COMP3630 / COMP6363

week 7: **Reductions and Complexities**

Not based on the book, but you may read from chapter 10

slides created by: Mostly by Pascal Bercher, some by Dirk Pattinson

convenor & lecturer: Pascal Bercher

The Australian National University

Semester 1, 2025

Content of this Chapter

- Motivation
- Non-Deterministic Turing Machines
- Big-$\mathcal{O}$ Notation
- (Most) Complexity Classes (covered in this course)
- Reductions
- Membership, Hardness, Completeness

Motivation

Motivation

- So far, we only focused on expressivity, i.e., which problems (i.e., languages) could be expressed at all using different kinds of automata, grammars, and TMs.
- Now, we want to know "how quickly" problems can be decided.

Motivation

- So far, we only focused on expressivity, i.e., which problems (i.e., languages) could be expressed at all using different kinds of automata, grammars, and TMs.
- Now, we want to know "how quickly" problems can be decided.
- You should also be able to understand the difference between solving and deciding a problem. (If by the en of week 11 you still don't, ask!!)
- We investigate the computational hardness of decision problems:

Motivation

- So far, we only focused on expressivity, i.e., which problems (i.e., languages) could be expressed at all using different kinds of automata, grammars, and TMs.
- Now, we want to know "how quickly" problems can be decided.
- You should also be able to understand the difference between solving and deciding a problem. (If by the en of week 11 you still don't, ask!!)
- We investigate the computational hardness of decision problems:
  - What's the "performance" of the best-known algorithm for deciding the problem?
  - Which problems are equally hard? Which ones are harder than others?
  - We look at how problems can be "turned into each other" (as before!).

Motivation

- So far, we only focused on expressivity, i.e., which problems (i.e., languages) could be expressed at all using different kinds of automata, grammars, and TMs.
- Now, we want to know "how quickly" problems can be decided.
- You should also be able to understand the difference between solving and deciding a problem. (If by the en of week 11 you still don't, ask!!)
- We investigate the computational hardness of decision problems:
  - What's the "performance" of the best-known algorithm for deciding the problem?
  - Which problems are equally hard? Which ones are harder than others?
  - We look at how problems can be "turned into each other" (as before!).
- We measure "performance" in terms of a Turing Machine's:
  - Time requirement (number of operations/transitions)
  - Space requirement (number of cells that can be read/written)

Why should we care for Problem Complexities?

Given a new problem to solve, we:

- ... can use existing solvers instead of designing new ones,
  - $\rightarrow$ Which software do you think is better? The one you design from scratch in a few weeks, or one that entire research communities (few or dozens to thousands of PhD students, post-docs, Professors) created over decades?

Why should we care for Problem Complexities?

Given a new problem to solve, we:

- . . . can use existing solvers instead of designing new ones,
  - → Which software do you think is better? The one you design from scratch in a few weeks, or one that entire research communities (few or dozens to thousands of PhD students, post-docs, Professors) created over decades?
- . . . know performance bounds for our yet-to-be-designed solver
  - → No need to look for an "efficient" algorithm if not a single genius so far was able to do that! (It makes a good excuse!)
    However, one will be the first, and that could be you!

Why should we care for Problem Complexities?

Given a new problem to solve, we:

- ... can use existing solvers instead of designing new ones,
  - → Which software do you think is better? The one you design from scratch in a few weeks, or one that entire research communities (few or dozens to thousands of PhD students, post-docs, Professors) created over decades?
- ... know performance bounds for our yet-to-be-designed solver
  - → No need to look for an "efficient" algorithm if not a single genius so far was able to do that! (It makes a good excuse!)
    However, one will be the first, and that could be you!
- ... understand the problems we solve much better.
  - → If you know that your (new) problem is equivalent to an existing (established) one, that surely helps... (Imagine, you take a course twice! The second time it's much easier...)

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime?

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Use a non-deterministic TM. Guess a valuation, then verify.
     What about Backtracking?

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Use a non-deterministic TM. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_4) \land (x_2 \lor \neg x_3) \land (\neg x_2 \lor x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Use a non-deterministic TM. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!
     Runtime?

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_4) \land (x_2 \lor \neg x_3) \land (\neg x_2 \lor x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Use a non-deterministic TM. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!
     Runtime? Polytime for the guessing, then polytime for the verification.

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Use a non-deterministic TM. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!
     Runtime? Polytime for the guessing, then polytime for the verification.
- So, what is the best runtime for such problems?
  - The first "runtime class" will be called **EXPTIME** (membership)
  - This also takes only polynomial space (if enumeration is implemented in-place), so we are also in **PSPACE**,
  - The second algorithm shows **NP** membership (**N** is for **n** on-deterministic)

  $\rightarrow$ SAT is in ... (as you will see)

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Use a non-deterministic TM. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!
     Runtime? Polytime for the guessing, then polytime for the verification.
- So, what is the best runtime for such problems?
  - The first "runtime class" will be called **EXPTIME** (membership)
  - This also takes only polynomial space (if enumeration is implemented in-place), so we are also in **PSPACE**,
  - The second algorithm shows **NP** membership (**N** is for **n** on-deterministic)

  $\rightarrow$ SAT is in **PSPACE**, **EXPTIME** and in **NP**, but **NP** is lower! (as you will see)

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Use a non-deterministic TM. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!
     Runtime? Polytime for the guessing, then polytime for the verification.
- So, what is the best runtime for such problems?
  - The first "runtime class" will be called **EXPTIME** (membership)
  - This also takes only polynomial space (if enumeration is implemented in-place), so we are also in **PSPACE**,
  - The second algorithm shows **NP** membership (**N** is for **n** on-deterministic)

  → SAT is in **PSPACE**, **EXPTIME** and in **NP**, but **NP** is lower! (as you will see)

Example (for the Relevance of this)

Boolean Satisfiability (SAT)

- Let ϕ be a boolean formula with n variables, e.g.,:
  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4)$ with $n = 4$.
- Can you find a valuation (assignment to the x_i values) that makes it true?
- Which runtimes does your algorithm have?
- Two approaches:
  1. Enumerate all possible choices, e.g., using truth tables.
     Runtime? 2^n, i.e., exponential! Table doubles with each further variable
  2. Use a non-deterministic TM. Guess a valuation, then verify.
     What about Backtracking? Not required! Non-Determinism is always right!
     Runtime? Polytime for the guessing, then polytime for the verification.
- So, what is the best runtime for such problems?
  - The first "runtime class" will be called **EXPTIME** (membership)
  - This also takes only polynomial space (if enumeration is implemented in-place), so we are also in **PSPACE**,
  - The second algorithm shows **NP** membership (**N** is for **n** on-deterministic)

  $\rightarrow$ SAT is in **PSPACE**, **EXPTIME** and in **NP**, but **NP** is lower! (as you will see)
- But is the problem also in **P**? (I.e., can we **P**-decide it without guessing?)

