Computational complexity

lecture 1

Introductory announcements

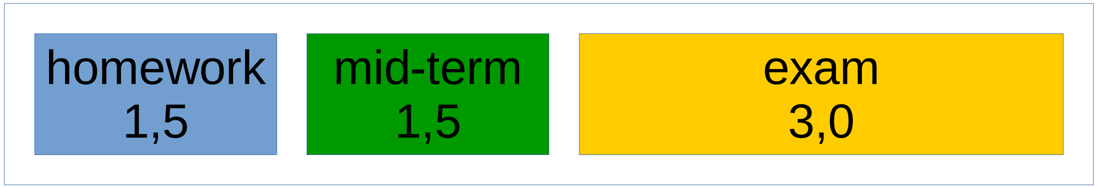
- Lectures will be in English
- Slides (and other materials) will be available on the webpage

- Book: S. Arora, B. Barak *Computational Complexity: A Modern Approach* (available online)

Introductory announcements

Grading rules:

- homeworks (3 series) → 1.5 pt
- mid-term → 1.5 pt
- everyone can write the June exam
  (no lower limit for homeworks + mid-term)
- final exam → 3 pt
  the final grade depends on: homeworks + mid-term + exam (max=6)

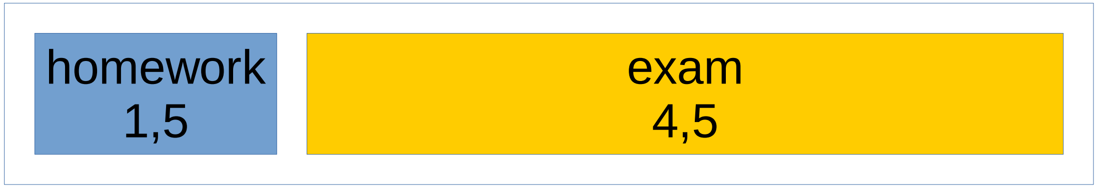


homework
1,5

mid-term
1,5

exam
3,0

- final exam, second try → 4.5 pt
  the final grade depends on: homeworks + exam (max = 6 pt)



homework
1,5

exam
4,5

- the exam consists of two parts: theory + „practical” problems

What is this course about?

- the subject of research: computational problems
  → the basic question: how fast a given problem can be solved?
  → what does it mean that a problem is difficult? (maybe we are not intelligent enough to solve it?)
- JAiO: there are decidable and undecidable problems
  → here: (decidable) problems can be easy or hard (in different senses)
- algorithmics: we have a fast algorithm
  → here: a fast algorithm does not exist

Plan of the course

- Deterministic Turing machines – basic facts
- Boolean circuits
- Nondeterministic Turing machines, reductions
- Probabilistic computations
- Fixed parameter tractability (FPT)
- Interactive proofs
- Alternating Turing machines
- Probabilistically checkable proofs (PCP)
- ...

Turing machine – a formal definition of a computation

- David Hilbert: Is there an algorithm deciding any mathematical hypothesis? („Entscheidungsproblem” - 1928)
- Alonzo Church (1936), Alan Turing (1937) – NO
- The answer required a formal definition
  (what does it mean „an algorithm”?)
  → a Turing machine (lambda-calculus - Church)

Turing machine – a formal definition of a computation

Components of a (multitape, deterministic) Turing machine:

- number of tapes k
- a finite working alphabet Γ, where $\triangleright,\perp\in\Gamma$
- an input alphabet $\Sigma\subseteq\Gamma\setminus\{\triangleright,\perp\}$
- a finite set of states Q
- states: initial, accepting, rejecting $q_I,q_A,q_R\in Q$
- a transition function $\delta{:}(Q\setminus\{q_A,q_R\})\times\Gamma^k\rightarrow Q\times\Gamma^k\times\{L,R,Z\}^k$ such that ...

