Natural numbers and Turing machines

If forced to say what the natural numbers `are really,' the answer is: nonnegative decimal integers.

$$\mathbb{N} = \{0, 1, 2, \ldots, 9, 10, \ldots, 999, \ldots\}$$

Isomorphic `models' include binary numbers, strings of 1s, hexadecimal numbers, and so on.

Partial functions notation $f : X \nrightarrow Y$

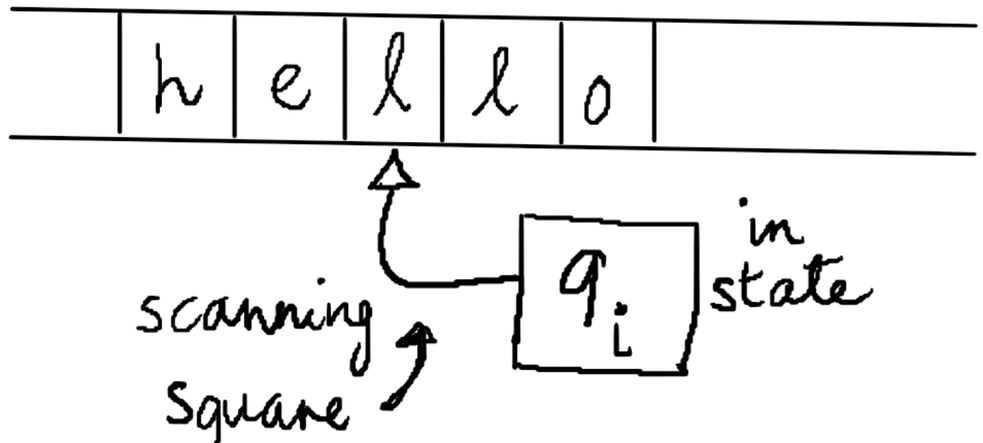
An **alphabet** is a finite set (any finite set)

Σ an alphabet. String over Σ is a finite sequence $a_1 a_2 \cdots a_k$, each $a_j \in \Sigma$ (a letter or symbol). If $k=0$ we have the empty string, $\boxed{\lambda}$

$$\Sigma^* = \{\text{strings over } \Sigma\}$$

catenation $(a_1 \cdots a_k)(b_1 \cdots b_\ell) = a_1 \cdots a_k b_1 \cdots b_\ell$

$$\mathbb{N} = \{0\} \cup \{1,\cdots,9\}\{0,\cdots,9\}^*$$



quintuple $q_i \; X \; Y \; \begin{Bmatrix} L \\ R \end{Bmatrix} \; q_j$

in state q_i
scanning X
overwrite Y
move head L or R
adopt state q_j

K: states

$K = \{q_0, \cdots, q_n\}$

Input alphabet Σ

tape alphabet $\Gamma \supseteq \Sigma$

blank $B \in \Gamma \setminus \Sigma$

initial q_0

quintuples δ.