Non-Deterministic Turing Machines

Deterministic Turing Machines, Recap

A Turing Machine has the form $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where

- Q: finite set of states
- Σ: finite set of input symbols
- Γ: finite set of tape symbols such that $\Sigma \subseteq \Gamma$
- δ is a (partial) transition function

$$\begin{array}{rcl} \delta \ : \ Q \times \Gamma & \rightharpoonup & Q \times \Gamma \times \{L, R\} \\ (\text{state}, \text{tape symbol}) & \mapsto & (\text{new state}, \text{new tape symbol}, \text{direction}) \end{array}$$

  The direction tells the read/write head which way to go next: Left or Right
- q_0: initial state of the TM
- $B \in \Gamma \setminus B$: is the blank symbol. All but a finite number of tape symbols are Bs
- $F \subseteq Q$: the set of final/accepting states of the TM

Non-Deterministic Turing Machines

A Turing Machine has the form $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where

- Q: finite set of states
- Σ: finite set of input symbols
- Γ: finite set of tape symbols such that $\Sigma \subseteq \Gamma$
- δ is a (partial) transition function

$$\begin{array}{rcl} \delta \; : \; Q \times \Gamma & \rightharpoonup & 2^{Q \times \Gamma \times \{L,R\}} \\ (\text{state}, \text{tape symbol}) & \mapsto & \text{sets of } (\text{new state}, \text{new tape symbol}, \text{direction}) \end{array}$$

  The direction tells the read/write head which way to go next: Left or Right
- q_0: initial state of the TM
- $B \in \Gamma \setminus B$: is the blank symbol. All but a finite number of tape symbols are Bs
- $F \subseteq Q$: the set of final/accepting states of the TM

Fundamental Properties of Non-Det. TMs

Basic Properties/Definitions

- Acceptance:
  - Recap: A det. TM accepts a string w iff $q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ for some $p \in F$.
  - For non-det. TMs that's the same, but now there is branching

Fundamental Properties of Non-Det. TMs

Basic Properties/Definitions

- Acceptance:
  - Recap: A det. TM accepts a string w iff $q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ for some $p \in F$.
  - For non-det. TMs that's the same, but now there is branching
- Rejection: Also the same. A word is rejected iff it is not accepted.
  (I.e., again the same as for deterministic TMs!)

Fundamental Properties of Non-Det. TMs

Basic Properties/Definitions

- Acceptance:
  - Recap: A det. TM accepts a string w iff $q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ for some $p \in F$.
  - For non-det. TMs that's the same, but now there is branching
- Rejection: Also the same. A word is rejected iff it is not accepted. (I.e., again the same as for deterministic TMs!)
- Halting: The same as for det. TMs, but for all possible traces.

Language Definitions.

- The language of a non-deterministic Turing Machine is the words accepted by it. (Just like for deterministic TMs.)
- Note how this now implies search: there might be many computation traces for an input; maybe just one accepts!

Fundamental Properties of Non-Det. TMs

Basic Properties/Definitions

- Acceptance:
  - Recap: A det. TM accepts a string w iff $q_0 w \overset{*}{\underset{M}{\vdash}} \alpha p \beta$ for some $p \in F$.
  - For non-det. TMs that's the same, but now there is branching
- Rejection: Also the same. A word is rejected iff it is not accepted. (I.e., again the same as for deterministic TMs!)
- Halting: The same as for det. TMs, but for all possible traces.

Language Definitions.

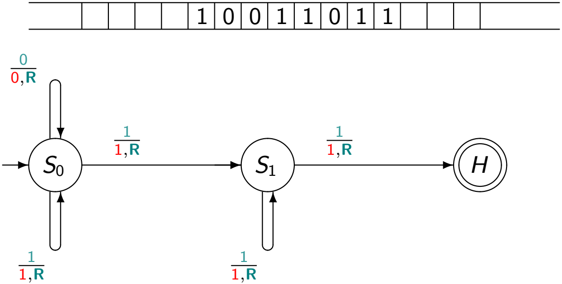
- The language of a non-deterministic Turing Machine is the words accepted by it. (Just like for deterministic TMs.)
- Note how this now implies search: there might be many computation traces for an input; maybe just one accepts!

Relationship to Deterministic TMs.

- Non-det. TMs can't do more than deterministic ones. (We proved that!)
- Non-det. TMs could be quicker than deterministic ones. (Unknown!)

Example (for a Non-Det. TM)

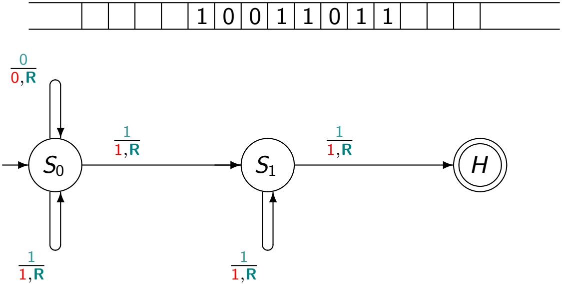
Consider the following TM, defined over an initial string over $\Sigma = \{0, 1\}$:



- What language does it accept?

Example (for a Non-Det. TM)

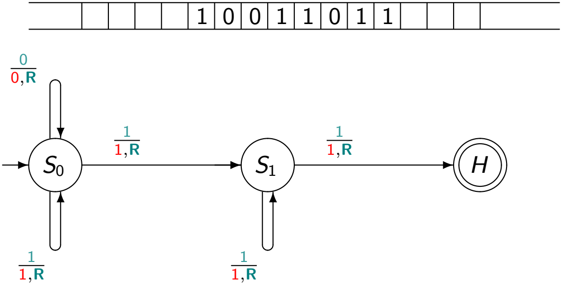
Consider the following TM, defined over an initial string over $\Sigma = \{0, 1\}$:



- What language does it accept?

Example (for a Non-Det. TM)

Consider the following TM, defined over an initial string over $\Sigma = \{0, 1\}$:



- What language does it accept?

$$\{w \mid w \text{ contains} \geq 2 \text{ consecutive 1s}\}$$
$$= \{w11w' \mid w, w' \in \Sigma^*\}$$

Big-$\mathcal{O}$ Notation

Introduction

- In the following we will define complexity classes based on whether some TM with specific properties exists.
- Example SAT:
  There is a non-det. TM that runs in "polynomial time" – what does that mean?
- We formalize this using the Big-$\mathcal{O}$ notation.

Poll. Who of you knows the big-$\mathcal{O}$ notation already?

Example

Let's decide $L = \{0^i 1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form $0^i 1^i$ for some i. M does:

- Scan word w and reject if 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

Example

Let's decide $L = \{0^i1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form 0^i1^i for some i. M does:

- Scan word w and reject if 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

Questions:

1. How much "time" does M need, as a function f of w's length?

Example

Let's decide $L = \{0^i 1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form $0^i 1^i$ for some i. M does:

- Scan word w and reject if 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

Questions:

1. How much "time" does M need, as a function f of w's length?
   f adheres $f(2k) = f(2(k-1)) + 4k + 1$, which is in $\mathcal{O}(n^2)$.

w	ϵ	01	$0^2 1^2$	$0^3 1^3$	$0^4 1^4$	$0^5 1^5$
$f(\|w\|)$	2	8	19	34	53	76

(exact numbers depend on implementation details)

Example

Let's decide $L = \{0^i1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form 0^i1^i for some i. M does:

- Scan word w and reject if 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

Questions:

1. How much "time" does M need, as a function f of w's length?
   f adheres $f(2k) = f(2(k-1)) + 4k + 1$, which is in $\mathcal{O}(n^2)$.

w	ϵ	01	0^21^2	0^31^3	0^41^4	0^51^5
$f(\|w\|)$	2	8	19	34	53	76

   (exact numbers depend on implementation details)

2. How much "space" does M need?

Example

Let's decide $L = \{0^i 1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form $0^i 1^i$ for some i. M does:

- Scan word w and reject if 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

Questions:

1. How much "time" does M need, as a function f of w's length?
   f adheres $f(2k) = f(2(k-1)) + 4k + 1$, which is in $\mathcal{O}(n^2)$.

w	ϵ	01	$0^2 1^2$	$0^3 1^3$	$0^4 1^4$	$0^5 1^5$
$f(\|w\|)$	2	8	19	34	53	76

   (exact numbers depend on implementation details)

2. How much "space" does M need? $\mathcal{O}(n)$

Example

Let's decide $L = \{0^i 1^i \mid i \in \mathbb{N}\}$ by TM M, i.e., check whether an arbitrary input has the form $0^i 1^i$ for some i. M does:

- Scan word w and reject if 10 is found.
- Repeat as long as there are 0s and 1s on the tape:
  Replace both the leftmost 0 and the rightmost 1 with blanks.
- If either only 0s or 1s are left: reject, otherwise accept.

