Slide 1

Algorithms, informally

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

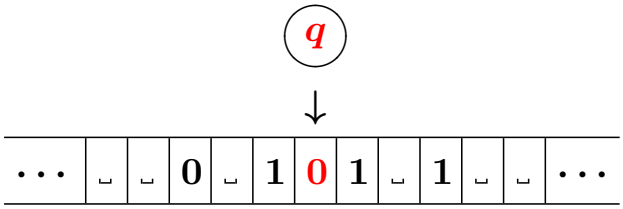
Common features of the examples of algorithms:

- 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.

Register Machine computation works with natural numbers and the associated elementary operations of increment/decrement/zero-test. It abstracts away from any particular, concrete representation of numbers (e.g. as bit strings). Turing’s original model of computation (now called a Turing machine) is more concrete. Numbers are represented in terms of a fixed, finite alphabet of symbols and the increment/decrement/zero-test programmed in terms of more elementary symbol-manipulating operations. In fact, Turing argued that he had formalized the notion of ”algorithm” in the most concrete possible form.

Slide 2

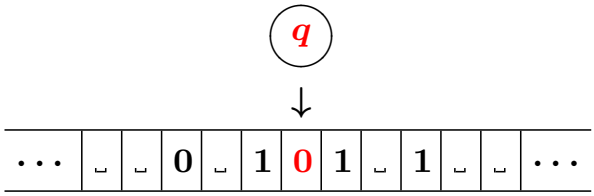
Turing machines, informally



A Turing machine consists of a linear tape unbounded to the left and right, which is divided into cells. Each cell contains either a symbol from a finite alphabet of *tape symbols*, in this case $\mathbf{0}$ and $\mathbf{1}$, or the special blank symbol ␣. Only finitely many cells may contain non-blank symbols. In slide 3, the Turing machine is in state q with its tape head (the arrow) scanning the tape cell with tape symbol $\mathbf{0}$.

Slide 3

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 (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.

There are many variations of the definition of a Turing machine. The definition of a Turing machine in this course consists of a two-way-infinite tape which starts at the left of the input string. It is also possible to define Turing machines where the tape is infinite in only one direction, or that can leave the tape head stationary as well as moving left or right. These variants all have the same computational power.

Slide 4

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

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

The machine has finite internal memory. It can remember which state it is in, nothing more. At each step of the computation, the machine reads the symbol currently under the tape head. If the machine is currently in state q and reading symbol a then $\delta(q, a)$ tells the machine what to do next. If $\delta(q, a)$ is undefined, the machine simply halts. Otherwise, $\delta(q, a) = (q', a', d)$ for some state q', symbol a' and direction d. The machine overwrites the current cell with the symbol a', moves along the tape in direction d (L for left, R for right), and changes its state to q'.

Now we need to describe formally what it means to compute with a Turing machine. In an analogous fashion to register-machine configurations, we define the notion of Turing-machine configurations.

Slide 5

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.

Slide 6

first and last

Define the functions **first** : $\Sigma^* \to \Sigma \times \Sigma^*$ and **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.

Slide 7

Turing Machine Computation

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

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

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

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 $\textbf{first}(u) = (a, u')$.

Slide 8

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
- $c_i \to_M c_{i+1}$ 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.

