Algorithms and Computability

Lecture 1: Introduction and Turing Machines

Martin Zimmermann (Aalborg University)

An Algorithm

What does this algorithm do?

```
def mystery(n):
  for i in range(2,int(sqrt(n))+1):
    if (n%i) == 0:
      return False
  return True
```

An Algorithm

What does this algorithm do? It tests n for primality.

```
def mystery(n):
  for i in range(2,int(sqrt(n))+1):
    if (n%i) == 0:
      return False
  return True
```

An Algorithm

What does this algorithm do? It tests n for primality.

```python
def mystery(n):
  for i in range(2,int(sqrt(n))+1):
    if (n%i) == 0:
      return False
  return True
```

- What is its runtime?

An Algorithm

What does this algorithm do? It tests n for primality.

```python
def mystery(n):
  for i in range(2,int(sqrt(n))+1):
    if (n%i) == 0:
      return False
  return True
```

- What is its runtime? $\mathcal{O}(\sqrt{n})$

An Algorithm

What does this algorithm do? It tests n for primality.

```python
def mystery(n):
  for i in range(2,int(sqrt(n))+1):
    if (n%i) == 0:
      return False
  return True
```

- What is its runtime? $\mathcal{O}(\sqrt{n})$
- Is this efficient?

An Algorithm

What does this algorithm do? It tests n for primality.

```python
def mystery(n):
  for i in range(2,int(sqrt(n))+1):
    if (n%i) == 0:
      return False
  return True
```

- What is its runtime? $\mathcal{O}(\sqrt{n})$
- Is this efficient?

Example

Testing a 256 bit number for primality this way takes up to 4×10^{38} divisions. At 1 billion divisions per second, that takes approximatively 1.5×10^{22} years.

An Algorithm

What does this algorithm do? It tests n for primality.

```python
def mystery(n):
  for i in range(2,int(sqrt(n))+1):
    if (n%i) == 0:
      return False
  return True
```

- What is its runtime? $\mathcal{O}(\sqrt{n})$
- Is this efficient?

  Example

  Testing a 256 bit number for primality this way takes up to 4×10^{38} divisions. At 1 billion divisions per second, that takes approximatively 1.5×10^{22} years.

- Not efficient: runtime is linear in $\sqrt{n}$, but exponential in the *size* of n (which is $\log n$ in binary representation).

Another Algorithm

What about this algorithm?

```
def mystery(n):
  if n = a^b for some b > 1:
    return False
  r = min{ r | ord(n,r) > log(n)^2 }
  if 1 < gcd(a,n) < n for some a ≤ r:
    return False;
  if n ≤ r:
    return True
  for a in range(1,sqrt(φ(r))*log(n):
    if (X+a)^n ≠ X^n+a (mod (X^r-1), n):
      return False
```

Another Algorithm

What about this algorithm? This is the Agrawal–Kayal–Saxena primality test, the first (2002) unconditional deterministic polynomial-time (in $\log n$) primality test.

```
def mystery(n):
  if n = a^b for some b > 1:
    return False
  r = min{ r | ord(n,r) > log(n)^2 }
  if 1 < gcd(a,n) < n for some a ≤ r:
    return False;
  if n ≤ r:
    return True
  for a in range(1,sqrt(φ(r))*log(n):
    if (X+a)^n ≠ X^n+a (mod (X^r-1), n):
      return False
```

Another Algorithm

What about this algorithm? This is the Agrawal–Kayal–Saxena primality test, the first (2002) unconditional deterministic polynomial-time (in $\log n$) primality test.

```
def mystery(n):
  if n = a^b for some b > 1:
    return False
  r = min{ r | ord(n,r) > log(n)^2 }
  if 1 < gcd(a,n) < n for some a ≤ r:
    return False;
  if n ≤ r:
    return True
  for a in range(1,sqrt(φ(r))*log(n):
    if (X+a)^n ≠ X^n+a (mod (X^r-1), n):
      return False
```

- Requires some number theory to prove correct and efficient!

Another Algorithm

What about this algorithm? This is the Agrawal–Kayal–Saxena primality test, the first (2002) unconditional deterministic polynomial-time (in $\log n$) primality test.

```
def mystery(n):
  if n = a^b for some b > 1:
    return False
  r = min{ r | ord(n,r) > log(n)^2 }
  if 1 < gcd(a,n) < n for some a ≤ r:
    return False;
  if n ≤ r:
    return True
  for a in range(1,sqrt(φ(r))*log(n)):
    if (X+a)^n ≠ X^n+a (mod (X^r-1), n):
      return False
```

- Requires some number theory to prove correct and efficient!
- Agrawal, Kayal, and Saxena received the Gödel and Fulkerson prizes for this work.

A Thief

A thief has a knapsack holding at most W pounds of loot. He robs a store that has items $1, \ldots, n$ of weight w_j and value c_j (each item only once). What is the maximal value the thief can put in his knapsack?

Example

$W = 50$ and the following items:

item	weight	value	value per pound
1	10 pound	\$60	\$6
2	20 pound	\$100	\$5
3	30 pound	\$120	\$4

A Thief

A thief has a knapsack holding at most W pounds of loot. He robs a store that has items $1, \ldots, n$ of weight w_j and value c_j (each item only once). What is the maximal value the thief can put in his knapsack?

Example

$W = 50$ and the following items:

item	weight	value	value per pound
1	10 pound	\$60	\$6
2	20 pound	\$100	\$5
3	30 pound	\$120	\$4

- "Greedy" solution (item 1 and item 2) has value \$160
- Optimal solution (item 2 and item 3) has value \$220

Collatz

Does the following algorithm return `True` for every possible input $n \geq 1$?

```
def collatz(n):
  while(n > 1):
    if n%2 == 0:
      n = n/2
    else:
      n = 3*n+1
  return True
```

Collatz

Does the following algorithm return `True` for every possible input $n \geq 1$?

```python
def collatz(n):
  while(n > 1):
    if n%2 == 0:
      n = n/2
    else:
      n = 3*n+1
  return True
```

- Can you compute whether a given program returns `True` for every input?

Collatz

Does the following algorithm return `True` for every possible input $n \geq 1$?

```
def collatz(n):
  while(n > 1):
    if n%2 == 0:
      n = n/2
    else:
      n = 3*n+1
  return True
```

- Can you compute whether a given program returns `True` for every input?
- All $n \leq 2^{68}$ have been checked, and yield `True`. But that does not mean much!

Collatz

Does the following algorithm return `True` for every possible input $n \geq 1$?

```
def collatz(n):
  while(n > 1):
    if n%2 == 0:
      n = n/2
    else:
      n = 3*n+1
  return True
```

- Can you compute whether a given program returns `True` for every input?
- All $n \leq 2^{68}$ have been checked, and yield `True`. But that does not mean much!
- Nobody knows whether the above algorithm always returns `True`! Erdős: "Mathematics may not be ready for such problems."

Purpose

In previous courses, you have

- seen algorithms that solve specific tasks, e.g., sorting an array, searching a path through a graph, and
- have analyzed the asymptotic runtime of such algorithms.