Questions:

1. How much "time" does M need, as a function f of w's length?
   f adheres $f(2k) = f(2(k-1)) + 4k + 1$, which is in $\mathcal{O}(n^2)$.

w	ϵ	01	$0^2 1^2$	$0^3 1^3$	$0^4 1^4$	$0^5 1^5$
$f(\|w\|)$	2	8	19	34	53	76

   (exact numbers depend on implementation details)

2. How much "space" does M need? $\mathcal{O}(n)$

So, in total, M has polynomial time and space restriction!

Time Complexity – Abstraction

Problem. Exact "number of steps function" usually very complicated

- for example, $2n^{17} + 23n^2 - 5$
- and hard to find in the first place (see last slide!).

Solution. Consider approximate number of steps

- focus on asymptotic behavior
- as we are only interested in large problems

Idea. Abstract details away by just focussing on upper bounds
E.g., $f(n) = 2n^{17} + 23n^2 - 5 \in \mathcal{O}(n^{17})$

Time Complexity – Abstraction

Problem. Exact "number of steps function" usually very complicated

- for example, $2n^{17} + 23n^2 - 5$
- and hard to find in the first place (see last slide!).

Solution. Consider approximate number of steps

- focus on asymptotic behavior
- as we are only interested in large problems

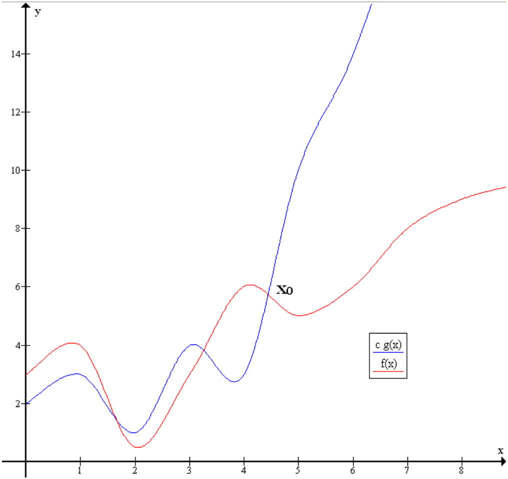
Idea. Abstract details away by just focussing on upper bounds
E.g., $f(n) = 2n^{17} + 23n^2 - 5 \in \mathcal{O}(n^{17})$

Big-$\mathcal{O}$ Notation. for f and g functions on natural numbers

- $f \in \mathcal{O}(g)$ if $\exists c.\ \exists n_0.\ \forall n \ \geq n_0.\ f(n) \leq c \cdot g(n)$
- "for large n, g is an upper bound to f up to a constant."
- E.g,. $f(n) \in \mathcal{O}(n^{17})$, since $g(n) = n^{17}$ and we can choose $c = 3$ so that we have $3n^{17} \geq f(n)$ for all $n \geq n_0$ (for a suitable n_0)

Graphical Illustration

Recall: $f \in \mathcal{O}(g)$ if $\exists c.\ \exists n_0.\ \forall n \geq n_0.\ f(n) \leq c \cdot g(n)$



Here, $f(x) \in \mathcal{O}(g(x))$ since $g(x)$ is at least as high as $f(x)$ for all $x \geq x_0$

Examples

Examples.

- Polynomials: leading exponent dominates
- e.g. "x^n + lower powers of x" $\in \mathcal{O}(x^n)$

- Exponentials: dominate polynomials
- e.g. "2^n + polynomial" $\in \mathcal{O}(2^n)$

Examples

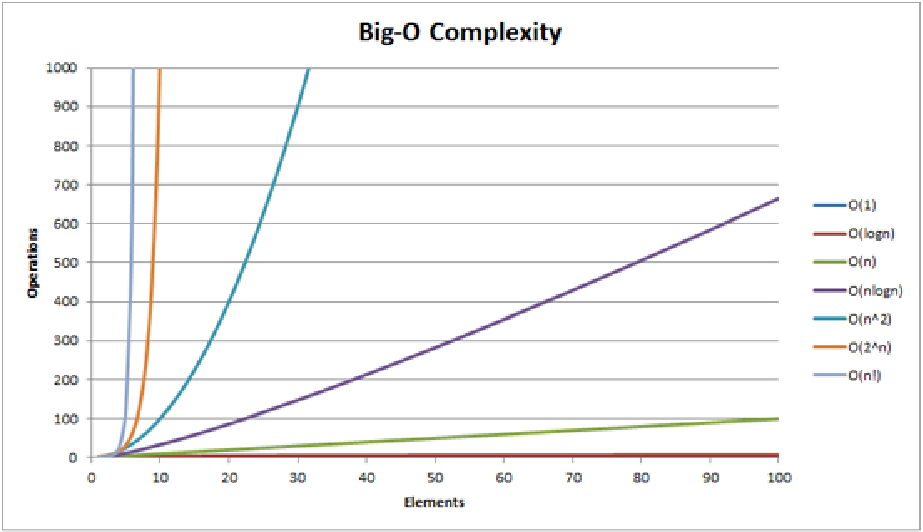
Examples.

- Polynomials: leading exponent dominates
- e.g. "x^n + lower powers of x" $\in \mathcal{O}(x^n)$
- Exponentials: dominate polynomials
- e.g. "$2^n +$ polynomial" $\in \mathcal{O}(2^n)$

Important Special Cases.

- <u>linear</u>. f is linear if $f \in \mathcal{O}(n)$
- <u>polynomial</u>. f is polynomial if $f \in \mathcal{O}(n^k)$, for some k
- <u>exponential</u>. f is exponential if $f \in \mathcal{O}(2^n)$
- Note: $f \in \mathcal{O}(2^{(n^k)})$ for $k > 1$ is technically not called an exponential, but later used in our definitions of **EXPTIME**.

Important Special Cases, Graphically



(Image copyright Lauren Kroner)

$\log$ in the Big-$\mathcal{O}$ Notation

We may safely omit the base of logarithms in the big-$\mathcal{O}$ notation because

$$\log_a n = \frac{\log_b n}{\log_b a} = \left(\frac{1}{\log_b a}\right) \cdot \log_b n \ .$$

So, two log functions with different bases just differ in a constant multiplier that does not depend on the input. However, we usually don't care for logs anyway, since we will only care whether something is an exponential or polynomial. (And a log doesn't make a difference, there.)

Complexity Classes

Complexity Classes

Recap: The "runtime" of a TM that's started on word w is its (maximal) number of computation steps or cells visited, expressed as a function of $|w|$.

Complexity Classes

Recap: The "runtime" of a TM that's started on word w is its (maximal) number of computation steps or cells visited, expressed as a function of $|w|$.

First some auxiliary definitions:

- $\underline{\mathbf{D}}\mathbf{TIME}(t(n)) = \{L \mid \underline{\text{d}}\text{et. TM decides } L \text{ in } \mathcal{O}(t(n)) \text{ time }\}$
- $\underline{\mathbf{N}}\mathbf{TIME}(t(n)) = \{L \mid \underline{\text{n}}\text{on-det. TM decides } L \text{ in } \mathcal{O}(t(n)) \text{ time }\}$

Complexity Classes

Recap: The "runtime" of a TM that's started on word w is its (maximal) number of computation steps or cells visited, expressed as a function of $|w|$.

First some auxiliary definitions:

- **DTIME**$(t(n)) = \{L \mid$ det. TM decides L in $\mathcal{O}(t(n))$ time $\}$
- **NTIME**$(t(n)) = \{L \mid$ non-det. TM decides L in $\mathcal{O}(t(n))$ time $\}$
- **DSPACE**$(t(n)) = \{L \mid$ det. TM decides L with $\mathcal{O}(t(n))$ space $\}$
- **NSPACE**$(t(n)) = \{L \mid$ non-det. TM decides L with $\mathcal{O}(t(n))$ space $\}$

Complexity Classes

Recap: The "runtime" of a TM that's started on word w is its (maximal) number of computation steps or cells visited, expressed as a function of $|w|$.