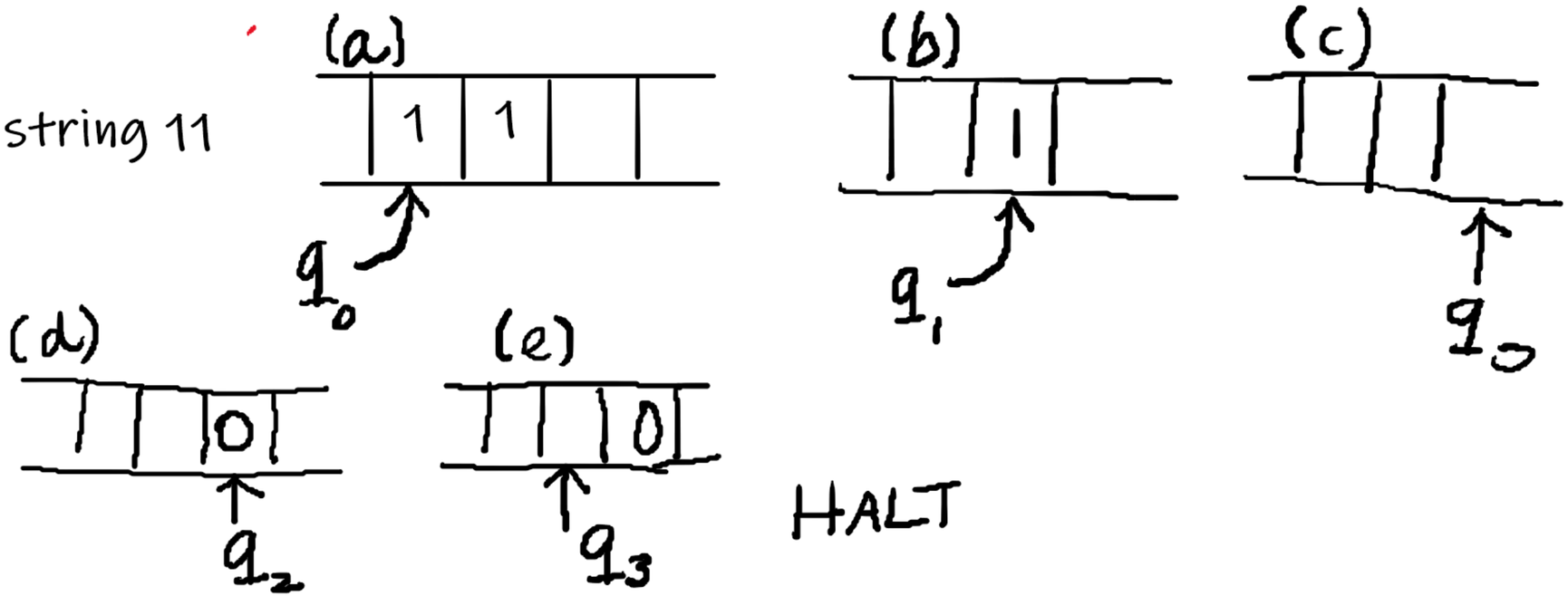
Example.

$\Sigma = \{1\}$ $\quad K = \{q_0, \cdots, q_3\}$ initial q_0

$\Gamma = \{1, B\}$ $\quad \delta$: $q_0 1 B R q_1$ $\quad q_1 1 B R q_0$

$q_0 B 0 R q_2$ $\quad q_1 B 1 R q_2$ $\quad q_2 B B L q_3$

EG input string 11



Logic and computation lecture 02

Given a string x, its length |x| is its length as a finite sequence of letters (symbols).

xy is the product of x and y, but the product under concatenation, not in an arithmetical sense. Similarly, x^n is the product of n copies of x concatenated together. 1^3 is the string 111

The Turing machine given as the first example has input alphabet {1}, so the input strings are 1^n, n >= 0. It is easy to see that on input 1^n, it halts with 0 on the tape if n is even and with 1 on the tape if n is odd: n mod 2.

Easy exercise. Design a Turing Machine with binary input alphabet ({0,1}) such that on input x it halts with |x| mod 2 on the tape.

Repeat. A Turing machine is an object specified by a tuple K, \Sigma, \Gamma, q_0, \delta where \delta is a set of quintuples defining a partial function

$K \times \Gamma \not\to \Gamma \times \{L,R\} \times K$

It passes through a sequence of `configurations.'

a configuration is defined by the current state q_i, the (nonblank) tape contents, and the position of the square being scanned (current square).

A Turing machine operates in a series of simple steps. Given x in \Sigma^* (input string), the initial configuration with input x is in the initial state q_0 with x stored (in adjacent squares) on the tape and the read/write head positioned at the square containing the leftmost letter in x, or blank if x is empty.

At a given step, let q_i be the state and X the symbol in the square being scanned.

If there is no quintuple q_i X ... the machine does nothing. It is in a halting configuration.

If there is a quintuple beginning q_i X ... then it is unique, q_i X Y L/R q_j, say.

(i) It replaces X by Y in the square being scanned

(ii) It moves to the next square on the left if L is specified, on the right if R is specified.

(iii) It adopts the state q_j

The semantics of Turing machines are (is?) very easy

The set K of states and the tape alphabet Γ must be disjoint sets --- not a problem.

