p_{m-1}, a_m, p_m$$

of an automaton, the corresponding sequence of labels

$$a_1 a_2 \dots a_{m-1} a_m \in \Sigma^\star$$

is referred to as the trace or label of the run.

Every run has exactly one associated trace, but the same trace may have several runs, even if we fix the source and target states (ambiguous automata).

So, a transition system is just an edge-labeled digraph where the labels are chosen from some alphabet.

In the spirit of Rabin/Scott's 1959 paper, it is perfectly acceptable to have nondeterministic transitions

$$p \xrightarrow{a} q \quad \text{and} \quad p \xrightarrow{a} q' \quad \text{where} \quad q \neq q'$$

Note that these transitions are somewhat problematic from a "real algorithm" perspective: are we supposed to go to q or to q'?

This idea may sound quaint today, but it was a huge conceptual breakthrough at the time. Ponder deeply.

Definition

A semi-automaton is complete if for all $p \in Q$ and $a \in \Sigma$ there is some $q \in Q$ such that

$$p \xrightarrow{a} q$$

is a transition.

In other words, the system cannot get stuck in any state, we always can consume all input symbols.

Definition

A semi-automaton is deterministic if for all $p, q, q' \in Q$ and $a \in \Sigma$

$$p \xrightarrow{a} q, p \xrightarrow{a} q' \quad \text{implies} \quad q = q'$$

Thus, a deterministic system can have at most one run from a given state for any input.

Definition

A finite state machine or finite automaton is a structure

$$\mathcal{A} = \langle T; \mathsf{acc} \rangle$$

where $T = \langle Q, \Sigma, \delta \rangle$ is a transition system and acc is an acceptance condition.

We will make no attempt to define the concept of an acceptance condition in general and simply explain various examples as we go along.

The most basic kind of acceptance condition is comprised of a collection of initial states $I \subseteq Q$ and a collection of final states $F \subseteq Q$. The idea is that $\mathcal{A}$ accepts some input $x \in \Sigma^\star$ if there is a run from a state in I to a state in F with label x.

The (acceptance) language $\mathcal{L}(\mathcal{A})$ of the automaton $\mathcal{A}$ is the set of all words accepted by the automaton.

The acceptance condition depends much on the automaton in question but it is always a condition on the runs associated with a word u.

A run is accepting if it starts in I and ends in F.

In a moment we will consider the special case $I = \{q_0\}$ (think about resetting the automaton to the unique initial state q_0). In general, though, multiple final states cannot be avoided.

We can also think in terms of configurations, snapshots that contains all the information needed to resume the computation later. In this case, we only need to keep track of the current state $p \in Q$ and the remainder $z \in \Sigma^\star$ of the input.

$$p\,z \qquad p \in Q, z \in \Sigma^\star$$

One step in a computation is then given by δ (just a lookup table), the so-called transition function.

$$p\,az \mathrel{\Big|\!\frac{1}{\mathcal{A}}} q\,z \iff \delta(p, a, q)$$

Here $p \in Q$, $a \in \Sigma$, $z \in \Sigma^\star$.

The computation on input x ends after exactly $|x|$ steps in some state q without any input left. We accept if that state is final:

$$p\,x \;\vdash_{\mathcal{A}}\; q \qquad p \in I, q \in F$$

There is no need for a special halting state, we can simply read off the "response" of the machine by inspecting the last state.

In other words, we solve the decision problem over $\Sigma^\star$ with yes-instances $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\star$.

Useful States

Only those states in a finite state machine are relevant that lie on a path from I to F. A state is called accessible if it is reachable from I, and coaccessible if F is reachable from it.

One uses the same terminology for the whole automaton. In particular, an automaton is trim if it is both accessible and coaccessible.

A state p is a trap if all transitions with source p also have target p. A state is a sink if it is a trap and is not final.

Combining the previous acceptance condition with completeness and determinism produces a particularly useful type of automaton.

Definition

A deterministic finite automaton (DFA) is a structure

$$\mathcal{A} = \langle Q, \Sigma, \delta; q_0, F \rangle$$

where $\langle Q, \Sigma, \delta \rangle$ is a deterministic and complete semi-automaton and $q_0 \in Q$, $F \subseteq Q$ is the standard acceptance condition.

It is straightforward to see that a DFA has exactly one trace (or run) on any possible input word.

Since we use the standard acceptance condition, a run is accepting if it leads from q_0 to some $q \in F$ (a slight asymmetry).

Claim: One can safely assume that all states in a DFA are accessible, but coaccessibility may fail.

Arguably, DFAs should be called complete, deterministic finite automata, acronym CDFA. No one does this.