First some auxiliary definitions:

- $\mathbf{\underline{D}TIME}(t(n)) = \{L \mid \underline{\text{d}}\text{et. TM decides } L \text{ in } \mathcal{O}(t(n)) \text{ time }\}$
- $\mathbf{\underline{N}TIME}(t(n)) = \{L \mid \underline{\text{n}}\text{on-det. TM decides } L \text{ in } \mathcal{O}(t(n)) \text{ time }\}$
- $\mathbf{\underline{D}SPACE}(t(n)) = \{L \mid \underline{\text{d}}\text{et. TM decides } L \text{ with } \mathcal{O}(t(n)) \text{ space }\}$
- $\mathbf{\underline{N}SPACE}(t(n)) = \{L \mid \underline{\text{n}}\text{on-det. TM decides } L \text{ with } \mathcal{O}(t(n)) \text{ space }\}$

Now we can define some basic complexity classes:

- $\mathbf{P} = \bigcup_{k\in\mathbb{N}} \mathbf{DTIME}(n^k)$ $\qquad$ $\mathbf{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathbf{DSPACE}(n^k)$

Complexity Classes

Recap: The "runtime" of a TM that's started on word w is its (maximal) number of computation steps or cells visited, expressed as a function of $|w|$.

First some auxiliary definitions:

- $\mathbf{DTIME}(t(n)) =$ $\{L \mid$ det. TM decides L in $\mathcal{O}(t(n))$ time $\}$
- $\mathbf{NTIME}(t(n)) =$ $\{L \mid$ non-det. TM decides L in $\mathcal{O}(t(n))$ time $\}$
- $\mathbf{DSPACE}(t(n)) =$ $\{L \mid$ det. TM decides L with $\mathcal{O}(t(n))$ space $\}$
- $\mathbf{NSPACE}(t(n)) =$ $\{L \mid$ non-det. TM decides L with $\mathcal{O}(t(n))$ space $\}$

Now we can define some basic complexity classes:

- $\mathbf{P} = \bigcup_{k \in \mathbb{N}} \mathbf{DTIME}(n^k)$ $\quad$ $\mathbf{PSPACE} = \bigcup_{k \in \mathbb{N}} \mathbf{DSPACE}(n^k)$
- $\mathbf{NP} = \bigcup_{k \in \mathbb{N}} \mathbf{NTIME}(n^k)$ $\quad$ $\mathbf{NPSPACE} = \bigcup_{k \in \mathbb{N}} \mathbf{NSPACE}(n^k)$

Complexity Classes

Recap: The "runtime" of a TM that's started on word w is its (maximal) number of computation steps or cells visited, expressed as a function of $|w|$.

First some auxiliary definitions:

- $\textbf{DTIME}(t(n)) =$ $\{L \mid$ det. TM decides L in $\mathcal{O}(t(n))$ time $\}$
- $\textbf{NTIME}(t(n)) =$ $\{L \mid$ non-det. TM decides L in $\mathcal{O}(t(n))$ time $\}$
- $\textbf{DSPACE}(t(n)) =$ $\{L \mid$ det. TM decides L with $\mathcal{O}(t(n))$ space $\}$
- $\textbf{NSPACE}(t(n)) =$ $\{L \mid$ non-det. TM decides L with $\mathcal{O}(t(n))$ space $\}$

Now we can define some basic complexity classes:

- $\textbf{P} = \bigcup_{k\in\mathbb{N}} \textbf{DTIME}(n^k)$ $\quad$ $\textbf{PSPACE} = \bigcup_{k\in\mathbb{N}} \textbf{DSPACE}(n^k)$
- $\textbf{NP} = \bigcup_{k\in\mathbb{N}} \textbf{NTIME}(n^k)$ $\quad$ $\textbf{NPSPACE} = \bigcup_{k\in\mathbb{N}} \textbf{NSPACE}(n^k)$
- $\textbf{EXPTIME} = \bigcup_{k\in\mathbb{N}} \textbf{DTIME}(2^{(n^k)})$ $\quad$ $\textbf{EXPSPACE} = \bigcup_{k\in\mathbb{N}} \textbf{DSPACE}(2^{(n^k)})$

Complexity Classes

Recap: The "runtime" of a TM that's started on word w is its (maximal) number of computation steps or cells visited, expressed as a function of $|w|$.

First some auxiliary definitions:

- $\textbf{DTIME}(t(n)) =$ $\{L \mid$ det. TM decides L in $\mathcal{O}(t(n))$ time $\}$
- $\textbf{NTIME}(t(n)) =$ $\{L \mid$ non-det. TM decides L in $\mathcal{O}(t(n))$ time $\}$
- $\textbf{DSPACE}(t(n)) =$ $\{L \mid$ det. TM decides L with $\mathcal{O}(t(n))$ space $\}$
- $\textbf{NSPACE}(t(n)) =$ $\{L \mid$ non-det. TM decides L with $\mathcal{O}(t(n))$ space $\}$

Now we can define some basic complexity classes:

- $\textbf{P} = \bigcup_{k\in\mathbb{N}} \textbf{DTIME}(n^k)$ $\textbf{PSPACE} = \bigcup_{k\in\mathbb{N}} \textbf{DSPACE}(n^k)$
- $\textbf{NP} = \bigcup_{k\in\mathbb{N}} \textbf{NTIME}(n^k)$ $\textbf{NPSPACE} = \bigcup_{k\in\mathbb{N}} \textbf{NSPACE}(n^k)$
- $\textbf{EXPTIME} = \bigcup_{k\in\mathbb{N}} \textbf{DTIME}(2^{(n^k)})$ $\textbf{EXPSPACE} = \bigcup_{k\in\mathbb{N}} \textbf{DSPACE}(2^{(n^k)})$
- $\textbf{NEXPTIME} = \bigcup_{k\in\mathbb{N}} \textbf{NTIME}(2^{(n^k)})$ $\textbf{NEXPSPACE} = \bigcup_{k\in\mathbb{N}} \textbf{NSPACE}(2^{(n^k)})$

Over the next weeks, we see different problems from these classes and how they relate.

Relationships among Complexity Classes

How do time and space relate?

- On which TM can you solve harder/more problems?
  Using n movements or using n cells?

Relationships among Complexity Classes

How do time and space relate?

- On which TM can you solve harder/more problems?
  Using n movements or using n cells?
- In order to "use" a cell, we need to have a transition towards it.
- Thus, each space class can potentially contain more problems than their corresponding time class:

Relationships among Complexity Classes

How do time and space relate?

- On which TM can you solve harder/more problems?
  Using n movements or using n cells?
- In order to "use" a cell, we need to have a transition towards it.
- Thus, each space class can potentially contain more problems than their corresponding time class:
  - $\mathbf{P} \subseteq \mathbf{PSPACE}$ and $\mathbf{NP} \subseteq \mathbf{NPSPACE}$
  - $\mathbf{EXPTIME} \subseteq \mathbf{EXPSPACE}$ and $\mathbf{NEXPTIME} \subseteq \mathbf{NEXPSPACE}$

How do deterministic and non-deterministic classes relate?

Relationships among Complexity Classes

How do time and space relate?

- On which TM can you solve harder/more problems?
  Using *n* movements or using *n* cells?
- In order to "use" a cell, we need to have a transition towards it.
- Thus, each space class can potentially contain more problems than their corresponding time class:
  - **P** $\subseteq$ **PSPACE** and **NP** $\subseteq$ **NPSPACE**
  - **EXPTIME** $\subseteq$ **EXPSPACE** and **NEXPTIME** $\subseteq$ **NEXPSPACE**

How do deterministic and non-deterministic classes relate?

- Deterministic TMs are a special case of non-deterministic TMs.
- Thus, non-deterministic classes can potentially contain more problems than their corresponding deterministic classes:

Relationships among Complexity Classes

How do time and space relate?

