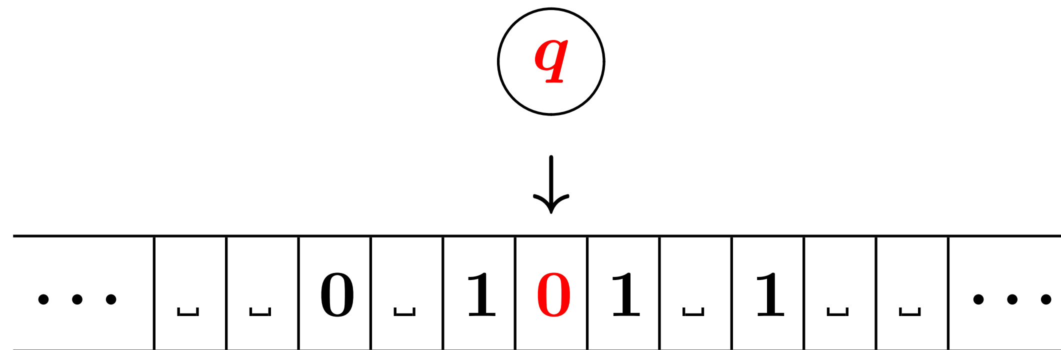
Algorithms, informally

No precise definition of “algorithm” at the time Hilbert posed the Entscheidungsproblem, just examples.

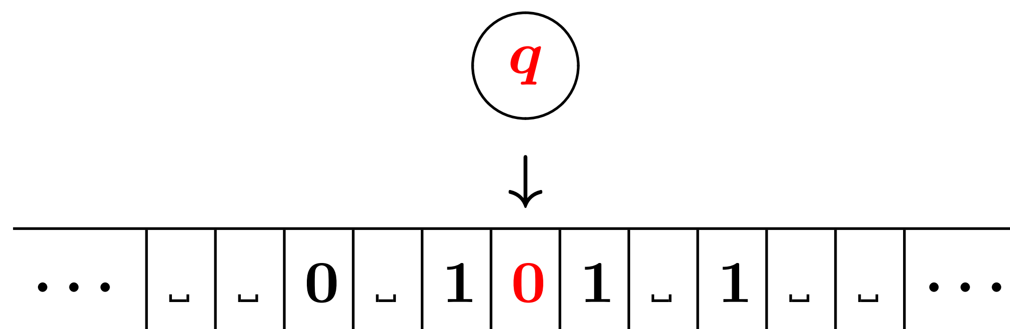
Common features of the examples of algorithms:

- finite description of the procedure in terms of elementary operations;
- deterministic, next step is uniquely determined if there is one;
- procedure may not terminate on some input data, but we can recognise when it does terminate and what the result is.

Turing machines, informally



Turing machines, informally



- The machine starts in state s with the tape head pointing to the first symbol of the finite input string. (Everything to the left and right of the input string is initially blank.)
- The machine computes in steps, each depending on the current state (q) and symbol being scanned by tape head ($\mathbf{0}$)
- An action at each step is to: overwrite the current tape cell with a symbol; move left or right one cell; and change state.

Turing Machine, formally

A **Turing machine** is specified by a quadruple $M = (Q, \Sigma, s, \delta)$ where

- Q is a finite set of machine states;
- Σ is a finite set of tape symbols, containing distinguished symbol $\sqcup$, called blank;
- an initial state $s \in Q$;
- a partial transition function

$$\delta \in (Q \times \Sigma) \rightharpoonup (Q \times \Sigma \times \{L, R\})$$

Turing Machine Configuration

A Turing Machine configuration (q, w, u) consists of

- the current state $q \in Q$;
- a finite, possibly-empty string $w \in \Sigma^*$ of tape symbols to the left of tape head;
- a finite, possibly empty string $u \in \Sigma^*$ of tape symbols under and to the right of tape head. ϵ denotes the empty string.

An initial configuration is (s, ϵ, u), for initial state s and string of tape symbols u.

The configuration only describes the contents of tape cells that are part of the input or have been visited by the Turing machine. Everything else is blank.

first and last

Define the functions $\textbf{first} : \Sigma^* \to \Sigma \times \Sigma^*$ and $\textbf{last} : \Sigma^* \to \Sigma \times \Sigma^*$ as follows

$$\textbf{first}(w) = \begin{cases} (a, v) & \text{if } w = av \\ (\text{␣}, \epsilon) & \text{if } w = \epsilon \end{cases}$$

$$\textbf{last}(w) = \begin{cases} (a, v) & \text{if } w = va \\ (\text{␣}, \epsilon) & \text{if } w = \epsilon \end{cases}$$

These functions split off the first and last symbols of a string, splitting off ␣ if the string is empty.