Slide 9

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'			$(q', 1, L)$

Slide 10

Graphical Representation of TMs

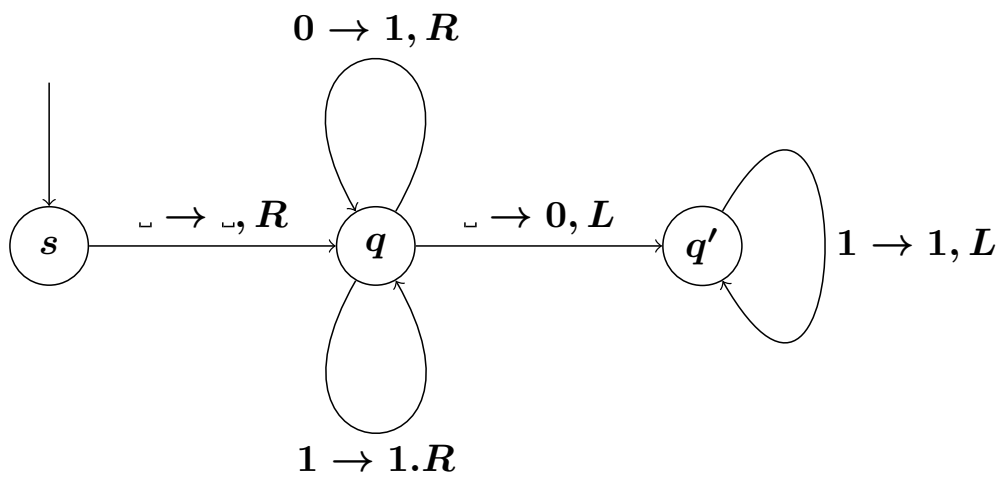
Construct a graph representing a TM:

- Nodes: states $q \in Q$,
- Edges: representing the transition function:

  For $((q, s), (q', s', M)) \in \delta$ with $q, q' \in Q$, $s, s' \in \Sigma$ and $M \in \{L, R\}$ there is a link from q to q' labelled by $s \to s', M$, and
- Initial state: $s \in Q$ indicated e.g. by an (unlabelled) edge.

Slide 11

Example TM Diagram



Slide 12

Example Computation

$$
\begin{aligned}
(s\,,\,\epsilon\,,\,\sqcup 1^n 0) &\to_M (q\,,\,\sqcup\,,\,1^n 0)\\
&\to_M (q\,,\,\sqcup 1\,,\,1^{n-1}0)\\
&\;\;\vdots\\
&\to_M (q\,,\,\sqcup 1^n\,,\,0)\\
&\to_M (q\,,\,\sqcup 1^{n+1}\,,\,\varepsilon)\\
&\to_M (q'\,,\,\sqcup 1^n\,,\,10)\\
&\to_M (q'\,,\,\sqcup 1^{n-1}\,,\,110)\\
&\;\;\vdots\\
&\to_M (q'\,,\,\sqcup\,,\,1^{n+1}0)
\end{aligned}
$$

Slide 13

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 $\rightarrow_M$.

Slide 14

Step 1

- Identify states and tape symbols with numbers:

$$Q = \{0, 1, \ldots, n\} \qquad \Sigma = \{0, 1, \ldots, m\}$$

  where $s = 0$ and $\text{␣} = 0$

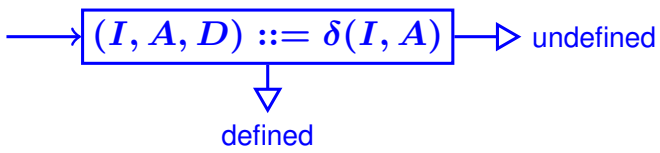
- Code configurations $c = (q, w, u)$ with three numbers:
  - q, the state number
  - $\ulcorner [a_i, \ldots, a_1] \urcorner$ where $w = a_1 \cdots a_i$
  - $\ulcorner [b_1, \ldots, b_j] \urcorner$ where $u = b_1 \cdots b_j$.

  [The reversal of w makes it easier to use our RM programs for list manipulation.]
- Identify directions with numbers: $L = 0$, $R = 1$

Slide 15

Step 2

Turn the finite table of (argument,result)-pairs specifying δ into a RM gadget



which has the behaviour:

If q, a and d are the initial values of registers I, A and D

- updates the registers to $I = q', A = a', D = d'$ and takes the defined exit if $\delta(q, a) = (q', a', d')$
- leaves the registers intact and takes the undefined exit if $\delta(q, a)$ is undefined

Slide 16

Step 3

The next slide gives a RM which implements the computation of TM M. It uses registers