Purpose

In previous courses, you have

- seen algorithms that solve specific tasks, e.g., sorting an array, searching a path through a graph, and
- have analyzed the asymptotic runtime of such algorithms.

Here, we study questions like:

- Can every problem be solved (efficiently) by an algorithm?
- Actually, what is an algorithm? And what is a problem? And what does efficiently mean?
- How can we still solve *hard* problems?

Purpose

In previous courses, you have

- seen algorithms that solve specific tasks, e.g., sorting an array, searching a path through a graph, and
- have analyzed the asymptotic runtime of such algorithms.

Here, we study questions like:

- Can every problem be solved (efficiently) by an algorithm?
- Actually, what is an algorithm? And what is a problem? And what does efficiently mean?
- How can we still solve *hard* problems?

And we will see how all the examples we have seen so far relate to these questions.

Agenda

1. **Course Formalities**
2. Setting the Stage
3. Turing Machines

Literature

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: Introduction to Algorithms, 3rd or 4th edition

- MIT Press
- ISBN: 9780262533058.
- Note that this is the same book you used for your semester 2 course on "Algorithms and Data Structures".

Hubie Chen: Computability and Complexity

- MIT Press
- ISBN: 9780262048620

Course Format

Schedule

- **Lectures:** Thursdays 12:30– 14:15 and Fridays 8:15-10:00
- **Exercises:** Thursdays 14:30 – 16:15 and Fridays 10:15–11:45
- Roughly every other week (check calendar on Moodle)
- Workload: ca. 150 hours (note that lectures and exercises only account for a third of the workload, the rest is your weekly reading, revising sample solutions, and exam preparation)

Course Format

Schedule

- **Lectures:** Thursdays 12:30– 14:15 and Fridays 8:15-10:00
- **Exercises:** Thursdays 14:30 – 16:15 and Fridays 10:15–11:45
- Roughly every other week (check calendar on Moodle)
- Workload: ca. 150 hours (note that lectures and exercises only account for a third of the workload, the rest is your weekly reading, revising sample solutions, and exam preparation)

Exercises

- Available on Moodle
- Essential preparation for the exam!

Course Format

Schedule

- **Lectures:** Thursdays 12:30– 14:15 and Fridays 8:15-10:00
- **Exercises:** Thursdays 14:30 – 16:15 and Fridays 10:15–11:45
- Roughly every other week (check calendar on Moodle)
- Workload: ca. 150 hours (note that lectures and exercises only account for a third of the workload, the rest is your weekly reading, revising sample solutions, and exam preparation)

Exercises

- Available on Moodle
- Essential preparation for the exam!

Exam

- Main exam written, re-exam (tentatively) oral. More information on Moodle.

More Stuff

In case of questions do (in this order):

1. Discuss in group.
2. Use the discussion forum on Moodle (everybody will benefit from the answer).
3. Write an email: `mzi@cs.aau.dk`.

More Stuff

In case of questions do (in this order):

1. Discuss in group.
2. Use the discussion forum on Moodle (everybody will benefit from the answer).
3. Write an email: `mzi@cs.aau.dk`.

Any questions?

Agenda

1. Course Formalities
2. **Setting the Stage**
3. Turing Machines

Motivating Question

- Actually, what is an algorithm? And what is a problem?

What is a Problem?

In the theoretical parts of this course, we are mostly concerned with so-called decision problems, e.g.,

- Is a given number prime?
- Does a given graph have a path from a given source vertex to a given destination vertex.
- Can the thief put $250 worth of loot in his knapsack?
- Is a given formula of propositional logic satisfiable?
- Can the vertices of a given graph be colored with three colors such that no two neighbors have the same color?
- Does a given program ever output `False`?
- ...

General format: Yes/No question over a (typically infinite) set of inputs.

Abstraction

There are just too many different types of inputs, e.g., numbers, graphs, formulas, programs, sets of linear inequalities, polynomials, etc.

Abstraction

There are just too many different types of inputs, e.g., numbers, graphs, formulas, programs, sets of linear inequalities, polynomials, etc.

To simplify our setting, we only consider sequences of symbols as inputs (i.e., words over an alphabet), e.g.,

- a number is encoded in binary or decimal,
- a graph is encoded by (a linearization) of its adjacency matrix,
- a program is given by its source code.

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.

  Examples:

  - The set $\mathbb{B} = \{0, 1\}$ of binary digits
  - The set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ of decimal digits
  - The roman alphabet $\{a, b, c, \dots\}$
  - $\Sigma_L = \{\neg, \wedge, \vee, (,), p, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.
- A word over an alphabet Σ is a finite sequence of letters from Σ.

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.
- A word over an alphabet Σ is a finite sequence of letters from Σ.

Examples:

- 0, 1, 110, 11111100111 over $\mathbb{B}$
- *alan* and *mathison* over the roman alphabet, but also *crwth* and *ghfbfdtnjs*.
- $(p0 \wedge p1) \vee p23$ over Σ_L, but also $\wedge\wedge)(23p$.

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.
- A word over an alphabet Σ is a finite sequence of letters from Σ.
- The empty string is denoted by ε (the unique empty sequence of letters).

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.
- A word over an alphabet Σ is a finite sequence of letters from Σ.
- The empty string is denoted by ε (the unique empty sequence of letters).
- The concatenation of words w_1 and w_2 is the word $w_1 w_2$.

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.
- A word over an alphabet Σ is a finite sequence of letters from Σ.
- The empty string is denoted by ε (the unique empty sequence of letters).
- The concatenation of words w_1 and w_2 is the word $w_1 w_2$.

Example:

- If $w_1 = \mathit{algorithms}$ and $w_2 = \mathit{computability}$, then $w_1 w_2 = \mathit{algorithmscomputability}$

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.
- A word over an alphabet Σ is a finite sequence of letters from Σ.
- The empty string is denoted by ε (the unique empty sequence of letters).
- The concatenation of words w_1 and w_2 is the word $w_1 w_2$.
- The length of a word w is denoted by $|w|$ and defined as the number of letters in w.

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.
- A word over an alphabet Σ is a finite sequence of letters from Σ.
- The empty string is denoted by ε (the unique empty sequence of letters).
- The concatenation of words w_1 and w_2 is the word $w_1 w_2$.
- The length of a word w is denoted by $|w|$ and defined as the number of letters in w.

Examples:

- $|0| = |1| = 1$, $|110| = 3$, and $|11111100111| = 11$
- $|alan| = 4$ and $|mathison| = 8$
- $|\varepsilon| = 0$

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.
- A word over an alphabet Σ is a finite sequence of letters from Σ.
- The empty string is denoted by ε (the unique empty sequence of letters).
- The concatenation of words w_1 and w_2 is the word $w_1 w_2$.
- The length of a word w is denoted by $|w|$ and defined as the number of letters in w.
- The set of all words over an alphabet Σ is denoted by Σ^*.

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.
- A word over an alphabet Σ is a finite sequence of letters from Σ.
- The empty string is denoted by ε (the unique empty sequence of letters).
- The concatenation of words w_1 and w_2 is the word $w_1 w_2$.
- The length of a word w is denoted by $|w|$ and defined as the number of letters in w.
- The set of all words over an alphabet Σ is denoted by Σ^*.