- On which TM can you solve harder/more problems? Using n movements or using n cells?
- In order to "use" a cell, we need to have a transition towards it.
- Thus, each space class can potentially contain more problems than their corresponding time class:
  - **P** $\subseteq$ **PSPACE** and **NP** $\subseteq$ **NPSPACE**
  - **EXPTIME** $\subseteq$ **EXPSPACE** and **NEXPTIME** $\subseteq$ **NEXPSPACE**

How do deterministic and non-deterministic classes relate?

- Deterministic TMs are a special case of non-deterministic TMs.
- Thus, non-deterministic classes can potentially contain more problems than their corresponding deterministic classes:
  - **P** $\subseteq$ **NP** and **EXPTIME** $\subseteq$ **NEXPTIME**
  - **PSPACE** $\subseteq$ **NPSPACE** and **EXPSPACE** $\subseteq$ **NEXPSPACE**

Relationships among Complexity Classes: Heads up

What's also known to the literature:

- **PSPACE** $=$ **NPSPACE** and **EXPSPACE** $=$ **NEXPSPACE** (Savitch's Theorem, 1970 – see week 9)
- **P** $\subsetneq$ **EXPTIME** (by time Hierarchy Theorem(s), 1972, 1978, 1983)

Relationships among Complexity Classes: Heads up

What's also known to the literature:

- **PSPACE** = **NPSPACE** and **EXPSPACE** = **NEXPSPACE**
  (Savitch's Theorem, 1970 – see week 9)
- **P** $\subsetneq$ **EXPTIME**
  (by time Hierarchy Theorem(s), 1972, 1978, 1983)

So, in total, we will get: (Check yourself what we did not prove yet.)

- $\underbrace{\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{(N)PSPACE} \subseteq \mathbf{EXPTIME}}_{\neq} \subseteq \mathbf{NEXPTIME} \subseteq \mathbf{(N)EXPSPACE}$

Note how every problem in a class "further left" is also in a class "further right".

Relationships among Complexity Classes: Heads up

What's also known to the literature:

- **PSPACE** = **NPSPACE** and **EXPSPACE** = **NEXPSPACE** (Savitch's Theorem, 1970 – see week 9)
- **P** $\subsetneq$ **EXPTIME** (by time Hierarchy Theorem(s), 1972, 1978, 1983)

So, in total, we will get: (Check yourself what we did not prove yet.)

- $\underbrace{\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{(N)PSPACE} \subseteq \mathbf{EXPTIME}}_{\neq} \subseteq \mathbf{NEXPTIME} \subseteq \mathbf{(N)EXPSPACE}$

Note how every problem in a class "further left" is also in a class "further right".

So, e.g., if a problem is in **NP**, is it an easy one from **P** or a hard one like SAT?
How to tell those class memberships apart? Why can't we just say $P \in \mathbf{NP}$ but $P \notin \mathbf{P}$

Relationships among Complexity Classes: Heads up

What's also known to the literature:

- **PSPACE** = **NPSPACE** and **EXPSPACE** = **NEXPSPACE** (Savitch's Theorem, 1970 – see week 9)
- **P** $\subsetneq$ **EXPTIME** (by time Hierarchy Theorem(s), 1972, 1978, 1983)

So, in total, we will get: (Check yourself what we did not prove yet.)

- $\underbrace{\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{(N)PSPACE} \subseteq \mathbf{EXPTIME}}_{\neq} \subseteq \mathbf{NEXPTIME} \subseteq \mathbf{(N)EXPSPACE}$

Note how every problem in a class "further left" is also in a class "further right".

So, e.g., if a problem is in **NP**, is it an <u>easy</u> one from **P** or a hard one like SAT?
How to tell those class memberships apart? Why can't we just say $P \in \mathbf{NP}$ but $P \notin \mathbf{P}$

Answer: <u>Hardness</u> and <u>Completeness!</u> (Later in this section.)

Reductions

Basic Definitions

This is the most important (and fun!) part of this week!

- We want to transform problems into each other – via "specific" reductions.
- I.e., we solve "our given problem" by turning it into a known one (which must be as least "as hard"; otherwise that's not possible).

Basic Definitions

This is the most important (and fun!) part of this week!

- We want to transform problems into each other – via "specific" reductions.
- I.e., we solve "our given problem" by turning it into a known one (which must be as least "as hard"; otherwise that's not possible).

Definition

$f : \Sigma^* \to \Sigma^*$ is a polytime-computable function if some polynomial time TM M exists that halts with just $f(w)$ on its tape, when started on any input $w \in \Sigma^*$.

Basic Definitions

This is the most important (and fun!) part of this week!

- We want to transform problems into each other – via "specific" reductions.
- I.e., we solve "our given problem" by turning it into a known one (which must be as least "as hard"; otherwise that's not possible).

Definition

$f : \Sigma^* \to \Sigma^*$ is a polytime-computable function if some polynomial time TM M exists that halts with just $f(w)$ on its tape, when started on any input $w \in \Sigma^*$.

Definition

$A \subseteq \Sigma_1^*$ is polynomial time mapping-reducible to $B \subseteq \Sigma_2^*$, written $A \leq_{\mathbf{P}} B$, if a polytime-computable function $f : \Sigma_1^* \to \Sigma_2^*$ exists that is also a reduction (from A to B).

Reductions

Definition

- *A reduction is a polynomial-time translation of the problem, say r.*
- *More precisely:*
  - $r(w)$ *can be computed in time polynomial in* $|w|$.
  - $w \in A$ *if and only if* $r(w) \in B$ *(so it "preserves the answer").*

Reductions

Definition

- *A reduction is a polynomial-time translation of the problem, say r.*
- *More precisely:*
  - *$r(w)$ can be computed in time polynomial in $|w|$.*
  - *$w \in A$ if and only if $r(w) \in B$ (so it "preserves the answer").*

Example:

- $\text{EVEN} := \{n \mid n \bmod 2 = 0\}$, $\text{ODD} := \{n \mid n \bmod 2 = 1\}$
- Reduction from ODD to EVEN: // i.e., deciding ODD by deciding EVEN
  - $r(k) = k + 1$, so we get $k \in \text{ODD}$ iff $r(k) \in \text{EVEN}$
  - So essentially we can define odd(n):=even($r(n)$) now.
  - This shows that EVEN is at least as hard as ODD.

Reductions

Definition

- *A reduction is a polynomial-time translation of the problem, say r.*
- *More precisely:*
  - *$r(w)$ can be computed in time polynomial in $|w|$.*
  - *$w \in A$ if and only if $r(w) \in B$ (so it "preserves the answer").*

Example:

- EVEN $:= \{n \mid n \bmod 2 = 0\}$, ODD $:= \{n \mid n \bmod 2 = 1\}$
- Reduction from ODD to EVEN: // i.e., deciding ODD by deciding EVEN
  - $r(k) = k + 1$, so we get $k \in$ ODD iff $r(k) \in$ EVEN
  - So essentially we can define odd(n):=even($r(n)$) now.
  - This shows that EVEN is at least as hard as ODD.
- If however our goal would have been to show that the 'new' problem ODD is at least as hard as EVEN, then we would have had to reduce from EVEN to ODD (though r would have been the same). Check this statement after "hardness" was introduced!

Reductions

Definition

- *A reduction is a polynomial-time translation of the problem, say r.*
- *More precisely:*
  - $r(w)$ *can be computed in time polynomial in* $|w|$.
  - $w \in A$ *if and only if* $r(w) \in B$ *(so it "preserves the answer").*

Example:

- EVEN $:= \{n \mid n \bmod 2 = 0\}$, ODD $:= \{n \mid n \bmod 2 = 1\}$
- Reduction from ODD to EVEN: // i.e., deciding ODD by deciding EVEN
  - $r(k) = k + 1$, so we get $k \in$ ODD iff $r(k) \in$ EVEN
  - So essentially we can define odd(n):=even($r(n)$) now.
  - This shows that EVEN is at least as hard as ODD.
- If however our goal would have been to show that the 'new' problem ODD is at least as hard as EVEN, then we would have had to reduce from EVEN to ODD (though r would have been the same). Check this statement after "hardness" was introduced!