A configuration is represented by a string in
\Gamma^* x K x \Gamma^*

The string \alpha q_i \beta is the configuration
in which \alpha\beta are the tape contents,
the rest of the tape being blank; the combined
string does not begin nor end with the blank B;
q_i is the state; and the square being scanned
contains the leftmost letter in \beta, or B if
\beta is empty. We can then define the relation

`Configuration A yields configuration B directly.'

With a simple case analysis. For example,

$q_i XYR q_j$ applies to any configuration $\alpha q_i X \gamma$

($\beta = X\gamma \neq \lambda$, say) : $\alpha q_i X \gamma \vdash \alpha Y q_j \gamma$

$\vdash$: yields directly

etcetera.

Given $x \in \Sigma^*$, the initial configuration on input x is $q_0 x$

$\{q_0, \ldots, q_3\}$ $\{1\}$ $\{1, B\}$ q_0

δ: $q_0 1 B R q_1$ $q_1 B R q_0$ $q_0 B 0 R q_2$ $q_1 B 1 R q_2$ $q_2 B B L q_3$

Computation

$q_0 1111 \vdash q_1 1111 \vdash q_0 111 \vdash q_1 11 \vdash q_0 1 \vdash q_1 \vdash 1 q_2 \vdash q_3 1$

DEFINITION a Turing machine M computes a partial function $f : \Sigma^* \to \Pi^*$

where \Sigma is the input alphabet and \Pi is a previously unmentioned `output alphabet,' not containing the blank symbol B, such that for every input string x, beginning with this input string, if M halts then it halts in a configuration qz where z \in \Pi^*. AND

If f(x) is defined, then M halts with (nonblank) tape contents f(x). AND

If f(x) is undefined, then M loops on input x.

Exercise. Design a Turing machine with binary input and output alphabets which is defined only on nonempty binary strings and computes |x| mod 2 on those inputs.

The halting problem. There is no software
which can tell if a program is looping or just slow.

If M is a turing machine and x is a (bitstring)
input then $M(x)\downarrow$ means M halts on input x

$M(x)\downarrow y$ M halts with output y

$M(x)\uparrow$ M loops on input x.

A set X of bitstrings is RECURSIVE if there
exists a Turing machine M such that
M(x) halts with output 1 if x \in X and
output 0 if x \notin X.

Notation $TM = \{$valid bitstring encodings of Turing machines$\}$

halting prob given machine M and input x,
does $M(x)\downarrow$?

$HALTING = \{$bitstrings $x : \exists\, y, z$

$x = yz \;\wedge\; y \in TM$ and $T_y(z)\downarrow \}$

T_y : the TM encoded by y.

Theorem The halting problem has no solution; HALTING is undecidable; HALTING is not recursive.

Proof Otherwise there would be a TM T such that T(x) always halts, with output 1 if x can be factored as yz where T_y(z) halts, and 0 otherwise.

One could construct a Turing machine M which does this on input x:

(i) it `doubles' the input, constructing xx on its tape.
(ii) it incorporates the quintuples of T to determine whether T(xx) = 0 or 1
(iii) If 0, M(x) halts.
(iv) If 1, M(x) loops.

So: M(x) halts if and only if xx \notin HALTING

Based on the assumption that T exists, M exists, and M = T_c for a suitable encoding c

If M(c) halts

$cc \notin HALTING$

for no factorisation yz of cc, $y \in TM$, $T_y(z)\downarrow$

$\therefore T_c(c)\uparrow$

$\therefore T_c \neq M$

If M(c) loops

$cc \in HALTING$

$\exists\, y, z \quad y \in TM \quad cc = yz \wedge T_y(z)\downarrow$

UNIQUE FACTORISATION

$y = z = c$

$T_c(c)\downarrow \qquad T_c \neq M$ QED

next HALTING is r.e.

Recursively enumerable sets. A set is recursive if its characteristic function can be computed by a Turing machine which halts on all inputs.

Definition: A set X (of bitstrings) is RECURSIVELY ENUMERABLE if there exists a TM M such that

x \in X iff M(x) halts

Over and above the notion of halting, there is also the notion of halting with output, and there is the notion of a Turing machine computing partial functions partially mapping bitstrings to bitstrings.

