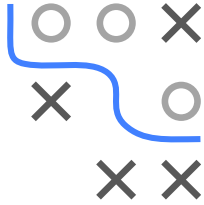
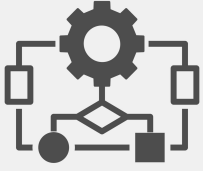
Algorithms and Data Structures



Big O
Algorithms & Turing Machines

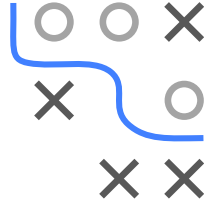


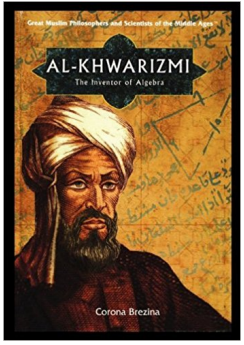
Learning goals

- Definition of algorithm
- Turing machines
- Elementary operations

ALGORITHM

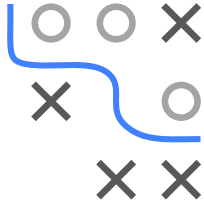
The word *algorithm* is a combination of the Latin word *algorismus*, named after the Persian mathematician Al-Khwarizmi, and the Greek word *arithmos* (number).





In *De numero Indorum* (about 825) Al-Khwarizmi introduced the number zero from the Indian to the Arabic number system. Furthermore, in *Ludus algebrae almucgrabalaeque* (around 830) he provides a new systematic method of solving linear and quadratic systems of equations. The term algebra goes back to this work.

ALGORITHM / 2



What is an algorithm?

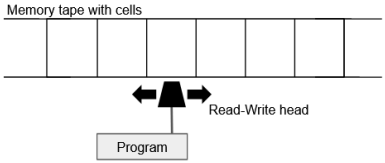
... a set of rules that precisely defines a sequence of operations such that each rule is effective and definite and such that the sequence terminates in a finite time.

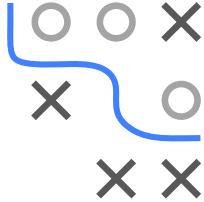
The above definition of the term algorithm is comprehensible, but mathematically inaccurate.

To receive a more precise definition, a number of approaches have been developed in the first half of the 20th century.

With the help of the **turing machine** the term can be specified much more precisely.

TURING MACHINES AND ALGORITHMS

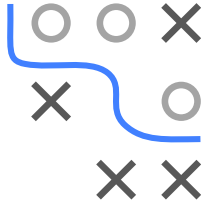




A turing machine consists of

- a control unit that contains the program,
- an infinitely long memory tape with discrete fields or cells,
- a program controlled read-and-write head.

TURING MACHINES AND ALGORITHMS / 2

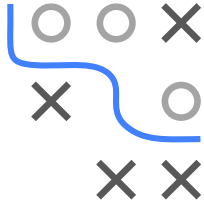


In each step, the turing machine changes to a new state defined in the program. The read-and-write head

- reads or scans the current symbol,
- overwrites it,
- moves a field to the left or right or stops.

The **number** of states is **finite**. The turing machine stops if no transition to a new state is defined for the current state and the symbol read from the current cell.

TURING MACHINES AND ALGORITHMS / 3



Using the concept of the turing machine we define:

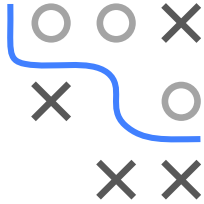
> *A calculation rule for solving a problem is called an algorithm if there exists an equivalent turing machine for this calculation rule which stops for every input that has a solution.*

If a (terminating) algorithm exists, the problem is called **computable**.

Examples:

- Computable: determination of the *n*-th member of the Fibonacci sequence
- Non-computable: The Halting problem (Problem of determining from a description of an arbitrary computer program and an input whether the program will finish running or continue to run forever)

TURING MACHINES AND ALGORITHMS / 4



The field of computability theory deals in particular with the question which problems are computable. In computer science it is usually assumed that the **Church-Turing thesis**

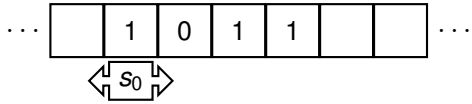
> *The class of turing-computable functions corresponds to the class of intuitively computable functions.*

holds. Since the term "intuitively computable function" cannot be formalized, the thesis cannot be proved.

Computability can also be defined equivalently on the basis of other equally powerful models of computation such as register machines or the lambda calculus. Due to their structural simplicity, turing machines are usually chosen in computability theory.