Turing machine – a formal definition of a computation

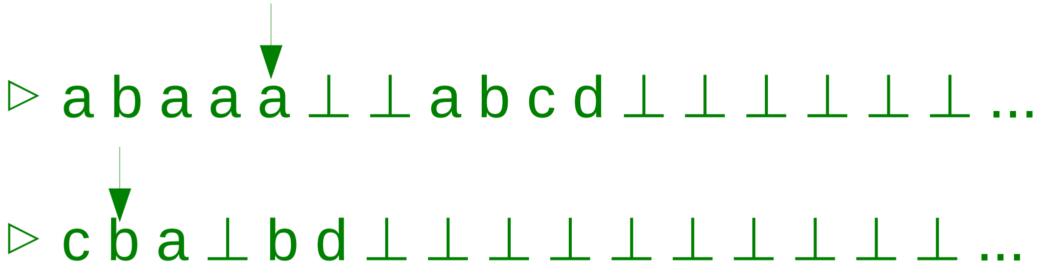
Components of a (multitape, deterministic) Turing machine:

- number of tapes k
- a finite working alphabet Γ, where $\triangleright,\perp\in\Gamma$
- an input alphabet $\Sigma\subseteq\Gamma\setminus\{\triangleright,\perp\}$
- a finite set of states Q
- states: initial, accepting, rejecting $q_I,q_A,q_R\in Q$
- a transition function $\delta:(Q\setminus\{q_A,q_R\})\times\Gamma^k\to Q\times\Gamma^k\times\{L,R,Z\}^k$ such that ...

A <u>configuration</u> of a Turing machine consists of:

- contents of k *tapes*; each of them is infinite to the right
- location of the *head* on every tape
- state

Example (2 tapes): q_5



$\triangleright$ a b a a a $\perp$ $\perp$ a b c d $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ ...

$\triangleright$ c b a $\perp$ b d $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ ...

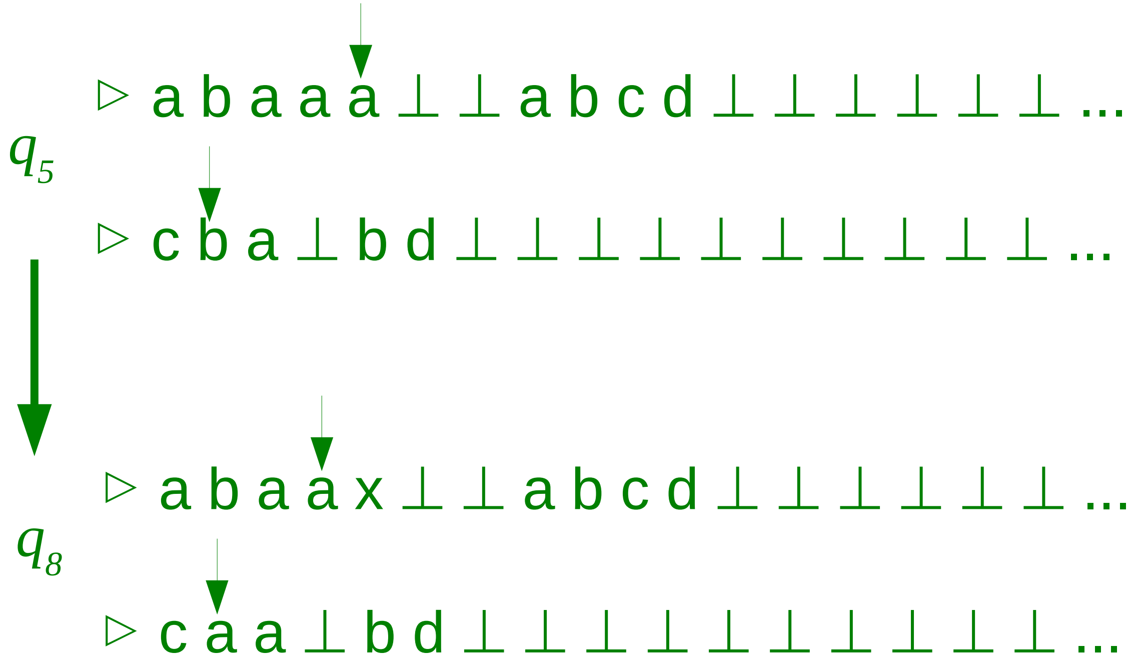
Turing machine – a formal definition of a computation

A computation: a function $\rightarrow_M$ between configurations

Example:

$\delta(q_5,a,b)=(q_8,x,a,L,Z)$ usual notation: $q_5,a,b \rightarrow q_8,x,a,L,Z$

- if the state is q_5, letters under the heads are a (on tape 1), b (on tape 2)
- then change the state to q_8
- write x on tape 1, a on tape 2
- move head 1 to the left, do not move head 2



$\triangleright$ a b a a $\overset{\downarrow}{\text{a}}$ $\perp$ $\perp$ a b c d $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ ...

q_5

$\triangleright$ c $\overset{\downarrow}{\text{b}}$ a $\perp$ b d $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ ...