Heads-up: Be cautious when numbers are involved! They grow exponentially!

Concatenation of **P**-reductions

Theorem w7.1

If $A \leq_{\mathbf{P}} B$ and $B \in \mathbf{P}$ then $A \in \mathbf{P}$.

Concatenation of **P**-reductions

Theorem w7.1

If $A \leq_{\mathbf{P}} B$ and $B \in \mathbf{P}$ then $A \in \mathbf{P}$.

Proof.

To decide $w \in A$ first compute $f(w)$ (in **P**) where f is the **P**-reduction from A to B, and then run a **P** decider for B. This is still in **P** because $p_1(p_2(n))$ is a polynomial if $p_1(n)$ and $p_2(n)$ are. □

Examples for what?

Disclaimer:

- The next few examples' main purpose is to show example reductions.
- However, they also:
  - Are examples for **NP**-complete problems (a concept not introduced yet).

Examples for what?

Disclaimer:

- The next few examples' main purpose is to show example reductions.
- However, they also:
  - Are examples for **NP**-complete problems (a concept not introduced yet).
  - Show how the respective problems could be solved (gives class memberships), which is completely irrelevant for (and independent of) reductions!

Examples for what?

Disclaimer:

- The next few examples' main purpose is to show example reductions.
- However, they also:
  - Are examples for **NP**-complete problems (a concept not introduced yet).
  - Show how the respective problems could be solved (gives class memberships), which is completely irrelevant for (and independent of) reductions!
  - Show the influence of numbers (and hence how tricky they are).

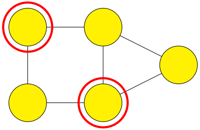
Thus: You should revisit this part after this week, after (**NP**-)completeness and hardness were defined.

Example: Independent Set

The Independent Set Problem:
Assume you want to throw a party. But you know that some of your friends don't get along. You only want to invite people that do get along.

Formalized as graph.

- vertices are your mates
- draw an edge between two vertices if people don't get along



Example of an independent set of size 2 (just the red-circled vertices)

Example: Independent Set

The Independent Set Problem:
Assume you want to throw a party. But you know that some of your friends don't get along. You only want to invite people that do get along.

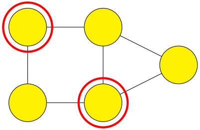
Formalized as graph.

- vertices are your mates
- draw an edge between two vertices if people don't get along

Problem:
Given a graph and $k \geq 0$, is there an independent set, i.e., a subset I of $\geq k$ vertices s.t.:

- no two elements of I are connected with an edge.
- i.e., everybody in I gets along

Formally, $IS := \{(\langle V, E\rangle, k) \mid \langle V, E\rangle$ has an independent set $\geq k,\ k \in \mathbb{N}\}$

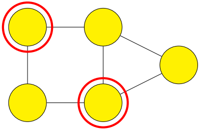


Example of an independent set of size 2 (just the red-circled vertices)

Solving the Independent Set Problem (Irrelevant for Reductions!)

Naive Implementation:

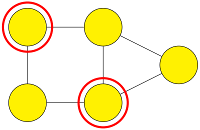
- loop through all subsets of size $\geq k$



Solving the Independent Set Problem (Irrelevant for Reductions!)

Naive Implementation:

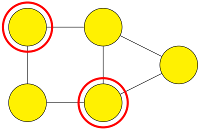
- loop through all subsets of size $\geq k$
  - How many are there? Note that k is an input to the tape: $n_{log(k)} \ldots n_1 n_0$
  - So, k grows exponential in $|k| = log(k)$. Always be aware of this when numbers are involved! (So, sometimes this increases the complexity.)
  - There are exponentially many subsets of size $\geq k$. So, why is that not doubly-exponential?



Solving the Independent Set Problem (Irrelevant for Reductions!)

Naive Implementation:

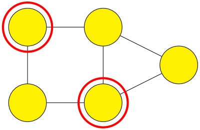
- loop through all subsets of size $\geq k$
  - How many are there? Note that k is an input to the tape: $n_{log(k)} \dots n_1 n_0$
  - So, k grows exponential in $|k| = log(k)$. Always be aware of this when numbers are involved! (So, sometimes this increases the complexity.)
  - There are exponentially many subsets of size $\geq k$. So, why is that not doubly-exponential? $\rightarrow$ Because the nodes are part of the input!



Solving the Independent Set Problem (Irrelevant for Reductions!)

Naive Implementation:

- loop through all subsets of size $\geq k$
  - How many are there? Note that k is an input to the tape: $n_{log(k)} \dots n_1 n_0$
  - So, k grows exponential in $|k| = log(k)$. Always be aware of this when numbers are involved! (So, sometimes this increases the complexity.)
  - There are exponentially many subsets of size $\geq k$. So, why is that not doubly-exponential? $\rightarrow$ Because the nodes are part of the input!
- and check whether they are independent sets



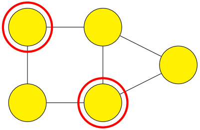
$\rightarrow$ Proves membership in

Solving the Independent Set Problem (Irrelevant for Reductions!)

Naive Implementation:

- loop through all subsets of size $\geq k$
  - How many are there? Note that k is an input to the tape: $n_{log(k)} \ldots n_1 n_0$
  - So, k grows exponential in $|k| = log(k)$. Always be aware of this when numbers are involved! (So, sometimes this increases the complexity.)
  - There are exponentially many subsets of size $\geq k$. So, why is that not doubly-exponential? $\rightarrow$ Because the nodes are part of the input!
- and check whether they are independent sets

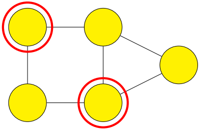
$\rightarrow$ Proves membership in **EXPTIME**



Solving the Independent Set Problem (Irrelevant for Reductions!)

Naive Implementation:

- loop through all subsets of size $\geq k$
  - How many are there? Note that k is an input to the tape: $n_{log(k)} \dots n_1 n_0$
  - So, k grows exponential in $|k| = log(k)$. Always be aware of this when numbers are involved! (So, sometimes this increases the complexity.)
  - There are exponentially many subsets of size $\geq k$. So, why is that not doubly-exponential? $\rightarrow$ Because the nodes are part of the input!
- and check whether they are independent sets



$\rightarrow$ Proves membership in **EXPTIME**

Using Non-deterministic Turing Machines:

- guess a subset of vertices of size $\geq k$ (with $k \leq |V|$)
- check whether it is an independent set

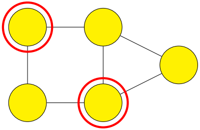
$\rightarrow$ Proves membership in **NP**

Solving the Independent Set Problem (Irrelevant for Reductions!)

Naive Implementation:

- loop through all subsets of size $\geq k$
  - How many are there? Note that k is an input to the tape: $n_{log(k)} \dots n_1 n_0$
  - So, k grows exponential in $|k| = log(k)$. Always be aware of this when numbers are involved! (So, sometimes this increases the complexity.)
  - There are exponentially many subsets of size $\geq k$. So, why is that not doubly-exponential? $\rightarrow$ Because the nodes are part of the input!
- and check whether they are independent sets

$\rightarrow$ Proves membership in **EXPTIME**



Using Non-deterministic Turing Machines:

- guess a subset of vertices of size $\geq k$ (with $k \leq |V|$)
- check whether it is an independent set

$\rightarrow$ Proves membership in **NP**

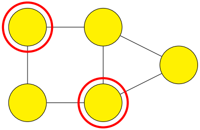
Question: Can we do better? Is there a **P** algorithm?

Solving the Independent Set Problem (Irrelevant for Reductions!)