Example:

- $\mathbb{B}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, \ldots\}$

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.
- A word over an alphabet Σ is a finite sequence of letters from Σ.
- The empty string is denoted by ε (the unique empty sequence of letters).
- The concatenation of words w_1 and w_2 is the word $w_1 w_2$.
- The length of a word w is denoted by $|w|$ and defined as the number of letters in w.
- The set of all words over an alphabet Σ is denoted by Σ^*.
- A language over an alphabet Σ is a subset of Σ^*.

Reminder: Formal Languages

- An alphabet is a finite, nonempty set of letters.
- A word over an alphabet Σ is a finite sequence of letters from Σ.
- The empty string is denoted by ε (the unique empty sequence of letters).
- The concatenation of words w_1 and w_2 is the word $w_1 w_2$.
- The length of a word w is denoted by $|w|$ and defined as the number of letters in w.
- The set of all words over an alphabet Σ is denoted by Σ^*.
- A language over an alphabet Σ is a subset of Σ^*.

Examples:

- $\{0^n1 \mid n \geq 0\}$ and $\{0^n1^n \mid n \geq 0\}$ over $\mathbb{B}$
- the set $\{2, 3, 5, 7, 11, 13, 17, 19, 23, \ldots\}$ of prime numbers
- the set $\{10, 11, 101, 111, 1011, \ldots\}$ of prime numbers
- the set of words in todays newspaper

Reminder: Operations on Languages

Let L_1, L_2 be two languages over Σ.

- union:

$$L_1 \cup L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ or } w \in L_2\}$$

- intersection:

$$L_1 \cap L_2 = \{w \in \Sigma^* \mid w \in L_1 \text{ and } w \in L_2\}$$

- complement (w.r.t. Σ^*):

$$\overline{L_1} = \{w \in \Sigma^* \mid w \notin L_1\}$$

- concatenation:

$$L_1 \cdot L_2 = \{w \in \Sigma^* \mid w = w_1 w_2 \text{ with } w_1 \in L_1 \text{ and } w_2 \in L_2\}$$

- Kleene star (iteration):

$$(L_1)^* = \{w \in \Sigma^* \mid w = w_1 w_2 \cdots w_k \text{ for some } k \geq 0 \text{ and } w_i \in L_1 \text{ for all } i \in \{1, 2, \ldots, k\}\}$$

From Problems to Languages

- Is a given number prime?

$$\{w \in \{0,1,2,3,4,5,6,7,8,9\}^* \mid w \text{ is prime}\}$$

From Problems to Languages

- Is a given number prime?

$$\{w \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \mid w \text{ is prime}\}$$

- Does a given graph have a path from a given source vertex to a given destination vertex.

$$\{w \in \mathbb{B}^* \mid w \text{ encodes adjacency matrix that has a path from vertex 1 to vertex 2}\}$$

From Problems to Languages

- Is a given number prime?

$$\{w \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \mid w \text{ is prime}\}$$

- Does a given graph have a path from a given source vertex to a given destination vertex.

$$\{w \in \mathbb{B}^* \mid w \text{ encodes adjacency matrix that has a path} \\ \text{from vertex 1 to vertex 2}\}$$

- Is a given formula of propositional logic satisfiable?

$$\{w \in \Sigma_L^* \mid w \text{ encodes satisfiable formula}\}$$

From Problems to Languages

- Is a given number prime?

$$\{w \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}^* \mid w \text{ is prime}\}$$

- Does a given graph have a path from a given source vertex to a given destination vertex.

$$\{w \in \mathbb{B}^* \mid w \text{ encodes adjacency matrix that has a path from vertex 1 to vertex 2}\}$$

- Is a given formula of propositional logic satisfiable?

$$\{w \in \Sigma_L^* \mid w \text{ encodes satisfiable formula}\}$$

- Does a given program ever output `False`?

$$\{w \in \{a, b, c, \dots\}^* \mid w \text{ is Python source code of function that outputs } \texttt{False} \text{ for some input}\}$$

Solving Problems = Language Membership

From now on: Decision problems = formal languages.

- So, to solve a decision problem $L \subseteq \Sigma^*$, we "just" need an algorithm that, given an input $w \in \Sigma^*$, returns `True` if $w \in L$ and `False` if $w \notin L$.
- This is easy enough for some problems, but seems much harder for others (e.g., Python programs outputting `false`).

Solving Problems = Language Membership

From now on: Decision problems = formal languages.

- So, to solve a decision problem $L \subseteq \Sigma^*$, we "just" need an algorithm that, given an input $w \in \Sigma^*$, returns `True` if $w \in L$ and `False` if $w \notin L$.
- This is easy enough for some problems, but seems much harder for others (e.g., Python programs outputting `false`).
- So, can every decision problem be algorithmically solved?

Solving Problems = Language Membership

From now on: Decision problems = formal languages.

- So, to solve a decision problem $L \subseteq \Sigma^*$, we "just" need an algorithm that, given an input $w \in \Sigma^*$, returns `True` if $w \in L$ and `False` if $w \notin L$.
- This is easy enough for some problems, but seems much harder for others (e.g., Python programs outputting `false`).
- So, can every decision problem be algorithmically solved?
- To answer this question, we need a formal definition of algorithm to be able to argue that there is a problem that is not solved by any algorithm.

In the remainder of this lecture, we present one such definition (we will later during the course discuss to which extent such a definition is actually possible).

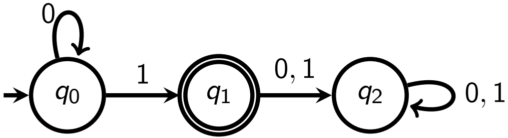
Reminder: Finite Automata

A deterministic finite automaton (DFA) has the form $(Q, \Sigma, q_I, \delta, F)$ where

- Q is a finite set of states,
- Σ is an alphabet,
- $q_I \in Q$ is the initial state,
- $\delta\colon Q \times \Sigma \to Q$ is the transition function, and
- $F \subseteq Q$ is a set of accepting states.

Example

A DFA for the language $\{0^n 1 \mid n \geq 0\}$:



Reminder: Finite Automata

A deterministic finite automaton (DFA) has the form $(Q, \Sigma, q_I, \delta, F)$ where

- Q is a finite set of states,
- Σ is an alphabet,
- $q_I \in Q$ is the initial state,
- $\delta\colon Q \times \Sigma \to Q$ is the transition function, and
- $F \subseteq Q$ is a set of accepting states.

- So, DFAs can be seen as a (very weak) formalization of algorithms for decision problems.
- In the remainder of this course, we study a much stronger formalization.

Agenda

1. Course Formalities
2. Setting the Stage
3. **Turing Machines**

Alan Turing: On Computable Numbers (1936)

Computing is normally done by writing certain symbols on paper. "We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares. [...]

Alan Turing: On Computable Numbers (1936)

Computing is normally done by writing certain symbols on paper. "We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares. [...]
The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. [..]

Alan Turing: On Computable Numbers (1936)

Computing is normally done by writing certain symbols on paper. "We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares. [...]
The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. [..]
We will also suppose that the number of states of mind which need be taken into account is finite. [..]