$\big\downarrow$

$\triangleright$ a b a $\overset{\downarrow}{\text{a}}$ x $\perp$ $\perp$ a b c d $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ ...

q_8

$\triangleright$ c $\overset{\downarrow}{\text{a}}$ a $\perp$ b d $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ $\perp$ ...

Turing machine – a formal definition of a computation

Additional assumptions about a configuration:

- from some moment on every tape there are only $\perp$ symbols
- the first symbol on every tape is $\triangleright$

Additional assumptions about a transition function:

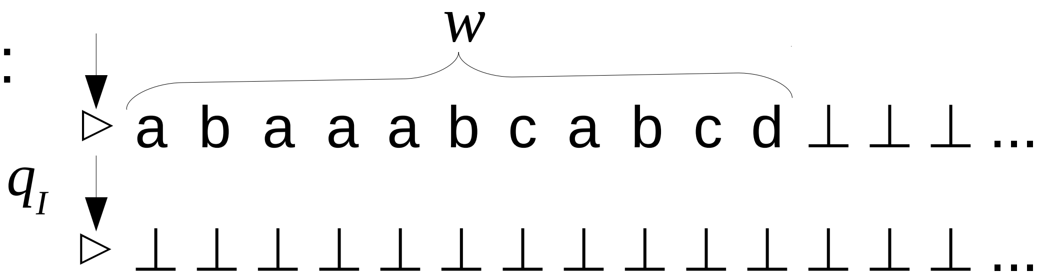
- the machine never replaces $\triangleright$ by any other symbol
- the machine never wants to go left when it sees $\triangleright$

(in particular, this ensures that every configuration has a successor, unless the state is q_A or q_R)

Turing machine – a formal definition of a computation

A <u>computation</u> on an input word $w \in \Sigma^*$:

- the initial configuration is:



w

q_I

▷ a b a a a b c a b c d ⊥ ⊥ ⊥ ...

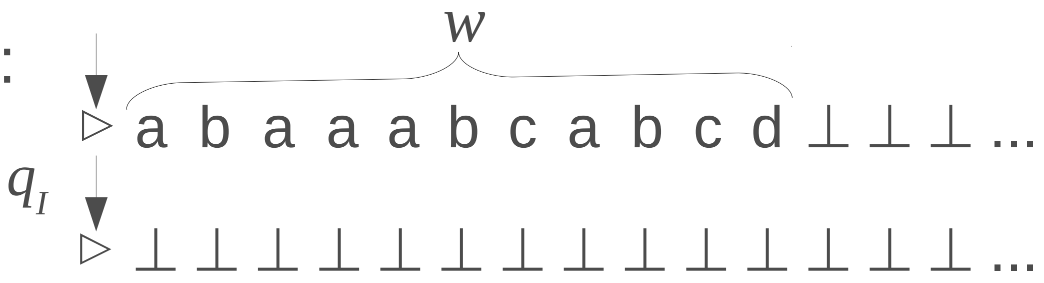
▷ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ...

- the machine *accepts* w, if it reaches a configuration with state q_A
- the machine *rejects* w, if it reaches a configuration with state q_R
- otherwise the computation is infinite (the machine *loops*)

Turing machine – a formal definition of a computation

A <u>computation</u> on an input word $w \in \Sigma^*$:

- the initial configuration is:



w

q_I

▷ a b a a a b c a b c d ⊥ ⊥ ⊥ ...

▷ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ⊥ ...