Naive Implementation:

- loop through all subsets of size $\geq k$
  - How many are there? Note that k is an input to the tape: $n_{log(k)} \dots n_1 n_0$
  - So, k grows exponential in $|k| = log(k)$. Always be aware of this when numbers are involved! (So, sometimes this increases the complexity.)
  - There are exponentially many subsets of size $\geq k$. So, why is that not doubly-exponential? $\rightarrow$ Because the nodes are part of the input!
- and check whether they are independent sets

$\rightarrow$ Proves membership in **EXPTIME**



Using Non-deterministic Turing Machines:

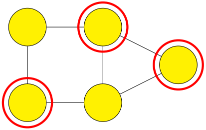
- guess a subset of vertices of size $\geq k$ (with $k \leq |V|$)
- check whether it is an independent set

$\rightarrow$ Proves membership in **NP**

Question: Can we do better? Is there a **P** algorithm?
Answer: We don't know! But "hardness" helps giving a partial answer.

Example 2: Vertex Cover

Given a graph $G = \langle V, E \rangle$, a vertex cover is a set C of vertices such that every edge in G has at least one vertex in C.



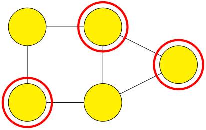
Example vertex cover: The red-circled vertices. (Since they "cover" all edges.)

Vertex Cover (Decision) Problem.

- Given graph $\langle V, E \rangle$ and $k \geq 0$, is there a vertex cover of size $\leq k$?
- $VC := \{(\langle V, E \rangle, k) \mid \langle V, E \rangle$ has a vertex cover $\leq k,\ k \in \mathbb{N}\ \}$

Example 2: Vertex Cover

Given a graph $G = \langle V, E\rangle$, a vertex cover is a set C of vertices such that every edge in G has at least one vertex in C.



Example vertex cover: The red-circled vertices. (Since they "cover" all edges.)

Vertex Cover (Decision) Problem.

- Given graph $\langle V, E\rangle$ and $k \geq 0$, is there a vertex cover of size $\leq k$?
- $VC := \{(\langle V, E\rangle, k) \mid \langle V, E\rangle$ has a vertex cover $\leq k,\ k \in \mathbb{N}\ \}$

Naive Algorithm:

- search through all subsets of size $\leq k$ (and $k \leq |V|$), which is exponential
- check whether it's a vertex cover

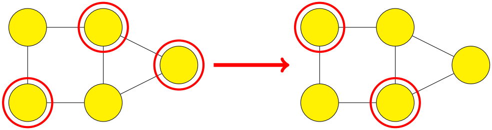
$\rightarrow$ This proves $VC \in$ **EXPTIME**, but we can do better!
(I.e., we could also guess and verify as before, giving $VC \in$ **NP**.)

From Vertex Cover to Independent Set

Reductions. Use solutions of one problem to solve another.

Observation. Let G be a graph with n vertices and $k \geq 0$.

› G has a VC of size $\leq k$ iff G has an IS of size $\geq n - k$



› Why?

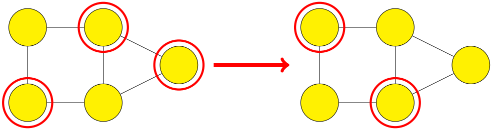
- VC with $\leq k$ vertices needs to cover all edges.
- IS with $\geq n - k$ vertices can't cover any edge.

From Vertex Cover to Independent Set

Reductions. Use solutions of one problem to solve another.

Observation. Let G be a graph with n vertices and $k \geq 0$.

- G has a VC of size $\leq k$ iff G has an IS of size $\geq n - k$



- Why?
  - VC with $\leq k$ vertices needs to cover all edges.
  - IS with $\geq n - k$ vertices can't cover any edge.

What's the reduction? Vertex cover to independent set:

- $\langle G, k \rangle \in VC$ iff $r(\langle G, k \rangle) \in IS$, where $= r(\langle G, k \rangle) = \langle G, n - k \rangle$.
- Here, the reduction r only changes the number, but nothing else. But for most reductions, we will have to "translate problems", e.g., when turning a SAT problem into a VC problem! (Or vice versa.)

Important Notes on Reductions

Be aware!

- So far, we only reduced problems, which were "equally hard", they were just "different flavors of the same problem":
  - EVEN vs. ODD
  - Independent Set (IS) vs. Vertex Cover (VC)

Important Notes on Reductions

Be aware!

- So far, we only reduced problems, which were "equally hard", they were just "different flavors of the same problem":
  - EVEN vs. ODD
  - Independent Set (IS) vs. Vertex Cover (VC)
- But reductions also work (in one direction!) when one problem is "strictly harder" than another!
  - You should be able to reduce EVEN (or ODD) to Vertex Cover!

Important Notes on Reductions

Be aware!

- So far, we only reduced problems, which were "equally hard", they were just "different flavors of the same problem":
  - EVEN vs. ODD
  - Independent Set (IS) vs. Vertex Cover (VC)
- But reductions also work (in one direction!) when one problem is "strictly harder" than another!
  - You should be able to reduce EVEN (or ODD) to Vertex Cover!
    (Reducing a problem in **P** to a problem that's **NP**-hard.)
- Reductions are not just used to show "hardness" (defined later), but also for membership! (Just depending on the direction of reduction.)

(Hardness and completeness are explained in the next section... So, re-visit this!)

Important Notes on Reductions

Be aware!

- So far, we only reduced problems, which were "equally hard", they were just "different flavors of the same problem":
  - EVEN vs. ODD
  - Independent Set (IS) vs. Vertex Cover (VC)
- But reductions also work (in one direction!) when one problem is "strictly harder" than another!
  - You should be able to reduce EVEN (or ODD) to Vertex Cover! (Reducing a problem in **P** to a problem that's **NP**-hard.)
  - Reducing Vertex Cover to EVEN (or ODD) is believed to be impossible. If you can, you proved **P** = **NP** (and hence a Millennium problem).
- Reductions are not just used to show "hardness" (defined later), but also for membership! (Just depending on the direction of reduction.)

(Hardness and completeness are explained in the next section... So, re-visit this!)

Important Notes on Reductions

Be aware!

- So far, we only reduced problems, which were "equally hard", they were just "different flavors of the same problem":
  - EVEN vs. ODD
  - Independent Set (IS) vs. Vertex Cover (VC)
- But reductions also work (in one direction!) when one problem is "strictly harder" than another!
  - You should be able to reduce EVEN (or ODD) to Vertex Cover! (Reducing a problem in **P** to a problem that's **NP**-hard.)
  - Reducing Vertex Cover to EVEN (or ODD) is believed to be impossible. If you can, you proved $\mathbf{P} = \mathbf{NP}$ (and hence a Millennium problem).
  - You should be able to reduce Vertex Cover to the Sokoban game. (Reducing an **NP**-complete problem to one that's **PSPACE**-hard.)
- Reductions are not just used to show "hardness" (defined later), but also for membership! (Just depending on the direction of reduction.)

(Hardness and completeness are explained in the next section... So, re-visit this!)

Membership, Hardness, Completeness

Informal summary and outlook

- We know that any complexity class **X**, like **P**, **NP**, etc., is just a set of languages.

Informal summary and outlook

- We know that any complexity class **X**, like **P**, **NP**, etc., is just a set of languages.
- Every "problem", like VC, is just a language.

Informal summary and outlook

- We know that any complexity class **X**, like $\mathbf{P}, \mathbf{NP}$, etc., is just a set of languages.
- Every "problem", like VC, is just a language.
- Thus, a problem having some complexity can be stated in different ways:
  - VC is in **NP**, and in **EXPTIME** etc, but don't know whether it's in **P**.
  - Or simply VC $\in$ **NP**

Informal summary and outlook

- We know that any complexity class **X**, like **P**, **NP**, etc., is just a set of languages.
- Every "problem", like VC, is just a language.
- Thus, a problem having some complexity can be stated in different ways:
  - VC is in **NP**, and in **EXPTIME** etc, but don't know whether it's in **P**.
  - Or simply VC $\in$ **NP**
- We'll introduce "hardness" and "completeness" as a boundary for upper and lower bounds of complexity class memberships.