Alas, some authors allow incomplete deterministic machines under the name DFA to accommodate the removal of states that are not coaccessible, such as sinks.

We will always refer to these devices as partial DFAs (PDFAs) or incomplete DFAs.

It is often convenient to think of the transition function as a map $\delta : Q \times \Sigma^\star \to Q$ defined by primitive recursion over words:

$$\begin{aligned} \delta(p, \varepsilon) &= p \\ \delta(p, xa) &= \delta(\delta(p, x), a) \end{aligned}$$

In terms of the extended transition function, acceptance can be expressed easily:

$\mathcal{A}$ accepts a word u iff $\delta(q_0, u) \in F$.

Note that for all words x and y:

$$\delta(p, xy) = \delta(\delta(p, x), y)$$

Definition

A language $L \subseteq \Sigma^\star$ is recognizable or regular* if there is a DFA M that accepts L: $\mathcal{L}(M) = L$.

Thus a recognizable language has a simple, finite description in terms of a particular type of finite state machine. As we will see, one can manipulate the languages in many ways by manipulating the corresponding machines.

In a sense, recognizable languages are the simplest kind of languages that are of interest (there are more complicated types of languages such as context-free languages that are critical for computer science).

*Regular is more popular in the US, but hopelessly overloaded.

Note that we are using a slightly strange approach here: usually one first defines a class of functions (RM computable, primitive recursive, polynomial time computable, ...).

Then one introduces the corresponding class of relations via characteristic functions. This time we have no functions, only languages (think of them as unary relations on $\Sigma^\star$).

There is a class of finite state machines that compute functions, so-called transducers, more later.

The diagram perspective is useful to show that the Majority language $M = \{ x \in \mathbf{2}^\star \mid \#_0 x < \#_1 x \}$ is not recognizable.

For assume otherwise and let n be the number of states in a DFA accepting M. By definition, $0^n 1^{n+1}$ is accepted.

But then there is a path from q_0 to a final state q, labeled $0^n 1^{n+1}$.

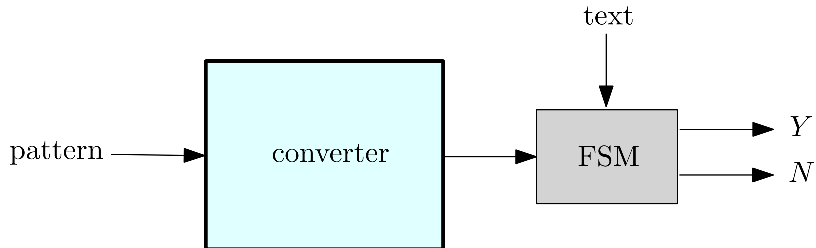
The first part must contain a loop that we can traverse multiple times, leading to an accepted input of the form $0^m 1^{n+1}$ where $m > n + 1$.

Contradiction.

This rather trivial observation is also known at the Pumping Lemma (for recognizable languages).

There are two somewhat separate reasons as to why finite state machines are hugely important.

1. Membership in a recognizable language can be tested blindingly fast, and using only sequential access to the letters of the word. This works very well with streams and is the foundation of many text searching and editing tools (such as `grep` and `emacs`). All compilers use similar tools.
2. Another important aspect is the close connection between finite state machines and logic. Here we don't care so much about acceptance of particular words but about the whole language. The truth of a formula can then be expressed as "some machine has non-empty acceptance language." Actually, this becomes really interesting for infinite words (where the first application disappears entirely).


text
pattern
converter
FSM
Y
N

Fast Acceptance Testing

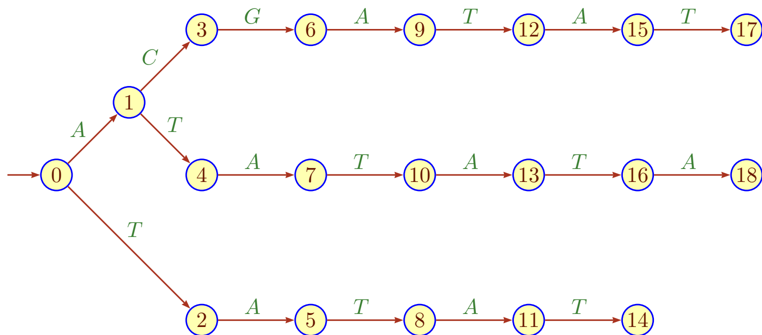
Proposition

For any DFA $\mathcal{A}$ and any input string x we can test in time linear in $|x|$ whether $\mathcal{A}$ accepts x, with very small constants.