- the machine *accepts* w, if it reaches a configuration with state q_A
- the machine *rejects* w, if it reaches a configuration with state q_R
- otherwise the computation is infinite (the machine *loops*)

→ notation: $L(M)=\{w : M \text{ accepts } w\}$

→ M has the *halting property*, if it halts on every input

→ a language $L \subseteq \Sigma^*$ is *semidecidable* (or *recursively enumerable*) if there exists a machine that accepts exactly words from L (i.e., $L(M)=L$)

→ if this machine has the halting property, then L is *computable* (*decidable*)

Turing machine – a formal definition of a computation

- M has the *halting property*, if it halts on every input
- a language $L \subseteq \Sigma^*$ is *semidecidable* (or *recursively enumerable*) if there exists a machine that accepts exactly words from L (i.e., $L(M)=L$)
- if this machine has the halting property, then L is *computable* (*decidable*)
- Tutorials: if L and the complement of L are semidecidable, then they are decidable

Turing machine – a formal definition of a computation

- M has the *halting property*, if it halts on every input

- a language $L \subseteq \Sigma^*$ is *semidecidable* (or *recursively enumerable*) if there exists a machine that accepts exactly words from L (i.e., $L(M)=L$)
- if this machine has the halting property, then L is *computable* (*decidable*)

- Tutorials: if L and the complement of L are semidecidable, then they are decidable

<u>Computing functions:</u>

- a partial function $f:\Sigma^* \rightarrow \Sigma^*$ is computable, if there exists a machine M
- that accepts every word $w \in dom(f)$, ending in a configuration with $\triangleright f(w)\perp^{\infty}$ on the last tape,
- and rejects every word $w \notin dom(f)$

Variants of Turing machines

- a tape that is infinite in both directions
- multiple accepting / rejecting states
- a single tape only
- never writes $\perp$
- nondeterministic machines, alternating machines (the machines defined above were deterministic)
- ...

Variants of Turing machines

- a tape that is infinite in both directions
- multiple accepting / rejecting states
- a single tape only
- never writes $\perp$
- nondeterministic machines, alternating machines (the machines defined above were deterministic)
- ...

Fact: All variants recognize the same class of languages.
[i.e., the notion of a Turing machine is *robust*]

Remark: it is enough to prove that for every machine of type X there exists an equivalent machine of type Y. In practice these constructions are computable, but to obtain the above fact we do not need to know this.
Such a distinction often appears on this lecture: when it is enough that something exists, and when we have to know how to (quickly) compute it?

Time complexity

A machine M works in time $T(n)$ (for a function $T:\mathbb{N}\to\mathbb{N}$) if for every word $w\in\Sigma^*$ it halts after at most $T(|w|)$ steps.

(in particular it has the halting property)

A language $L\subseteq\Sigma^*$ is decidable in time $T(n)$ if there exists a (**multitape**) machine that recognizes this language and works in time $T(n)$.

We usually talk about the asymptotic behavior of the complexity, i.e., that the time is $O(T(n))$.

Time complexity

A machine M works in time $T(n)$ (for a function $T:\mathbb{N}\rightarrow\mathbb{N}$) if for every word $w\in\Sigma^*$ it halts after at most $T(|w|)$ steps.

(in particular it has the halting property)

A language $L\subseteq\Sigma^*$ is decidable in time $T(n)$ if there exists a (**multitape**) machine that recognizes this language and works in time $T(n)$.

We usually talk about the asymptotic behavior of the complexity, i.e., that the time is $O(T(n))$.

Tutorials: The language of palindromes is decidable in linear time, but on a single-tape machine it requires a quadratic time.

Time complexity

A machine M works in time $T(n)$ (for a function $T:\mathbb{N}\rightarrow\mathbb{N}$) if for every word $w\in\Sigma^*$ it halts after at most $T(|w|)$ steps.

(in particular it has the halting property)

A language $L\subseteq\Sigma^*$ is decidable in time $T(n)$ if there exists a (**multitape**) machine that recognizes this language and works in time $T(n)$.

We usually talk about the asymptotic behavior of the complexity, i.e., that the time is $O(T(n))$.

Tutorials: The language of palindromes is decidable in linear time, but on a single-tape machine it requires a quadratic time.

Theorem (linear speed-up):
If a language L is decidable in time $T(n)$, then for every constant $c>0$ it is also decidable in time $c\cdot T(n)+O(n)$.

Proof: on tutorials (the idea: one counts the number of steps, so it is enough to simulate multiple steps while performing a single step).

Space complexity

Space complexity: the number of tape cells used by the machine

Question: should we include the length of the input?