Alan Turing: On Computable Numbers (1936)

Computing is normally done by writing certain symbols on paper. "We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares. [...]
The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. [..]
We will also suppose that the number of states of mind which need be taken into account is finite. [..]
Let us imagine the operations performed by the computer to be split up into "simple operations" which are so elementary that it is not easy to imagine them further divided. Every such operation consists of some change of the physical system consisting of the computer and his tape. We know the state of the system if we know the sequence of symbols on the tape, which of these are observed by the computer [..], and the state of mind of the computer. We may suppose that in a simple operation not more than one symbol is altered.

Conceptual View

A Turing Machine:

- An infinite tape of paper, divided into squares (often called cells).

Conceptual View

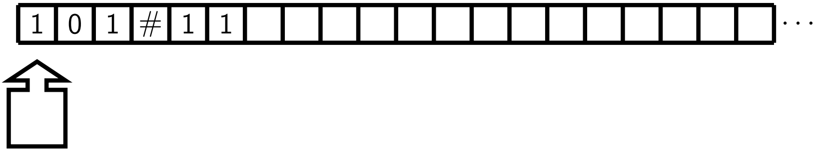
A Turing Machine:

1	0	1	#	1	1															$\cdots$

- An infinite tape of paper, divided into squares (often called cells).
- Symbols in some squares.

Conceptual View

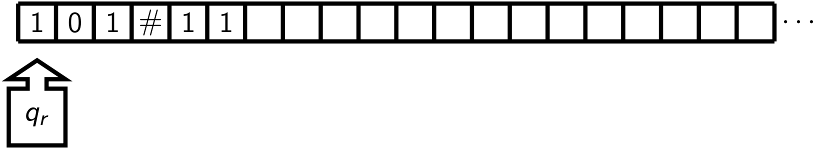
A Turing Machine:



- An infinite tape of paper, divided into squares (often called cells).
- Symbols in some squares.
- A single square that is currently observed (with a reading/writing head).

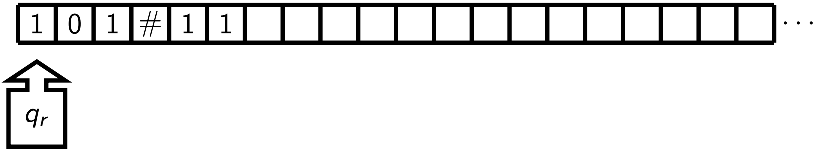
Conceptual View

A Turing Machine:



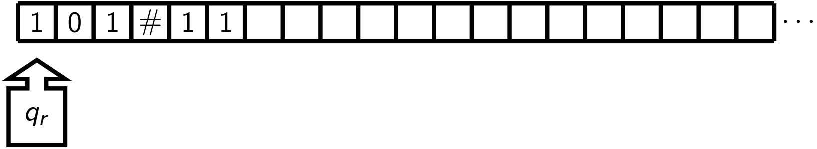
- An infinite tape of paper, divided into squares (often called cells).
- Symbols in some squares.
- A single square that is currently observed (with a reading/writing head).
- A “state of mind”.

A Turing Machine:



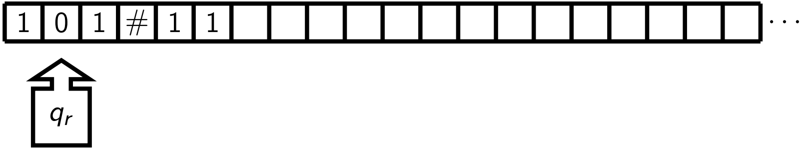
- An infinite tape of paper, divided into squares (often called cells).
- Symbols in some squares.
- A single square that is currently observed (with a reading/writing head).
- A "state of mind".
- Rules updating the state and currently observed square.

A Turing Machine:



- `If state is` q_r `and symbol is` 0 `then change to state` q_r `change symbol to` 0 `and move in direction 'right'`
- `If state is` q_r `and symbol is` 1 `then change to state` q_r `change symbol to` 1 `and move in direction 'right'`
- `If state is` q_r `and symbol is` $\#$ `then change to state` q_r `change symbol to` $\#$ `and move in direction 'right'`
- `If state is` q_r `and symbol is 'empty' then change to state` q_s `change symbol to 'empty' and move in direction 'left'`

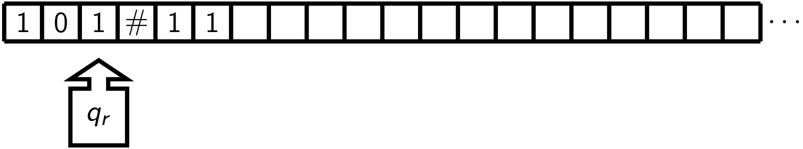
A Turing Machine:



- If state is q_r and symbol is 0 then change to state q_r change symbol to 0 and move in direction 'right'
- If state is q_r and symbol is 1 then change to state q_r change symbol to 1 and move in direction 'right'
- If state is q_r and symbol is # then change to state q_r change symbol to # and move in direction 'right'
- If state is q_r and symbol is 'empty' then change to state q_s change symbol to 'empty' and move in direction 'left'

Conceptual View

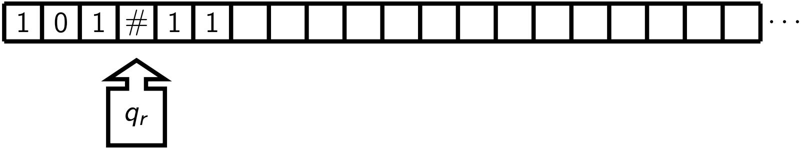
A Turing Machine:



- If state is q_r and symbol is 0 then change to state q_r change symbol to 0 and move in direction 'right'
- If state is q_r and symbol is 1 then change to state q_r change symbol to 1 and move in direction 'right'
- If state is q_r and symbol is # then change to state q_r change symbol to # and move in direction 'right'
- If state is q_r and symbol is 'empty' then change to state q_s change symbol to 'empty' and move in direction 'left'

Conceptual View

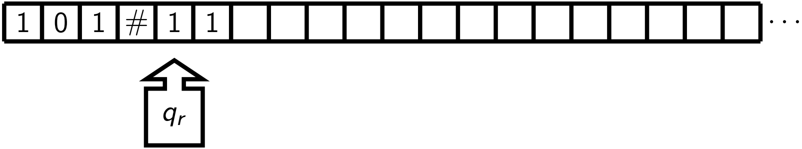
A Turing Machine:



- If state is q_r and symbol is 0 then change to state q_r change symbol to 0 and move in direction 'right'
- If state is q_r and symbol is 1 then change to state q_r change symbol to 1 and move in direction 'right'
- If state is q_r and symbol is # then change to state q_r change symbol to # and move in direction 'right'
- If state is q_r and symbol is 'empty' then change to state q_s change symbol to 'empty' and move in direction 'left'