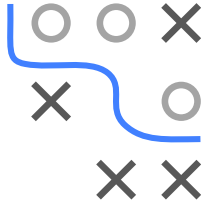
TURING MACHINES AND ALGORITHMS / 5

Example for a turing machine: Addition of 1

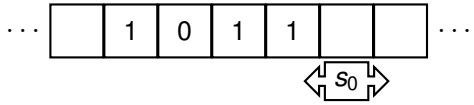


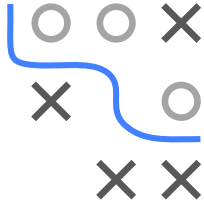
State	Symbol read	Write	Move	Next state
s_0	Blank	Blank	$\leftarrow$	s_1
s_0	0	0	$\rightarrow$	s_0
s_0	1	1	$\rightarrow$	s_0
s_1	Blank	1	$\leftarrow$	s_2
s_1	0	1	$\leftarrow$	s_2
s_1	1	0	$\leftarrow$	s_1
s_2	Blank	Blank	$\rightarrow$	Stop
s_2	0	0	$\leftarrow$	s_2
s_2	1	1	$\leftarrow$	s_2

Interactive example implementation of different basic operations



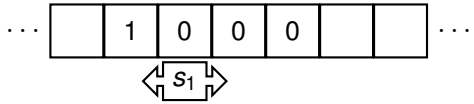
TURING MACHINES AND ALGORITHMS / 6

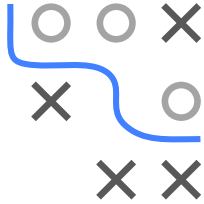




State	Symbol read	Write	Move	Next state
s_0	Blank	Blank	$\leftarrow$	s_1
s_0	0	0	$\rightarrow$	s_0
s_0	1	1	$\rightarrow$	s_0
s_1	Blank	1	$\leftarrow$	s_2
s_1	0	1	$\leftarrow$	s_2
s_1	1	0	$\leftarrow$	s_1
s_2	Blank	Blank	$\rightarrow$	Stop
s_2	0	0	$\leftarrow$	s_2
s_2	1	1	$\leftarrow$	s_2

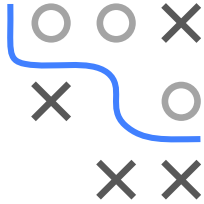
TURING MACHINES AND ALGORITHMS / 7





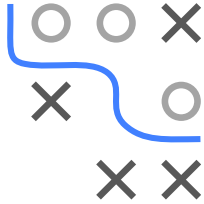
State	Symbol read	Write	Move	Next state
s_0	Blank	Blank	$\leftarrow$	s_1
s_0	0	0	$\rightarrow$	s_0
s_0	1	1	$\rightarrow$	s_0
s_1	Blank	1	$\leftarrow$	s_2
s_1	0	1	$\leftarrow$	s_2
s_1	1	0	$\leftarrow$	s_1
s_2	Blank	Blank	$\rightarrow$	Stop
s_2	0	0	$\leftarrow$	s_2
s_2	1	1	$\leftarrow$	s_2

ALGORITHMS AND ELEMENTARY OPERATIONS



- We can theoretically describe the runtime of an algorithm by using turing machines and counting the number of steps a turing machine needs to solve the problem.
- However, it is usually sufficient to consider an algorithm - less precise but more intuitive - as a **list of instructions**.
- Using this definition, we can "measure" the runtime by counting the number of "elementary operations" performed.
- An elementary operation is a step that does **not** depend on the size of the problem.
- This distinction makes it easier for us to measure the runtime of an algorithm.

ALGORITHMS AND ELEMENTARY OPERATIONS / 2



For simplicity, we assume that the time needed to evaluate a function is **proportional** to the number of "elementary operations" performed.

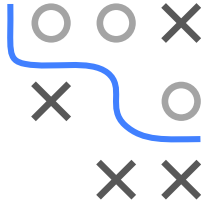
Examples of elementary operations:

- An arithmetic operation, e.g. `x + y`
- An assignment, e.g. `x = 1`
- A test, e.g. `x == 3`

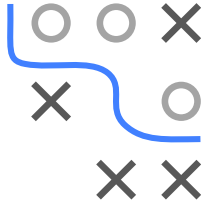
ALGORITHMS AND ELEMENTARY OPERATIONS / 3

Caution:

- Strictly speaking, the runtime of arithmetic operations depends on the problem, e.g. when adding $x + y$, it depends on the size of the numbers x, y.
- Nevertheless, we count each arithmetic operation as one step and assume that the runtime of the operation is not influenced by the number itself.
- In most cases this is not a problem, since the size of the numbers does not grow systematically as the algorithm progresses.
- However, there are exceptions where the complexity depends significantly on the size of the number (e.g., testing whether a number is a prime). Here one would use a different calculation model such as the number of bit operations.



RUNTIME GROWTH (EXAMPLE)



Let's say you want to check if integer `x` is a **prime number**!

For solving the problem you use two different algorithms:

1. The first algorithm divides `x` by every integer from 2 to `x-1` to determine its divisibility.
2. The second algorithm divides `x` by every integer from 2 to `sqrt(x)` to determine its divisibility.

Both algorithms will provide the correct result*, but while the number of basic operations in algorithm 1 is `x-2`, the number of basic operations in algorithm 2 is `< sqrt(x)` and hence for `x > 4` algorithm 2 is much faster and its runtime grows much slower!

*Obviously, if $x_1 > sqrt(x)$, then $x_2 = x/x_1$ results in $x_2 < sqrt(x)$ so you just have to check the integers $\leq sqrt(x)$!