Membership, Hardness, and Completeness

Definition (membership, hardness, completeness)

Let $\mathbf{X}$ *be a complexity class. (Not* $\mathbf{P}$ *in case of hardness.)*

A language B *is* <u>$\mathbf{X}$*-complete*</u> *iff*

1. $B \in \mathbf{X}$ $= \mathbf{X}$ *membership*
2. *For all* $A \in \mathbf{X}$, $A \leq_{\mathbf{P}} B$ *(i.e., every* A *in* $\mathbf{X}$ *is poly-reducable to* B*)* $= \mathbf{X}$*-hardness*

Membership, Hardness, and Completeness

Definition (membership, hardness, completeness)

Let **X** *be a complexity class. (Not* **P** *in case of hardness.)*

A language B is **X**-*complete iff*

1. $B \in \mathbf{X}$ $= \mathbf{X}$ *membership*
2. *For all $A \in \mathbf{X}$, $A \leq_{\mathbf{P}} B$ (i.e., every A in* **X** *is poly-reducable to B)* $=$ **X**-*hardness*

- This definition applies to all complexity classes **X**.
- The concept of **P**-hardness is however defined differently and not covered.
- So we have "for all A holds $A \leq_{\mathbf{P}} B$", and therefore we know that B is "hard/expressive enough" to decide all other problems in **X**.
  (Since we decide these other A-problems using our B-problem!)

Membership, Hardness, and Completeness

Definition (membership, hardness, completeness)

Let $\mathbf{X}$ *be a complexity class. (Not* $\mathbf{P}$ *in case of hardness.)*

A language B *is* $\underline{\mathbf{X}\text{-complete}}$ *iff*

1. $B \in \mathbf{X}$ $= \mathbf{X}$ *membership*
2. *For all* $A \in \mathbf{X}$, $A \leq_{\mathbf{P}} B$ *(i.e., every* A *in* $\mathbf{X}$ *is poly-reducable to* B*)* $= \mathbf{X}$*-hardness*

- This definition applies to all complexity classes **X**.
- The concept of **P**-hardness is however defined differently and not covered.
- So we have "for all A holds $A \leq_{\mathbf{P}} B$", and therefore we know that B is "hard/expressive enough" to decide all other problems in **X**.
  (Since we decide these other A-problems using our B-problem!)
- Therefore, **X**-complete problems are the hardest ones in **X**.
  For example: we can now say that VC is **NP**-complete, rather than just saying that it's in **NP** – thus indicating that it's a "hard" problem from **NP**, not an easy one that's in **P** (like ODD).

Motivation / further Explanations

Why are we interested in showing hardness/completeness in the first place?

Motivation / further Explanations

Why are we interested in showing hardness/completeness in the first place?

- For example, if we fail in providing a **P** procedure for some problem, it could be:
  - Because we just did not discover it (yet? – keep searching!)
  - or because doesn't even exist! (bail!)

Motivation / further Explanations

Why are we interested in showing hardness/completeness in the first place?

- For example, if we fail in providing a **P** procedure for some problem, it could be:
  - Because we just did not discover it (yet? – keep searching!)
  - or because doesn't even exist! (bail!)
- So ... How to find out whether we should just work harder?

Motivation / further Explanations

Why are we interested in showing hardness/completeness in the first place?

- For example, if we fail in providing a **P** procedure for some problem, it could be:
  - Because we just did not discover it (yet? – keep searching!)
  - or because doesn't even exist! (bail!)
- So ... How to find out whether we should just work harder?
  - If we can prove, e.g., **NP**-completeness, then at least we know that nobody before you found a **P** procedure (similar arguments hold for higher classes). (And maybe none even exists, which follows directly once somebody proves **P** $\neq$**NP**.)

Motivation / further Explanations

Why are we interested in showing hardness/completeness in the first place?

- For example, if we fail in providing a **P** procedure for some problem, it could be:
  - Because we just did not discover it (yet? – keep searching!)
  - or because doesn't even exist! (bail!)
- So ... How to find out whether we should just work harder?
  - If we can prove, e.g., **NP**-completeness, then at least we know that nobody before you found a **P** procedure (similar arguments hold for higher classes). (And maybe none even exists, which follows directly once somebody proves **P** $\neq$**NP**.)
- Why completeness? Why not just showing hardness?

Motivation / further Explanations

Why are we interested in showing hardness/completeness in the first place?

- For example, if we fail in providing a **P** procedure for some problem, it could be:
  - Because we just did not discover it (yet? – keep searching!)
  - or because doesn't even exist! (bail!)
- So ... How to find out whether we should just work harder?
  - If we can prove, e.g., **NP**-completeness, then at least we know that nobody before you found a **P** procedure (similar arguments hold for higher classes). (And maybe none even exists, which follows directly once somebody proves **P** $\neq$**NP**.)
- Why completeness? Why not just showing hardness?
  - Since the problem could be even harder! (E.g., **PSPACE**-hard, **EXPTIME**-hard, etc., rather than than **NP-complete**.)
  - Each problem class has specific "properties". E.g., "**NP**-complete looks like Logic", "**PSPACE**-complete looks like planning", etc. (But if you only say, e.g., **NP**-hard, it could still be **PSPACE**-complete etc.)

Alternative way to show Hardness and Completeness

Theorem w7.1

If B is $\mathbf{X}$-hard and $B \leq_{\mathbf{P}} C$, then C is $\mathbf{X}$-hard.
(Not for $\mathbf{P}$, as we don't cover $\mathbf{P}$-hardness.)

Alternative way to show Hardness and Completeness

Theorem w7.1

If B is $\mathbf{X}$-hard and $B \leq_{\mathbf{P}} C$, then C is $\mathbf{X}$-hard.
(Not for $\mathbf{P}$, as we don't cover $\mathbf{P}$-hardness.)

Corollary w7.2

If B is $\mathbf{X}$-complete and $B \leq_{\mathbf{P}} C$ for $C \in \mathbf{X}$, then C is $\mathbf{X}$-complete.

Alternative way to show Hardness and Completeness

Theorem w7.1

If B is $\mathbf{X}$-hard and $B \leq_{\mathbf{P}} C$, then C is $\mathbf{X}$-hard.
(Not for $\mathbf{P}$, as we don't cover $\mathbf{P}$-hardness.)

Corollary w7.2

If B is $\mathbf{X}$-complete and $B \leq_{\mathbf{P}} C$ for $C \in \mathbf{X}$, then C is $\mathbf{X}$-complete.

Proof.

Polynomial time reductions compose. □

Alternative way to show Hardness and Completeness

Theorem w7.1

If B is $\mathbf{X}$-hard and $B \leq_{\mathbf{P}} C$, then C is $\mathbf{X}$-hard.
(Not for $\mathbf{P}$, as we don't cover $\mathbf{P}$-hardness.)

Corollary w7.2

If B is $\mathbf{X}$-complete and $B \leq_{\mathbf{P}} C$ for $C \in \mathbf{X}$, then C is $\mathbf{X}$-complete.

Proof.

Polynomial time reductions compose. □

Important! This Corollary is of major importance!! Why?

Alternative way to show Hardness and Completeness

Theorem w7.1

If B is $\mathbf{X}$-hard and $B \leq_{\mathbf{P}} C$, then C is $\mathbf{X}$-hard.
(Not for $\mathbf{P}$, as we don't cover $\mathbf{P}$-hardness.)

Corollary w7.2

If B is $\mathbf{X}$-complete and $B \leq_{\mathbf{P}} C$ for $C \in \mathbf{X}$, then C is $\mathbf{X}$-complete.

Proof.

Polynomial time reductions compose. □

Important! This Corollary is of major importance!! Why?
→ It gives us a convenient procedure to show completeness!

- First, show membership in $\mathbf{X}$. (That's almost always the easy part.)
- Then, show hardness by grabbing an $\mathbf{X}$-complete problem and reduce it to yours!