Space complexity

Space complexity: the number of tape cells used by the machine

Question: should we include the length of the input?

Yes? Then the complexity is always at least linear (and we cannot distinguish between a machine M_1 really using linear space, and a machine M_2 that only reads the input but beyond that works in constant space). ↯

Space complexity

Space complexity: the number of tape cells used by the machine

Question: should we include the length of the input?

Yes? Then the complexity is always at least linear (and we cannot distinguish between a machine M_1 really using linear space, and a machine M_2 that only reads the input but beyond that works in constant space). ↯

No? The same problem: maybe M_1 uses only the cells occupied originally by the input word (no additional memory), so again we cannot distinguish it from M_2. ↯

Space complexity

Space complexity: the number of tape cells used by the machine

Question: should we include the length of the input?

Yes? Then the complexity is always at least linear (and we cannot distinguish between a machine M_1 really using linear space, and a machine M_2 that only reads the input but beyond that works in constant space). ↯

No? The same problem: maybe M_1 uses only the cells occupied originally by the input word (no additional memory), so again we cannot distinguish it from M_2. ↯

Solution – allow only *off-line machines*: ☼

- the input tape is read-only (when I see $\perp$, I cannot move right)
- working tapes
- while computing functions: output tape, where the head cannot move left (i.e., write-only)

Space complexity

In space complexity we do not include the length of the input (important when space complexity is smaller than linear)

Formally, we allow only *off-line machines*:

- the input tape is read-only (when I see $\perp$, I cannot move right)
- working tapes
- while computing functions: output tape, where the head cannot move left (i.e., write-only)

A machine M works in space $S(n)$ (for a function $S:\mathbb{N}\to\mathbb{N}$) if for every word $w\in\Sigma^*$ visits at most $S(|w|)$ cells on its working tapes.

A language $L\subseteq\Sigma^*$ is *recognizable in space* $S(n)$ if there exists a multitape machine that halts on every input, accepts L, and works in space $S(n)$.

Usually we talk about space $O(S(n))$ (asymptotic behavior).

It is easy to reduce space usage „times a constant" - we remember a few cells in one.

It is possible to convert a multitape machine into a machine with one working tape, which works in the same space.

Machines vs. languages

- Sometimes we talk about time / space complexity of a language (there exists a machine such that ...)
- Sometimes we talk about working time / space of a particular machine (particular algorithm)

Languages vs. decision problems

Example: reachability in a graph

- Input: a set of nodes, a set of edges, two distinguished nodes.
  → The input is not a word, it is a more complicated object.
  → A Turing machine reads words.
- But – a graph can be written as a word:
  number_of_nodes,
  number_of_edges,
  a list of pair of nodes connected by edges (where we assume that nodes are numbered by consecutive natural number);
  particular numbers are separated by a special $ sign.
  → Multiple possible representations of a graph
  → It is easy to convert from one representation to another.
- Usually, we talk about complexity of a problem – which means “complexity of the corresponding language under some natural representation of inputs as words” (typically the complexity does not depend on the choice of the representation)
- Sometimes it depends, and then we should be more precise
  (we should say which representation of inputs is considered)

Languages vs. decision problems

- While considering a concrete problem we think about an algorithm understood in an abstract way, and usually we do not refer to a particular representation – but we are aware that it is possible to implement basic programming concepts (variables, loops, etc.) on a Turing machine
- While proving general theorems we consider Turing machines (a model that is simple, but strong enough).

Church-Turing thesis

Church-Turing thesis: every physically realizable computation device can be simulated by a Turing machine.

(this is not a mathematical theorem – it is not sure what can be physically realizable)

A stronger thesis: problems “easy” for other devices are also “easy” for Turing machines – every physically realizable computation device can be simulated by a Turing machine *with polynomial overhead*.