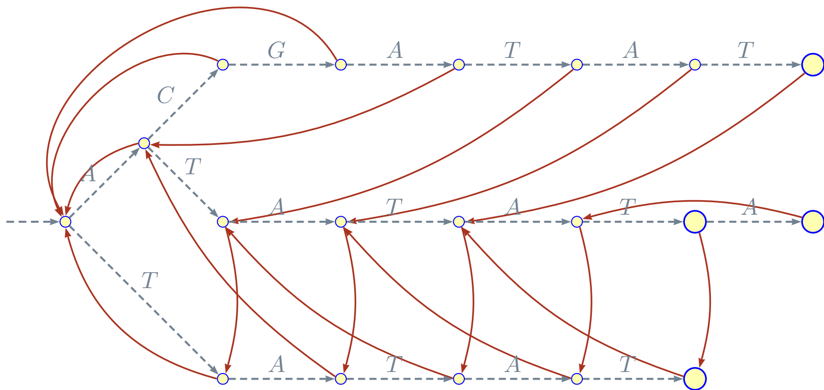
```
p = q0;                         // reset
while(  a = x.next() )          // next input symbol
      p = delta[p][a];          // table look-up

return  p in F;                 // table look-up
```

Of course, it might take some time to compute the lookup table δ in the first place, but once we have it acceptance testing is very fast.



This is the skeleton of a machine that searches for strings ACGATAT, ATATATA and TATAT.



If a mismatch occurs, take a back-transition and then try again. From here is not hard to construct a proper DFA.

We will shortly discuss closure properties of the languages associated with finite state machines. It will follow from these general results that a machine searching for words like ACGATAT, ATATATA and TATAT trivially exists.

The important point is that there are algorithms that construct the machines very efficiently, given the words as input. For example, the algorithm used in the last example is due to Aho and Corasick.

These algorithms can be quite sophisticated and clever; there is a whole field referred to as stringology that deals with them.

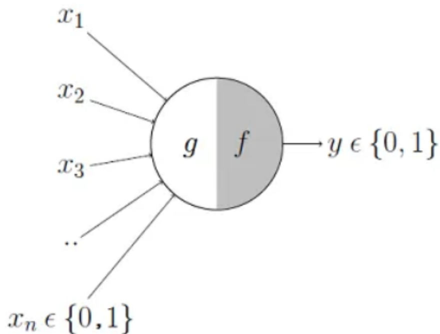
However, we will focus on the second killer app, finite state machines and logic.

W. S. McCulloch, W. Pitts
A logical calculus of the ideas immanent in nervous activity
Bull. Math. Biophysics 5 (1943) 115–133

S. C. Kleene
Representation of events in nerve nets and finite automata
in *Automata Studies* (C. Shannon and J. McCarthy, eds.)
Princeton UP, 1956, 3–41.

M. O. Rabin and D. Scott
Finite automata and their decision problems
IBM J. Research and Development, 3 (1959), 114–125.

McCulloch (neuroscientist) and Pitts (logician) present the first attempt to define the functionality of a neuron abstractly. The current AI craze goes back to this paper.



The references in the paper are rather remarkable.

R. Carnap, The Logical Syntax of Language
Harcourt, Brace and Company 1938.

D. Hilbert, W. Ackermann, Grundzüge der Theoretischen Logik
Springer Verlag 1927.

B. Russell, A. N. Whitehead, Principia Mathematica
Cambridge University Press 1925.

1 Zero Space

2 Finite State Machines

3 DFA Decision Problems

The 19559 paper by Rabin and Scott was an absolute breakthrough. For many years it was the most highly cited paper in CS. In particular, it introduced two major ideas:

- nondeterminism in machines,
- decision problems as a tool to study FSMs

Prior to the paper, computations on machines were always deterministic (even though nondeterminism pops up in the λ-calculus and in Chomsky grammars).

Given any language one is faced with a natural decision problem: determine whether some word belongs to the language. In this particular case the language is represented by a DFA.

Problem:	**DFA Membership (DFA Recognition)**
Instance:	A DFA $\mathcal{A}$ and a word x.
Question:	Does $\mathcal{A}$ accept input x?

Lemma

The DFA Membership Problem is solvable in linear time.

As we will see, there are other representations for recognizable languages where the membership problem is more difficult to solve. This is of great practical importance; many pattern matching problems can be phrased as membership in recognizable languages but using descriptions that are more difficult to deal with than DFAs.

Apart from membership testing there are several more complicated decision problems associated with finite state machines that have efficient solutions as long as the machine is a DFA. Again, these are crucial in many applications.