Turing Machine Computation

Given $M = (Q, \Sigma, s, \delta)$, define $(q, w, u) \rightarrow_M (q', w', u')$ by

$$\frac{\mathsf{first}(u) = (a, u') \qquad \delta(q, a) = (q', a', L) \qquad \mathsf{last}(w) = (b, w')}{(q, w, u) \rightarrow_M (q', w', ba'u')}$$

$$\frac{\mathsf{first}(u) = (a, u') \qquad \delta(q, a) = (q', a', R)}{(q, w, u) \rightarrow_M (q', wa', u')}$$

We say that a configuration (q, w, u) is a normal form if it has no computation step. This is the case exactly when $\delta(q, a)$ is undefined for $\mathsf{first}(u) = (a, u')$.

Turing Machine Computation

A computation of a TM M is a (finite or infinite) sequence of configurations $c_0, c_1, c_2, \ldots$ where

- $c_0 = (s, \epsilon, u)$ is an initial configuration
- $c_i \rightarrow_M c_{i+1}$ holds for each $i = 0, 1, \ldots$.

The computation

- does not halt if the sequence is infinite
- halts if the sequence is finite and its last element (q, w, u) is a normal form.

Example Turing Machine

Consider the TM $M = (Q, \Sigma, s, \delta)$ where $Q = \{s, q, q'\}$, $\Sigma = \{\sqcup, 0, 1\}$ and the transition function $\delta \in (Q \times \Sigma) \rightharpoonup (Q \times \Sigma \times \{L, R\})$ is given by:

δ	$\sqcup$	0	1
s	$(q, \sqcup, R)$		
q	$(q', 0, L)$	$(q, 1, R)$	$(q, 1, R)$
q'			$(q', 1, L)$

Theorem. The computation of a Turing machine M can be implemented by a register machine.

Proof (sketch).

Step 1: fix a numerical encoding of M's states, tape symbols, tape contents and configurations.

Step 2: implement M's transition function (finite table) using RM instructions on codes.

Step 3: implement a RM program to repeatedly carry out $\rightarrow_M$.

Tape encoding of lists of numbers

Definition. A tape over $\Sigma = \{\text{␣}, 0, 1\}$ **codes a list of numbers** if precisely two cells contain $\mathbf{0}$ and the only cells containing $\mathbf{1}$ occur between these.

Such tapes look like:

$$\underbrace{\cdots \text{␣␣}}_{\text{all ␣'s}} \mathbf{0} \underbrace{\mathbf{1}\cdots\mathbf{1}}_{n_1} \text{␣} \underbrace{\mathbf{1}\cdots\mathbf{1}}_{n_2} \text{␣} \underset{\cdots}{\cdots} \text{␣} \underbrace{\mathbf{1}\cdots\mathbf{1}}_{n_k} \mathbf{0} \underbrace{\text{␣␣}\cdots}_{\text{all ␣'s}}$$

which corresponds to the list $[n_1, n_2, \ldots, n_k]$.

Note the blank spaces: ␣'!

Turing computable function

Definition. $f \in \mathbb{N}^n \rightharpoonup \mathbb{N}$ is **Turing computable** if and only if there is a Turing machine M with the following property:

> Starting M from its initial state with tape head on the leftmost 0 of a tape coding $[x_1, \ldots, x_n]$, M halts if and only if $f(x_1, \ldots, x_n)\downarrow$, and in that case the final tape codes a list (of length ≥ 1) whose first element is y where $f(x_1, \ldots, x_n) = y$.

Theorem. A partial function is Turing computable if and only if it is register machine computable.

Proof (sketch). We've seen how to implement any TM by a RM. Hence

f TM computable implies f RM computable.

For the converse, one has to implement the computation of a RM in terms of a TM operating on a tape coding RM configurations. To do this, one has to show how to carry out the action of each type of RM instruction on the tape. It should be reasonably clear that this is possible in principle, even if the details are omitted (because they are tedious).

Notions of computability

- Church (1936): **λ-calculus**
- Turing (1936): **Turing machines**.

Turing showed that the two very different approaches determine the same class of computable functions. Hence:

Church-Turing Thesis. Every algorithm [in intuitive sense] can be realized as a Turing machine.

Models of computability

Church-Turing Thesis. Every algorithm can be realized as a Turing machine. Further evidence:

- Gödel and Kleene (1936): **partial recursive functions**
- Church (1936): **λ-calculus**
- Post (1943) and Markov (1951): canonical systems for generating the theorems of a formal system
- Lambek (1961) and Minsky (1961): **register machines**
- Variations on all of the above (e.g. multiple tapes, non-determinism, parallel execution. . .)

All determine the same collection of computable functions.