We are mostly concerned with partial functions
whose domain and range are sets of bitstrings.

Recall that $M(x)\downarrow y$ means that with input x, M halts with output y. DEFINITION: a UNIVERSAL TURING MACHINE U has the property that $U(x)\downarrow w$ if $\exists y \in TM,\ z \in \{0,1\}^*$ $x = yz$, and $T_y(z)\downarrow w$; $U(x)\uparrow$ if either $\neg\forall y,z$, $y \in TM \Rightarrow x \neq yz$ or $\exists y,z$ $y \in TM \wedge x = yz \wedge T_y(z)\uparrow$

Proposition: universal Turing machines exist. The nearest to a proof of this fact is an inspection of the program TuringinC.c, which accomplishes something similar to a Universal turing machine.

The domain of the Universal Turing Machine, the set of x on which U(x) halts, is the set of all strings of the form yz where y \in TM and $T_y(z)$ halts.

In other words the domain is the set HALTING. HALTING is recursively enumerable but not recursive.

There are many other example of non-recursiveness. For example, it is undecidable whether a semigroup has decidable word-problem.

Propositional logic. In general, logic is concerned with theories, proof systems, and interpretations. Propositional logic is the simplest form in which everything is composed of boolean variables and interpretations are connected with truth tables.

There are two TRUTH-values, true and false, IDENTIFIED with 1 and 0 respectively.

A TRUTH-FUNCTION is a map from {0,1}^n to {0,1} for some n.

A truth-table is just a presentation of a truth-function.

We begin with four logical connectives whose meaning is defined by truth tables.

They are $\neg$ not, $\wedge$ and, $\vee$ or, $\Rightarrow$ implies

$\neg$:

X	$\neg X$
0	1
1	0

$\wedge$:

X	Y	$X \wedge Y$
0	0	0
0	1	0
1	0	0
1	1	1

$\vee$:

X	Y	$X \vee Y$
0	0	0
0	1	1
1	0	1
1	1	1

$\Rightarrow$:

X	Y	$X \Rightarrow Y$
0	0	1
0	1	1
1	0	0
1	1	1

Philonian

Truth-functions, and, or, and not

X	Y	Z	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0

X	Y	Z	f
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

use $\overline{X}$ for $\neg X$

DNF $(\overline{X} \wedge \overline{Y} \wedge Z) \vee (X \wedge \overline{Y} \wedge \overline{Z}) \vee (X \wedge Y \wedge \overline{Z})$

CNF $(X \vee Y \vee Z) \wedge (X \vee \overline{Y} \vee Z) \wedge (X \vee \overline{Y} \vee \overline{Z}) \wedge$
$(\overline{X} \vee Y \vee Z) \wedge (\overline{X} \vee \overline{Y} \vee \overline{Z})$

Logic is concerned with formulae and interpretations. In propositional logic, an interpretation is just an assignment of truth-values to the boolean variables in the formulae. A formula is INCONSISTENT if it is false under every interpretations (i.e., the truth-table has 0 everywhere in the right-hand column).

A CNF (conjunctive normal form) is a formula like this (X_i are boolean variables, not necessarily distinct)

$$(\pm X_1 \vee \pm X_2 \vee \cdots \vee \pm X_k) \wedge (\pm X_{k+1} \vee \cdots \vee \pm X_\ell) \wedge \cdots \wedge (\cdots \pm X_m)$$

$$\{\{\pm X_1, \cdots, \pm X_k\}, \{\pm X_{k+1}, \cdots, \pm X_\ell\} \wedge \cdots \wedge \{\cdots \pm X_m\}\}$$

$$\left\{\frac{X_i}{X_i}\right\} : \pm X_i$$

Set of clauses ALSO $\{\} = \square$ empty

Resolution $C \vee L,\ D \vee \overline{L} \mapsto C \vee D$

$\{\{X\}, \{\overline{X}, Y\}, \{\overline{Y}\}\}$ or in brief $X,\ \overline{X}Y,\ \overline{Y}$

$X,\ \overline{X}Y \mapsto Y$

$Y,\ \overline{Y} \mapsto \square$

$\square$ is inconsistent contradictory

$\therefore\ X,\ \overline{X}Y,\ \overline{Y}$ is.