Problem:	**Emptiness**
Instance:	A DFA $\mathcal{A}$.
Question:	Does $\mathcal{A}$ accept no input?

Problem:	**Finiteness**
Instance:	A DFA $\mathcal{A}$.
Question:	Does $\mathcal{A}$ accept only finitely many inputs?

Problem:	**Universality**
Instance:	A DFA $\mathcal{A}$.
Question:	Does $\mathcal{A}$ accept all inputs?

Theorem

The Emptiness, Finiteness and Universality problem for DFAs are decidable in linear time.

Proof.

Consider the unlabeled diagram G of the machine. Emptiness means that there is no path in G from q_0 to any state in F, a property that can be tested by standard linear time graph algorithms (such as DFS or BFS). □

Exercise

Show in detail how to deal with Finiteness and Universality.

A general problem related to computation that we have encountered previously but not really pursued is program size complexity:

> What is the (size of the) smallest program for a given task?

Note that this is somewhat orthogonal to the usual time and space complexity of an algorithm: here the issue is the size of the code, not it's efficiency. Can you program a SAT solver on your wrist watch?

In general, identifying smallest programs is very hard. In particular for Turing/register machines the problem is highly undecidable.

But for DFAs there is a very good solution.

It is easy to see that the same language can be recognized by many different machines.

Definition

Two DFAs $\mathcal{A}_1$ and $\mathcal{A}_2$ over the same alphabet are equivalent if they accept the same language: $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$.

Given a few equivalent machines, we are naturally interested in the smallest one. In some sense, the smallest machine is the best representation of the corresponding recognizable language.

Definition

The state complexity of a DFA is the number of its states.
The state complexity of a recognizable language L is the size of a smallest DFA accepting L.

Naturally there is a computational problem lurking in the dark:

Problem:	**State Complexity**
Instance:	A recognizable language L.
Solution:	The state complexity of L.

The input L is given by a DFA. We will show in a moment that state complexity is computable, but efficient solutions require more work.

Note that we could ask similar questions for register and Turing machines (Kolmogorov-Chaitin complexity). Alas, in this general setting everything becomes highly undecidable. For example, one cannot determine the smallest machine that generates a given target output and then halts.

How about primitive recursive functions?

The obvious algorithm for state complexity is to

- generate all smaller DFAs, and
- check them for equivalence.

There are exponentially many smaller machines, but at this point we are not concerned with efficiency.

Lemma

Equivalence testing of DFA is decidable.

Proof.

It suffices to find some bound β so that $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent iff they agree on all words of length β.

We may safely assume that both machines have n states. Consider some word x of length m that the machines disagree on.

We trace the computations of both machines on x. Write $x_{\leq i}$ for the prefix of x of length i.

$$S_i = \big(\delta_1(q_{01}, x_{\leq i}), \delta_2(q_{02}, x_{\leq i})\big)$$

If $m \geq n^2$, then for some $0 \leq i < j \leq m$ we have $S_i = S_j$. But then we can shorten x by removing the factor $x_{i+1}, \ldots, x_j$.

Repeting this sort of surgery ultimately produces a string of length at $\beta = n^2$ works.

□

Checking all strings of length n^2 is wildly exponential, but with effort one can get the running time of Equivalence testing down to almost linear.

Note that the state complexity of a recognizable language always exists, albeit for a silly reason: the natural numbers are well-ordered.

However, there are two potential problems that could make a smallest machine somewhat useless.

- There might be several DFAs of minimal size.
- Even if there is only one (up to isomorphism), larger DFAs for the same language might have no reasonable connection to the minimal one.

The first problem would make it difficult to compare languages on the basis of their smallest machines.

The second problem could make it difficult to obtain a smallest machine given an arbitrary one.

We will see that for DFAs neither problem occurs.

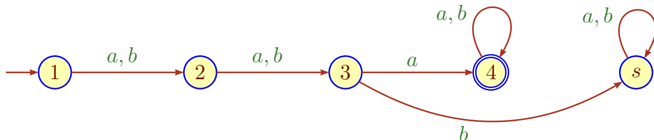
Consider the recognizable languages

$$L_{a,k} = \{\, x \in \{a,b\}^\star \mid x_k = a \,\}.$$

Thus $x \in L_{a,k}$ iff the kth symbol in x is an a.

What is the state complexity of these languages?

For positive k this is not a problem: we can just skip over the first $k-1$ symbols and then verify that x_k really is a. State complexity is $k+2$.



But what if k is negative and we are looking for the $|k|$th symbol from the end? E.g,

$$L_{a,-3} = \{aaa, aab, aba, abb, aaaa, aaab, aaba, aabb, \ldots\}$$

The crucial problem here is that the DFA does not know ahead of time when the last input will appear. And we cannot just go backwards from the end (our machines are one-way).