Random Access Machine (RAM)

[This is a side remark – RAM machines will not appear more during the lecture]

A model closer to computers than Turing machines:

- Cells contain arbitrarily large numbers (instead of letters from a finite alphabet)
- A program amounts to a list of instructions. Available instructions:
  $X[i] \leftarrow k$ (where i,j,k,m – constants written in a program)
  $X[i] \leftarrow X[j]+X[k]$
  $X[i] \leftarrow X[j]-X[k]$
  $X[i] \leftarrow X[X[j]]$
  $X[X[i]] \leftarrow X[j]$
  $\mathit{if}\ X[i]>0\ \mathit{then\ goto}\ m$
- Every operation is performed in constant time (by definition)
- There is no multiplication – it can be realized in time linear in the number of bits

Random Access Machine (RAM)

[This is a side remark – RAM machines will not appear more during the lecture]

A model closer to computers than Turing machines:

- Cells contain arbitrarily large numbers (instead of letters from a finite alphabet)
- A program amounts to a list of instructions. Available instructions:

  $X[i] \leftarrow k$ (where i,j,k,m – constants written in a program)

  $X[i] \leftarrow X[j]+X[k]$

  $X[i] \leftarrow X[j]-X[k]$

  $X[i] \leftarrow X[X[j]]$

  $X[X[i]] \leftarrow X[j]$

  $\mathit{if}\ X[i]>0\ \mathit{then\ goto}\ m$
- Every operation is performed in constant time (by definition)
- There is no multiplication – it can be realized in time linear in the number of bits
- Input (and output) in cells $X[1],\dots,X[n]$; additionally $X[0]=n$
- The size of the input is defined as the total number of bits

Random Access Machine (RAM)

- A computation of a Turing machine using time $T(n)$ can be simulated on RAM in time $O(T(n))$
- A computation of a RAM using time $T(n)$ can be simulated on a Turing machine in time $O(T(n)^3)$

Time complexity – basic classes

[Now we come back to Turing machines]

- DTIME($T(n)$) – languages recognizable in time $O(T(n))$
- P = $\bigcup_{k\in\mathbb{N}}$DTIME(n^k) – i.e., languages recognizable in time $p(n)$ for some polynomial p
- EXPTIME = $\bigcup_{k\in\mathbb{N}}$DTIME(2^{n^k})

Space complexity – basic classes

- DSPACE($S(n)$) – languages recognizable in space $O(S(n))$
- L = $\bigcup_{k\in\mathbb{N}}$DSPACE(log n^k) = DSPACE(log n)
- PSPACE = $\bigcup_{k\in\mathbb{N}}$DSPACE(n^k) – i.e., languages recognizable in space $p(n)$ for some polynomial p
- EXPSPACE = $\bigcup_{k\in\mathbb{N}}$DSPACE(2^{n^k})

Time vs space

DTIME($f(n)$) ⊆ DSPACE($f(n)$)

Proof: In time $f(n)$ a machine can visit at most $k \cdot f(n)$ cells (k = the number of tapes)

Time vs space

Conversely: $\mathrm{DSPACE}(f(n)) \subseteq \bigcup_{c>0} \mathrm{DTIME}(n\cdot c^{f(n)})$

if $f(n) \geq log(n)$, then simply: $\mathrm{DSPACE}(f(n)) \subseteq \bigcup_{c>0} \mathrm{DTIME}(c^{f(n)})$

Proof: Take some $L \in \mathrm{DSPACE}(f(n))$, recognized by M.
M does not loop, so
(the number of visited configurations) = (the number of steps)
(the number of all configurations) $\geq$ (the number of steps)

the number of all configuration equals:

$$|Q|\cdot(n+2)\cdot(4|\Gamma|)^{df(n)}$$

state

position on the input tape

contents of working tapes + a special marker for:

- the position of the head
- the last visited cell on the tape

Time vs space

$\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n)) \subseteq \bigcup_{c>0} \mathrm{DTIME}(n \cdot c^{f(n)})$

In particular:
$\mathrm{L} \subseteq \mathrm{P} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME} \subseteq \mathrm{EXPSPACE}$

Time vs space

$\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n)) \subseteq \bigcup_{c>0}\mathrm{DTIME}(n \cdot c^{f(n)})$

In particular:
L ⊆ P ⊆ PSPACE ⊆ EXPTIME ⊆ EXPSPACE

Are these classes different?

it is **NOT** known whether:

- L≠P
- P≠PSPACE
- PSPACE≠EXPTIME
- EXPTIME≠EXPSPACE

It is known (and we will prove this soon), that

- L≠PSPACE≠EXPSPACE
- P≠EXPTIME