Conceptual View

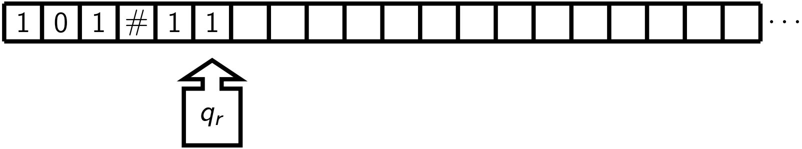
A Turing Machine:



- If state is q_r and symbol is 0 then change to state q_r change symbol to 0 and move in direction 'right'
- If state is q_r and symbol is 1 then change to state q_r change symbol to 1 and move in direction 'right'
- If state is q_r and symbol is # then change to state q_r change symbol to # and move in direction 'right'
- If state is q_r and symbol is 'empty' then change to state q_s change symbol to 'empty' and move in direction 'left'

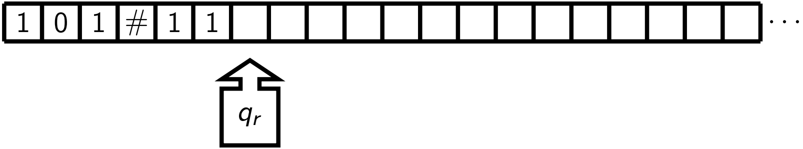
Conceptual View

A Turing Machine:



- `If state is` q_r `and symbol is` 0 `then change to state` q_r `change symbol to` 0 `and move in direction 'right'`
- `If state is` q_r `and symbol is` 1 `then change to state` q_r `change symbol to` 1 `and move in direction 'right'`
- `If state is` q_r `and symbol is` # `then change to state` q_r `change symbol to` # `and move in direction 'right'`
- `If state is` q_r `and symbol is 'empty' then change to state` q_s `change symbol to 'empty' and move in direction 'left'`

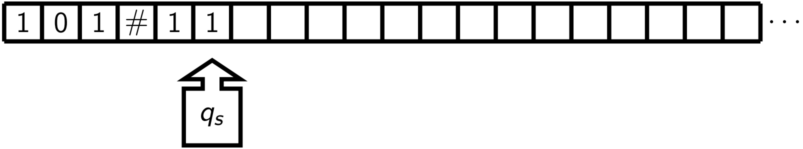
A Turing Machine:



- If state is q_r and symbol is 0 then change to state q_r change symbol to 0 and move in direction 'right'
- If state is q_r and symbol is 1 then change to state q_r change symbol to 1 and move in direction 'right'
- If state is q_r and symbol is # then change to state q_r change symbol to # and move in direction 'right'
- If state is q_r and symbol is 'empty' then change to state q_s change symbol to 'empty' and move in direction 'left'

Conceptual View

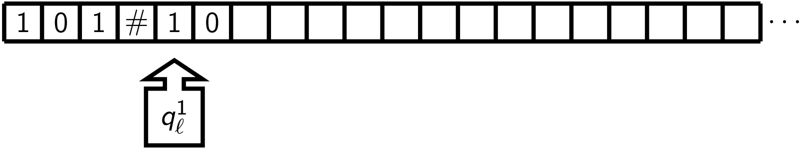
A Turing Machine:



- If state is q_s and symbol is 1 then change to state q_ℓ^1 change symbol to 0 and move in direction 'left'
- If state is q_ℓ^1 and symbol is 1 then change to state q_ℓ^1 change symbol to 1 and move in direction 'left'
- If state is q_ℓ^1 and symbol is # then change to state q_a^1 change symbol to # and move in direction 'left'
- If state is q_a^1 and symbol is 1 then change to state q_a^1 change symbol to 0 and move in direction 'left'

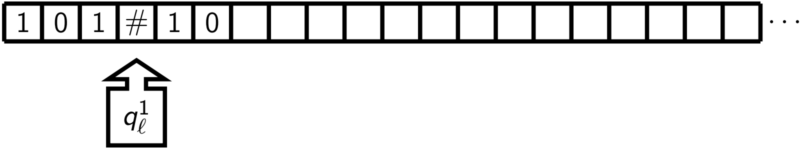
Conceptual View

A Turing Machine:



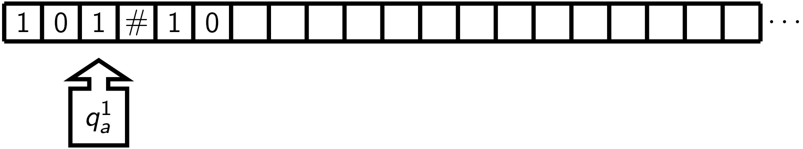
- If state is q_s and symbol is 1 then change to state q_ℓ^1 change symbol to 0 and move in direction 'left'
- If state is q_ℓ^1 and symbol is 1 then change to state q_ℓ^1 change symbol to 1 and move in direction 'left'
- If state is q_ℓ^1 and symbol is $\#$ then change to state q_a^1 change symbol to $\#$ and move in direction 'left'
- If state is q_a^1 and symbol is 1 then change to state q_a^1 change symbol to 0 and move in direction 'left'

A Turing Machine:



- If state is q_s and symbol is 1 then change to state q_ℓ^1 change symbol to 0 and move in direction 'left'
- If state is q_ℓ^1 and symbol is 1 then change to state q_ℓ^1 change symbol to 1 and move in direction 'left'
- If state is q_ℓ^1 and symbol is # then change to state q_a^1 change symbol to # and move in direction 'left'
- If state is q_a^1 and symbol is 1 then change to state q_a^1 change symbol to 0 and move in direction 'left'

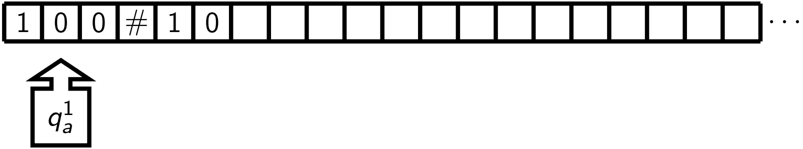
A Turing Machine:



- If state is q_s and symbol is 1 then change to state q_ℓ^1 change symbol to 0 and move in direction 'left'
- If state is q_ℓ^1 and symbol is 1 then change to state q_ℓ^1 change symbol to 1 and move in direction 'left'
- If state is q_ℓ^1 and symbol is # then change to state q_a^1 change symbol to # and move in direction 'left'
- If state is q_a^1 and symbol is 1 then change to state q_a^1 change symbol to 0 and move in direction 'left'

Conceptual View

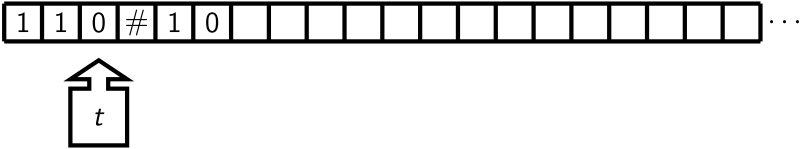
A Turing Machine:



- `If state is` q_a^1 `and symbol is` 0 `then change to state` t `change symbol to` 1 `and move in direction 'right'`

Conceptual View

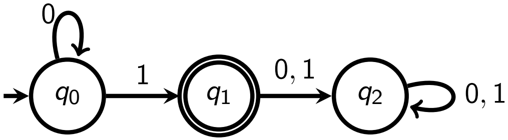
A Turing Machine:



- `If state is` q_a^1 `and symbol is` 0 `then change to state` t `change symbol to` 1 `and move in direction 'right'`

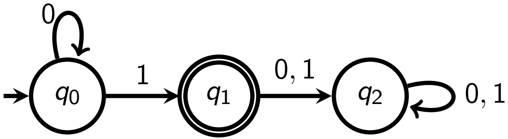
DFA as Turing Machines

Meaning of a transition $\delta(q, a) = q'$: `if in state` q `and next letter is` a `change to state` q'



DFA as Turing Machines

Meaning of a transition $\delta(q, a) = q'$: `if in state` q `and next letter is` a `change to state` q'

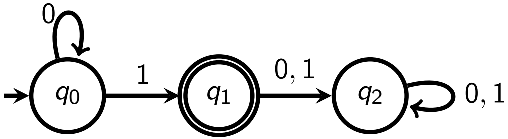


We can understand a DFA as a Turing machine that

- moves its head to the right until the first blank cell is reached and
- only reads the tape, but never writes to it.

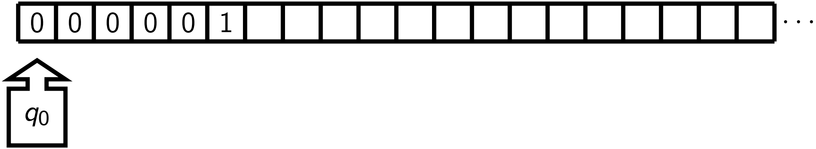
DFA as Turing Machines

Meaning of a transition $\delta(q, a) = q'$: `if in state` q `and next letter is` a `change to state` q'



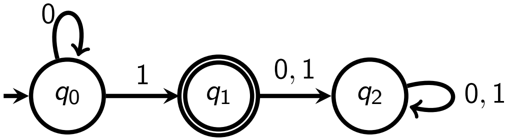
We can understand a DFA as a Turing machine that

- moves its head to the right until the first blank cell is reached and
- only reads the tape, but never writes to it.



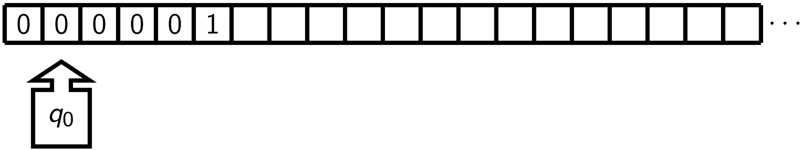
DFA as Turing Machines

Meaning of a transition $\delta(q, a) = q'$: `if in state` q `and next letter is` a `change to state` q'



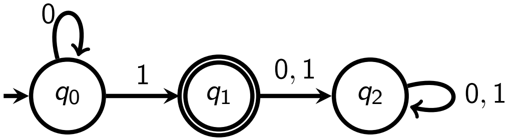
We can understand a DFA as a Turing machine that

- moves its head to the right until the first blank cell is reached and
- only reads the tape, but never writes to it.



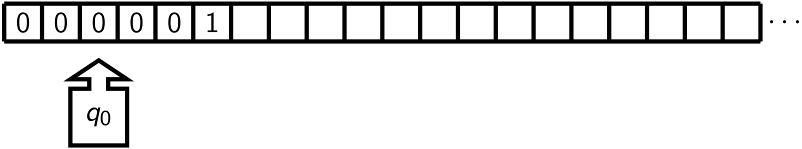
DFA as Turing Machines

Meaning of a transition $\delta(q, a) = q'$: `if in state` q `and next letter is` a `change to state` q'



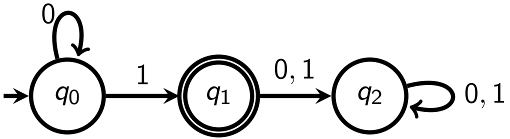
We can understand a DFA as a Turing machine that

- moves its head to the right until the first blank cell is reached and
- only reads the tape, but never writes to it.



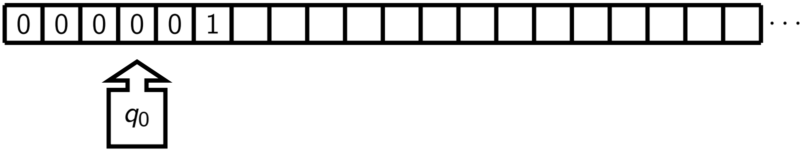
DFA as Turing Machines

Meaning of a transition $\delta(q, a) = q'$: `if in state` q `and next letter is` a `change to state` q'

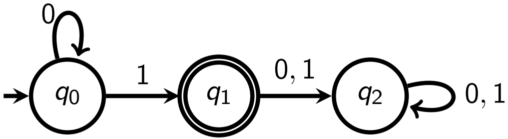


We can understand a DFA as a Turing machine that

- moves its head to the right until the first blank cell is reached and
- only reads the tape, but never writes to it.

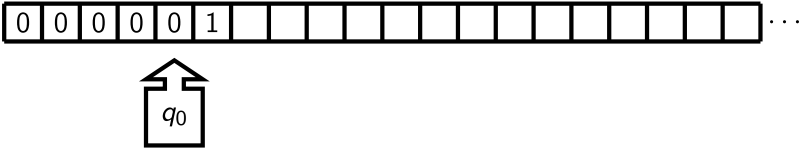


Meaning of a transition $\delta(q, a) = q'$: `if in state` q `and next letter is` a `change to state` q'



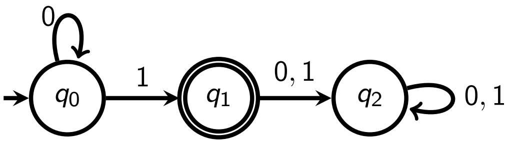
We can understand a DFA as a Turing machine that

- moves its head to the right until the first blank cell is reached and
- only reads the tape, but never writes to it.



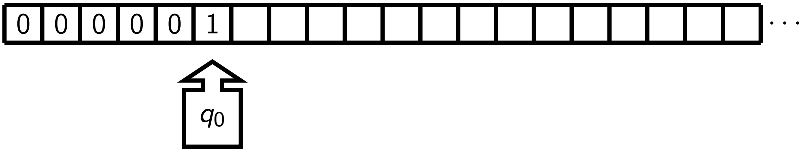
DFA as Turing Machines

Meaning of a transition $\delta(q, a) = q'$: `if in state` q `and next letter is` a `change to state` q'



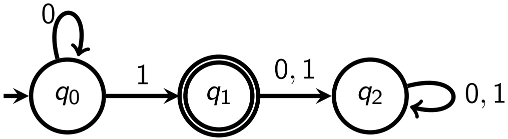
We can understand a DFA as a Turing machine that

- moves its head to the right until the first blank cell is reached and
- only reads the tape, but never writes to it.