This may seem like a preposterous restriction, but streams do behave just like this; we don't know when the last input will come along.

Exercise

Figure out the state complexity of $L_{a,k}$ for negative k. No strict lower bound is required at this point, just come up with a machine that feels best possible.

Here is a much harder problem that deals with standard radix representations of integers.

Write $\nu_B(x)$ or simply $\nu(x)$ for the numerical value of string x written in base B, so $x \in \{0, 1, \ldots, B-1\}^\star$.

Also, one has to be a bit careful about the MSD and LSD. Unless otherwise noted, we assume that the MSD is the first digit, so

$$\nu(x_k x_{k-1} \ldots x_1 x_0) = \sum_{i \leq k} x_i \, B^i.$$

If the LSD is first we have a reverse radix representation.

We already know that divisibility by a fixed number m can be tested by a DFA with respect to base $B = 2$. But there are many other, useful numeration systems and it is not entirely clear whether one can build DFAs for all of them.

Lemma

Divisibility by m can be tested by a DFA in any base B.

Proof.

We can construct a canonical Horner automaton for this task.

Keep the state set $Q = \{0, 1, \ldots, m-1\}$.

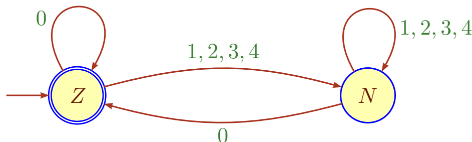
Change the transition function to

$$\delta(p, a) = p \cdot B + a \pmod{m}.$$

Initial state and only final state is 0.

Since $\delta(q_0, x) = \nu(x) \pmod{m}$ this works.

By the lemma, the state complexity for m and B is at most m. But that bound is not tight in general. For example, to check whether a number written in base 5 is divisible by 5, the canonical solution looks like this:



This suggests the question whether there is an easy way to describe the state complexity of our divisibility languages. Easy means: find some closed form, some nice function of m and B.

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	2	3	3	2	3	3	2	3	3	2	3	3	2	3
4	3	4	2	4	3	4	2	4	3	4	2	4	3	4	2
5	5	5	5	2	5	5	5	5	2	5	5	5	5	2	5
6	4	3	4	6	2	6	4	3	4	6	2	6	4	3	4
7	7	7	7	7	7	2	7	7	7	7	7	7	2	7	7
8	4	8	3	8	5	8	2	8	5	8	3	8	5	8	2
9	9	3	9	9	4	9	9	2	9	9	4	9	9	4	9
10	6	10	6	3	6	10	6	10	2	10	6	10	6	3	6
11	11	11	11	11	11	11	11	11	11	2	11	11	11	11	11
12	5	5	4	12	3	12	4	5	7	12	2	12	7	5	4
13	13	13	13	13	13	13	13	13	13	13	13	2	13	13	13
14	8	14	8	14	8	3	8	14	8	14	8	14	2	14	8
15	15	6	15	4	6	15	15	6	4	15	6	15	15	2	15
16	5	16	3	16	8	16	3	16	9	16	5	16	9	16	2

$m :\downarrow \quad B :\rightarrow$

The problem is to extract useful information from this table.

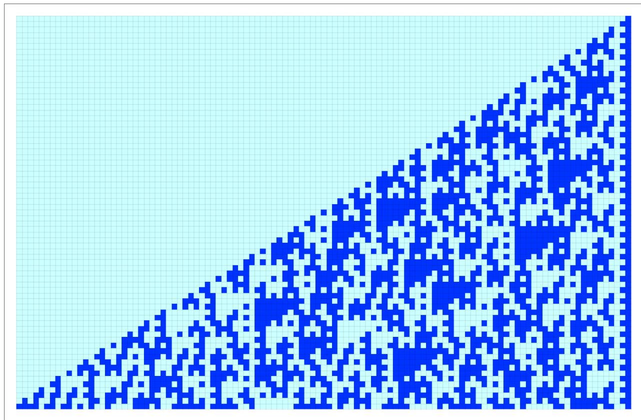
Unfortunately, there aren't too many patterns that are clearly visible.

- For m a prime things seem straightforward.
- Base $B = 2$ seems potentially doable (but not obvious).

It seems rather difficult to come up with a really good answer; incidentally, a perfect term project.

How about more complicated properties of numbers?

Suppose we want to recognize powers of 3 written base 2.



This looks rather complicated. In fact there is no DFA that could recognize these numbers (but the proof is quite hard, see Cobham's theorem).