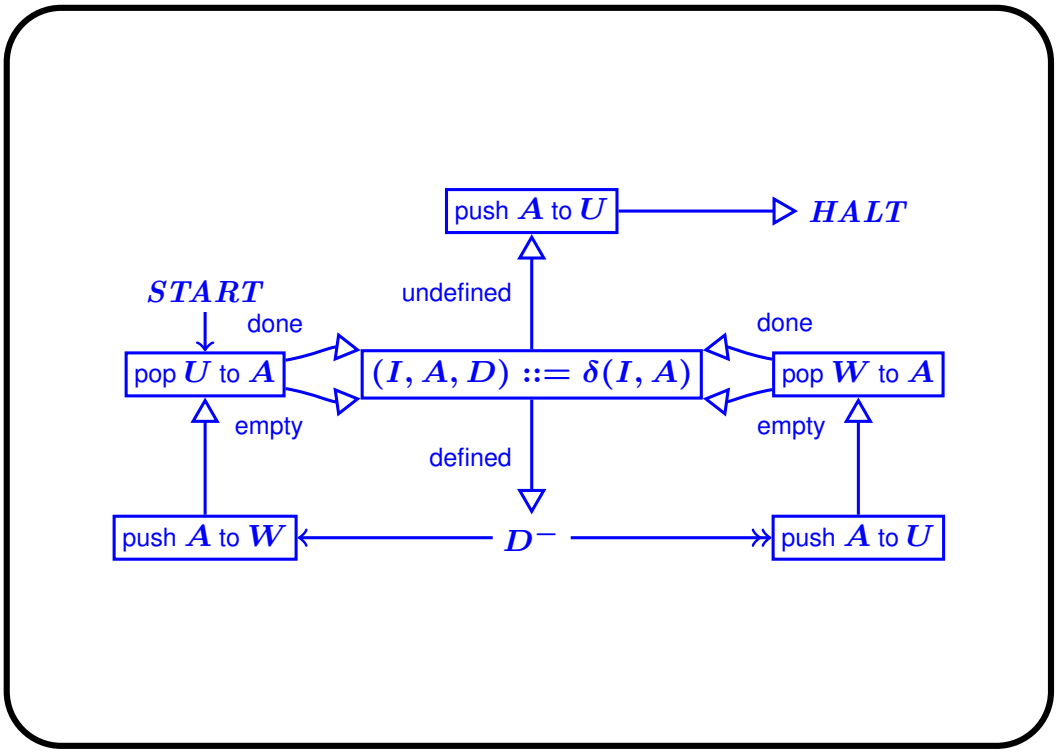
I = current state

W = code of tape symbols left of tape head (reading right-to-left)

U = code of tape symbols at and right of tape head (reading left-to-right)

Starting with U containing the code of the input string (and all other registers zeroed), the RM program halts if and only if M halts; and in that case I, W and U hold the code of the final configuration.

Slide 17



The behaviour of the RM gadget $(Q, A, D) ::= \delta(Q, A)$ has already been described. The behaviour of RM gadget $\ulcorner[Q, W, U]\urcorner ::= C$ is to decode the value of register C to obtain values for the registers Q, W and U. The behaviour of RM gadget $C ::= \ulcorner[Q, W, U]\urcorner$ is to code the list of values of registers Q, W and U to obtain a new value for C. The behaviour of the RM gadget $Q < 2$ is evident.

We've seen that a Turing machine's computation can be implemented by a register machine. The converse holds: the computation of a register machine can be implemented by a Turing machine. To make sense of this, we first have to fix a tape representation of RM configurations, and hence of numbers, lists of numbers.... We will not give the full implementation. We will demonstrate how to encode lists of numbers, and use this enoding to describe **Turing computable** functions.

Slide 18

Tape encoding of lists of numbers

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

Such tapes look like:

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

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

Note the blank spaces: ␣'!

Slide 19

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 $f(x_1, \ldots, x_n) = y$.

Slide 20

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).

Slide 21

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.

Slide 22

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.

The work on Turing machines provided one path to the invention of computers, and plays a significant role in **complexity theory**, the classification of computational problems according to their inherent difficulty and relating these classes to each other. The work on partial recursive functions has led to a branch of mathematics called **recursion theory**. The work on the λ-calculus has led to a branch of computer science called **functional programming**.