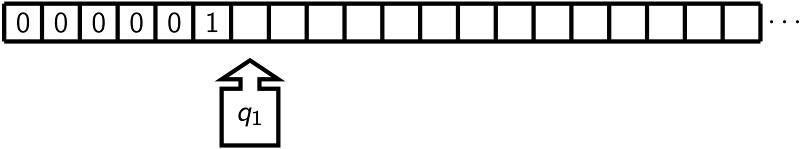
DFA as Turing Machines

Meaning of a transition $\delta(q, a) = q'$: `if in state` q `and next letter is` a `change to state` q'



We can understand a DFA as a Turing machine that

- moves its head to the right until the first blank cell is reached and
- only reads the tape, but never writes to it.



Formal Definition

A deterministic Turing machine (DTM) is a 7-tuple $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$

- Q is a finite set of states,
- Σ is the input alphabet,
- $\Gamma \supseteq \Sigma$ is the tape alphabet s.t. $\sqcup \in \Gamma \setminus \Sigma$ (the blank symbol),
- $s \in Q$ is the starting (or initial) state,
- $t \in Q$ is the accepting state, and
- $r \in Q$ is the rejecting state (we require $t \neq r$), and
- $\delta\colon (Q \setminus \{t, r\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$ is the transition function.

Formal Definition

A deterministic Turing machine (DTM) is a
7-tuple $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$

- Q is a finite set of states,
- Σ is the input alphabet,
- $\Gamma \supseteq \Sigma$ is the tape alphabet s.t. $\sqcup \in \Gamma \setminus \Sigma$ (the blank symbol),
- $s \in Q$ is the starting (or initial) state,
- $t \in Q$ is the accepting state, and
- $r \in Q$ is the rejecting state (we require $t \neq r$), and
- $\delta \colon (Q \setminus \{t, r\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$ is the transition function.

Notation

So, we write $\delta(q, a) = (q', a', d)$ (using -1 for `left` and $+1$ for `right`) instead of
"`If state is` q `and symbol is a then change to state` q' `change symbol to a' and move in direction d`".

Running a Turing Machine: Intuition

Let $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ be a Turing machine. Given an input $w \in \Sigma^*$:

- Initialization: w is on the tape (all other cells are blank), reading head is on first letter of w (if w is nonempty), in state s.
- Execution: Apply transition function repeatedly until termination, thereby updating the tape contents, the state, and the position of the reading head.
- Termination: Stop, if state t or r is reached.

Question

Is there another option than reaching t or r?

Configurations

Definition

Let $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ be a DTM. A configuration of M is a triple $[q, \tau, \ell]$ where

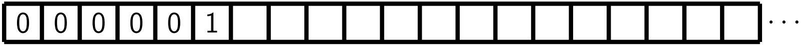
- $q \in Q$ is the current state,
- $\tau\colon \mathbb{N}^+ \to \Gamma$ representing the current tape content, and
- $\ell \in \mathbb{N}^+$ denotes the current location of the reading head.

Configurations

Definition

Let $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ be a DTM. A configuration of M is a triple $[q, \tau, \ell]$ where

- $q \in Q$ is the current state,
- $\tau \colon \mathbb{N}^+ \to \Gamma$ representing the current tape content, and
- $\ell \in \mathbb{N}^+$ denotes the current location of the reading head.

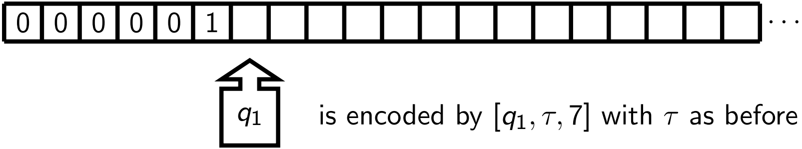


is encoded by $[q_0, \tau, 3]$ with $\tau(n) = \begin{cases} 0 & \text{if } n \leq 5, \\ 1 & \text{if } n = 6, \\ \sqcup & \text{if } n \geq 7. \end{cases}$

Definition

Let $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ be a DTM. A configuration of M is a triple $[q, \tau, \ell]$ where

- $q \in Q$ is the current state,
- $\tau \colon \mathbb{N}^+ \to \Gamma$ representing the current tape content, and
- $\ell \in \mathbb{N}^+$ denotes the current location of the reading head.



is encoded by $[q_1, \tau, 7]$ with τ as before

A Remark on Notation

- Although the tape is infinite, only finitely many cells are non-blank at any time.
- Accordingly, let τ be a function such that there is an n_0 such that $\tau(n) = \llcorner\!\lrcorner$ for all $n > n_0$. We then often write $\tau(1)\tau(2)\cdots\tau(n_0)\llcorner\!\lrcorner\ldots$ for τ.

A Remark on Notation

- Although the tape is infinite, only finitely many cells are non-blank at any time.
- Accordingly, let τ be a function such that there is an n_0 such that $\tau(n) = \sqcup$ for all $n > n_0$. We then often write $\tau(1)\tau(2)\cdots\tau(n_0)\sqcup\ldots$ for τ.

Example

Let $\tau(n) = \begin{cases} 0 & \text{if } n \leq 5, \\ 1 & \text{if } n = 6, \\ \sqcup & \text{if } n \geq 7. \end{cases}$. It is represented by $000001\sqcup\ldots$.

More on Configurations

Definition

Let $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ be a DTM.

More on Configurations

Definition

Let $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ be a DTM.

- The initial configuration of M on input $w \in \Sigma^*$ is $[s, w\llcorner\ldots, 1]$.

More on Configurations

Definition

Let $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ be a DTM.

- The initial configuration of M on input $w \in \Sigma^*$ is $[s, w\textvisiblespace\ldots, 1]$.
- The successor configuration of a configuration $[q, \tau, \ell]$ is defined whenever $q \in Q \setminus \{t, r\}$. In this case, let $\delta(q, \tau(\ell)) = (p, a, d)$. Then, the (unique!) successor configuration of $[q, \tau, \ell]$ is defined as $[p, \tau', \max\{\ell + d, 1\}]$ where

$$\tau'(n) = \begin{cases} a & \text{if } n = \ell, \\ \tau(n) & \text{otherwise.} \end{cases}$$

More on Configurations

Definition

Let $M = (Q, \Sigma, \Gamma, s, t, r, \delta)$ be a DTM.

- The initial configuration of M on input $w \in \Sigma^*$ is $[s, w\sqcup \ldots, 1]$.
- The successor configuration of a configuration $[q, \tau, \ell]$ is defined whenever $q \in Q \setminus \{t, r\}$. In this case, let $\delta(q, \tau(\ell)) = (p, a, d)$. Then, the (unique!) successor configuration of $[q, \tau, \ell]$ is defined as $[p, \tau', \max\{\ell + d, 1\}]$ where
$$\tau'(n) = \begin{cases} a & \text{if } n = \ell, \\ \tau(n) & \text{otherwise.} \end{cases}$$
- A configuration is accepting if its state is t.
- A configuration is rejecting if its state is r.
- A configuration is halting if it is accepting or rejecting.

Definition

Let α and β be configurations of a DTM M.

- We write $\alpha \vdash_M \beta$ if β is a successor configuration of α.

Runs

Definition

Let α and β be configurations of a DTM M.