~~Matrix~~

X	Y	$\overline{X}Y$	$\overline{Y}$	$X \wedge \overline{X}Y \wedge \overline{Y}$	
0	0	1	1	0	inconsistent. ← FALSE under all truth-assignments.
0	1	1	0	0	
1	0	0	1	0	
1	1	1	0	0	

Evaluating a CNF under a truth-assignment (which assigns 0/1 to all boolean variables)

$C_1 \wedge \cdots \wedge C_N$ is true iff all clauses are true

$\{\ell \pm X_1, \cdots, \pm X_k\}$ is true iff at least one of the LITERALS is true. ℓ literal: X or $\overline{X}$. Convention $\overline{\overline{X}} = X$

X_i is true iff $X_i \mapsto 1$. $\overline{X_i}$ is true iff $X_i \mapsto 0$

NB $\square$ is FALSE

Notation $f \models A$ if A is true under f.

Lemma. if $f \models C \vee L$ and $f \models D \vee \overline{L}$ then $f \models C \vee D$

Proof. $f \models C \vee L \quad f \models D \vee \bar{L}$

if $f \models L$ then $f \models D$ and $f \models C \vee D$

$f \models \bar{L}$ $\quad f \models C$ $\quad f \models C \vee D$ QED

resolution derive $\square$ by repeated resolution.

$\therefore$ resolution REFUTATIONS are sound.

A more informative definition. A RESOLUTION REFUTATION of a set S of clauses is a list of clauses, each clause either belonging to S or a resolvent of two clauses occurring earlier in the list, so that the list contains (usually, ends with) the empty clause.

RESOLUTION IS SOUND If a set S of clauses admits a resolution refutation, then S is unsatisfiable.

PROOF If S is satisfiable, i.e., not inconsistent, then there is an interpretation which satisfies S. By induction, this interpretation satisfies every clause in the refutation, impossible since it cannot satisfy the empty clause.

Resolution is complete. This will show how set notation is suitable for CNFs. Let S be a set of clauses and L a literal occurring in S. Define

$$S \setminus L = \{C \setminus \{L\} : C \in S \wedge \overline{L} \notin C\}.$$

Lemma If $S \setminus L$ is satisfiable, then so is S

Proof Let f be a truth-assignment to the boolean variables in $S \setminus L$. L (or $\overline{L}$ if $L = \overline{X}$ for some X) is not one of the variables in $S \setminus L$. There may be other variables in S besides L (or $\overline{L}$). Say V are those extra. Extend f to S by setting V arbitrarily and making L false

Let F be this extension of f; $F(L) = 0$

If C is a clause in S not containing $\overline{L}$,
then $f \models C \vee L$ so $F \models C$ since the only
literal in C, if any, not covered by f, is L,
and $f \models C \vee L$.

If C is a clause in S and $\overline{L} \in C$ then $F(\overline{L}) = 1$
and $F \models C$. QED (If $S \wedge \overline{L}$ is satisfiable
then so is S)

Completeness of resolution

If S is an inconsistent set of clauses then the empty clause can be derived from S by resolution.

Assume S does not contain the empty clause --- otherwise trivial. Then use induction on the number of (distinct) boolean variables occurring in S.

The only unsatisfiable CNF with 1 B.V. is $\{\{X_1\}, \{\overline{X}_1\}\}$ and $X_1, \overline{X}_1 \longmapsto \square$ (resolvent)

Induction Choose any literal L in S.

Given S is inconsistent; so are $S \cup L$ and $S \cup \overline{L}$.

Consider a refutation of $S \cup L$, and a resolvent occurring: $C \vee X$, $D \vee \overline{X} \mapsto C \vee D$

Early in the refutation, $C \vee X$, $D \vee \overline{X}$ both come from $S \cup L$ and come from clauses

$C \vee X \,[\vee L]$ (possibly containing L)
and $D \vee \overline{X} \,[\vee L]$ in S

Then the resolvent is $C \vee D \,[\vee L]$

By an inductive argument, the refutation of $S \cup L$ leads to a resolution derivation of $\square$ or L from S.