- We write $\alpha \vdash_M \beta$ if β is a successor configuration of α.
- For $n \geq 0$, we write $\alpha \vdash_M^n \beta$ if there exist configurations $\gamma_0, \gamma_1, \ldots, \gamma_n$ such that
  - $\gamma_0 = \alpha$,
  - $\gamma_n = \beta$, and
  - $\gamma_0 \vdash_M \gamma_1 \vdash_M \cdots \vdash_M \gamma_{n-1} \vdash_M \gamma_n$.

Runs

Definition

Let α and β be configurations of a DTM M.

- We write $\alpha \vdash_M \beta$ if β is a successor configuration of α.
- For $n \geq 0$, we write $\alpha \vdash_M^n \beta$ if there exist configurations $\gamma_0, \gamma_1, \ldots, \gamma_n$ such that
  - $\gamma_0 = \alpha$,
  - $\gamma_n = \beta$, and
  - $\gamma_0 \vdash_M \gamma_1 \vdash_M \cdots \vdash_M \gamma_{n-1} \vdash_M \gamma_n$.
- We write $\alpha \vdash_M^* \beta$ if there exists an $n \geq 0$ such that $\alpha \vdash_M^n \beta$.

Accepting, Rejecting, and Looping

Definition

Let w be an input for a DTM M and let α_w be the initial configuration of M on w.

- M accepts w if there exists an accepting configuration β such that $\alpha_w \vdash_M^* \beta$.

Accepting, Rejecting, and Looping

Definition

Let w be an input for a DTM M and let α_w be the initial configuration of M on w.

- M accepts w if there exists an accepting configuration β such that $\alpha_w \vdash_M^* \beta$.
- M rejects w if there exists a rejecting configuration β such that $\alpha_w \vdash_M^* \beta$.

Accepting, Rejecting, and Looping

Definition

Let w be an input for a DTM M and let α_w be the initial configuration of M on w.

- M accepts w if there exists an accepting configuration β such that $\alpha_w \vdash_M^* \beta$.
- M rejects w if there exists a rejecting configuration β such that $\alpha_w \vdash_M^* \beta$.
- Otherwise, M loops on the input w.

Accepting, Rejecting, and Looping

Definition

Let w be an input for a DTM M and let α_w be the initial configuration of M on w.

- M accepts w if there exists an accepting configuration β such that $\alpha_w \vdash_M^* \beta$.
- M rejects w if there exists a rejecting configuration β such that $\alpha_w \vdash_M^* \beta$.
- Otherwise, M loops on the input w.

We say M halts on input w if it accepts or rejects w (i.e., it doesn't loop).

Definition

- The language of a DTM M (say with input alphabet Σ), denoted by $L(M)$, is defined as

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}.$$

Classes of Languages

Definition

- The language of a DTM M (say with input alphabet Σ), denoted by $L(M)$, is defined as

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}.$$

- A halting DTM is a DTM that halts on every input (so either accepts or rejects it).

Definition

- The language of a DTM M (say with input alphabet Σ), denoted by $L(M)$, is defined as

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}.$$

- A halting DTM is a DTM that halts on every input (so either accepts or rejects it).
- A language L is computable, if there is a halting DTM M such that $L = L(M)$.

Definition

- The language of a DTM M (say with input alphabet Σ), denoted by $L(M)$, is defined as

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}.$$

- A halting DTM is a DTM that halts on every input (so either accepts or rejects it).
- A language L is computable, if there is a halting DTM M such that $L = L(M)$.
- A language L is computably-enumerable, if there is a DTM M such that $L = L(M)$.

Classes of Languages

Definition

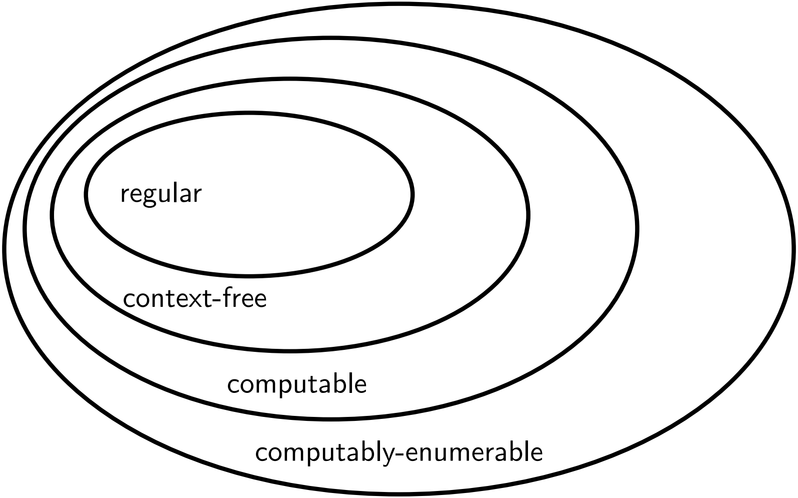
- The language of a DTM M (say with input alphabet Σ), denoted by $L(M)$, is defined as

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}.$$

- A halting DTM is a DTM that halts on every input (so either accepts or rejects it).
- A language L is computable, if there is a halting DTM M such that $L = L(M)$.
- A language L is computably-enumerable, if there is a DTM M such that $L = L(M)$.

  But: M might not terminate on all inputs $w \notin L(M)$!

Language Classes



Are all inclusions strict? Is there a language that is not computably-enumerable?

A Note on Terminology

Some authors use different terminology for the concepts we have introduced today:

- "decider" instead of "halting TM",
- "decidable" instead of "computable", and
- "semi-decidable" instead of "computably-enumerable".

Conclusion

We have seen

- Problems = Formal Languages
- DTM's as an abstract model of computation
- The difference between computable and computably-enumerable languages:
  - L is computably-enumerable $\Leftrightarrow$ there exists DTM M such that $L(M) = L$, i.e.,
    - $w \in L \Rightarrow M$ accepts w,
    - but $w \notin L \Rightarrow M$ rejects w or loops.
  - L is computable $\Leftrightarrow$ there exists a halting DTM M such that $L(M) = L$, i.e.,
    - $w \in L \Rightarrow M$ accepts w and
    - $w \notin L \Rightarrow M$ rejects w.

Conclusion

We have seen

- Problems = Formal Languages
- DTM's as an abstract model of computation
- The difference between computable and computably-enumerable languages:
  - L is computably-enumerable $\Leftrightarrow$ there exists DTM M such that $L(M) = L$, i.e.,
    - $w \in L \Rightarrow M$ accepts w,
    - but $w \notin L \Rightarrow M$ rejects w or loops.
  - L is computable $\Leftrightarrow$ there exists a halting DTM M such that $L(M) = L$, i.e.,
    - $w \in L \Rightarrow M$ accepts w and
    - $w \notin L \Rightarrow M$ rejects w.
- And. too. many. slides.

Reading

In “Computability and Complexity”:

- The Introduction and Agreements (pages xiii to xvii)
- Section 2.1 (pages 71 to 85)
- Also skim Section 1 (pages 1 to 21 suffice) to get used to the notation used in the book (and slides) and to recall what you have learned about finite automata on the 4th semester.

Finally, have a look at Turing's paper introducing what we call today Turing machines, keeping in mind that it was published 1936 before the advent of *nonhuman* computers:

`https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf`