Computational Complexity; slides 4, HT 2019
Deterministic complexity classes

Prof. Paul W. Goldberg (Dept. of Computer Science,
University of Oxford)

HT 2019

Measuring Complexity

Complexity Theory: Look at the fine structure of decidable languages.

Goal: Classify languages according to the amount of resources needed to solve them.

Resources: In this lecture we will primarily consider

- **time** – the running time of algorithms (steps on a Turingmachine)
- **space** – the amount of additional memory needed (cells on the Turing tapes)

Measuring Complexity

Definition.
Let $\mathcal{M}$ be a Turing acceptor and let $S, T : \mathbb{N} \to \mathbb{N}$ be functions.

1. $\mathcal{M}$ is T-time bounded if it halts on every input $w \in \Sigma^*$ after $\leq T(|w|)$ steps.
2. $\mathcal{M}$ is S-space bounded if it halts on every input $w \in \Sigma^*$ using $\leq S(|w|)$ cells on its tapes.

   (Here we assume that the Turing machines have a separate input tape that we do not count in measuring space complexity.)

Deterministic Complexity Classes

Definition.

Let $T, S : \mathbb{N} \rightarrow \mathbb{N}$ be monotone growing functions. Define

1. DTIME(T) as the class of languages $\mathcal{L}$ for which there is a T-time bounded k-tape Turing acceptor deciding $\mathcal{L}$, for some $k \geq 1$.
2. DSPACE(S) as the class of languages $\mathcal{L}$ for which there is a S-space bounded k-tape Turing acceptor deciding $\mathcal{L}$, $k \geq 1$.

Deterministic Complexity Classes

Definition.

Let $T, S : \mathbb{N} \to \mathbb{N}$ be monotone growing functions. Define

1. DTIME(T) as the class of languages $\mathcal{L}$ for which there is a T-time bounded k-tape Turing acceptor deciding $\mathcal{L}$, for some $k \geq 1$.
2. DSPACE(S) as the class of languages $\mathcal{L}$ for which there is a S-space bounded k-tape Turing acceptor deciding $\mathcal{L}$, $k \geq 1$.

Important Complexity Classes:

- Time classes:
  - PTIME $:= \bigcup_{d \in \mathbb{N}} \mathrm{DTIME}(n^d)$ — polynomial time
  - EXPTIME $:= \bigcup_{d \in \mathbb{N}} \mathrm{DTIME}(2^{n^d})$ — exponential time
  - 2-EXPTIME $:= \bigcup_{d \in \mathbb{N}} \mathrm{DTIME}(2^{2^{n^d}})$ — double exp time
- Space classes:
  - LOGSPACE $:= \bigcup_{d \in \mathbb{N}} \mathrm{DSPACE}(d \log n)$
  - PSPACE $:= \bigcup_{d \in \mathbb{N}} \mathrm{DSPACE}(n^d)$
  - EXPSPACE $:= \bigcup_{d \in \mathbb{N}} \mathrm{DSPACE}(2^{n^d})$

But wait...

But wait...

Do these classes depend on exact def of "Turing machine"?

But wait...

Do these classes depend on exact def of "Turing machine"?

Yes, for DTIME(T), DPSPACE(S);
No for the others

Indeed, usually don't need to be refer explicitly to "Turing machine".
But watch out for nondeterminism (details later)

Time Complexity Classes

Important Time Complexity Classes:

- PTIME $:= \bigcup_{d\in\mathbb{N}} \mathrm{DTIME}(n^d)$ polynomial time
- EXPTIME $:= \bigcup_{d\in\mathbb{N}} \mathrm{DTIME}(2^{n^d})$ exponential time

Not quite so important:

- 2-EXPTIME $:= \bigcup_{d\in\mathbb{N}} \mathrm{DTIME}(2^{2^{n^d}})$ double exp time

Note: Complexity classes are classes of languages.

Time Complexity:

$$\mathrm{PTIME} \subseteq \mathrm{EXPTIME} \subseteq \text{2-EXPTIME} \subseteq \cdots \subseteq i\text{-EXPTIME} \subseteq \ldots$$

Time Complexity Classes

Important Time Complexity Classes:

- PTIME $:= \bigcup_{d\in\mathbb{N}} \mathrm{DTIME}(n^d)$ polynomial time
- EXPTIME $:= \bigcup_{d\in\mathbb{N}} \mathrm{DTIME}(2^{n^d})$ exponential time

Not quite so important:

- 2-EXPTIME $:= \bigcup_{d\in\mathbb{N}} \mathrm{DTIME}(2^{2^{n^d}})$ double exp time

Note: Complexity classes are classes of languages.

Time Complexity:

$$\mathrm{PTIME} \subseteq \mathrm{EXPTIME} \subseteq 2\text{-}\mathrm{EXPTIME} \subseteq \cdots \subseteq i\text{-}\mathrm{EXPTIME} \subseteq \ldots$$

Alternative Definition: Sometimes PTIME is defined as

$$\mathrm{PTIME} := \bigcup_{d\in\mathbb{N}} \mathrm{DTIME}(\mathcal{O}(n^d))$$

Theorem. (Linear Speed-Up Theorem)

Let $k > 1$ and $c > 0$ $T : \mathbb{N} \rightarrow \mathbb{N}$ $\mathcal{L} \subseteq \Sigma^*$ be a language.

If $\mathcal{L}$ can be decided by a $T(n)$ time-bounded k-tape TM

$$\mathcal{M} := (Q, \Sigma, \Gamma, q_0, \delta, F)$$

then $\mathcal{L}$ can be decided by a $(\frac{1}{c} \cdot T(n) + n + 2)$ time-bounded k-tape TM

$$\mathcal{M}^* := (Q', \Sigma, \Gamma', q'_0, \delta', F').$$

Proof idea. Let $\Gamma' := \Sigma \cup \Gamma^m$ where $m := \lceil 6c \rceil$. We construct $\mathcal{M}^*$:

Step 1: Compress $\mathcal{M}$'s input.

Copy (in $n + 2$ steps) the input onto tape 2, compressing m symbols into one (i.e., each symbol correspondes to an m-tuple from Γ^m)

Step 2: Simulate $\mathcal{M}$'s computation, m steps at once.

1. Read (in 4 steps) symbols to the left, right and the current position
   and "store" in Γ' (using $|Q \times \{1, \dots, m\}^k \times \Gamma^{3mk}|$ extra states).
2. Simulate (in 2 steps) the next m steps of $\mathcal{M}$ (as $\mathcal{M}$ can only modify the current position and one of its neighbours)
3. $\mathcal{M}^*$ accepts (rejects) if $\mathcal{M}$ accepts (rejects)

(see Papadimitriou Thm 2.2, page 32)

A Hierarchy of Complexity Classes?

Questions:

- Can we always solve more problems if we have more resources?
- If not, how much more resources do we need to be able to solve strictly more problems?
- How do the complexity classes relate to each other?

  $\rightsquigarrow$ see later in the course.

- How do we classify "efficient" in terms of complexity classes?

  $\rightsquigarrow$ see next section

A Hierarchy of Complexity Classes?

Questions:

- Can we always solve more problems if we have more resources?
- If not, how much more resources do we need to be able to solve strictly more problems?
- How do the complexity classes relate to each other?

  $\rightsquigarrow$ see later in the course.

- How do we classify "efficient" in terms of complexity classes?

  $\rightsquigarrow$ see next section

- Are there any tools by which we can show that a problem is in any of these classes but not in another?

A Hierarchy of Complexity Classes?

Questions:

- Can we always solve more problems if we have more resources?
- If not, how much more resources do we need to be able to solve strictly more problems?
- How do the complexity classes relate to each other?

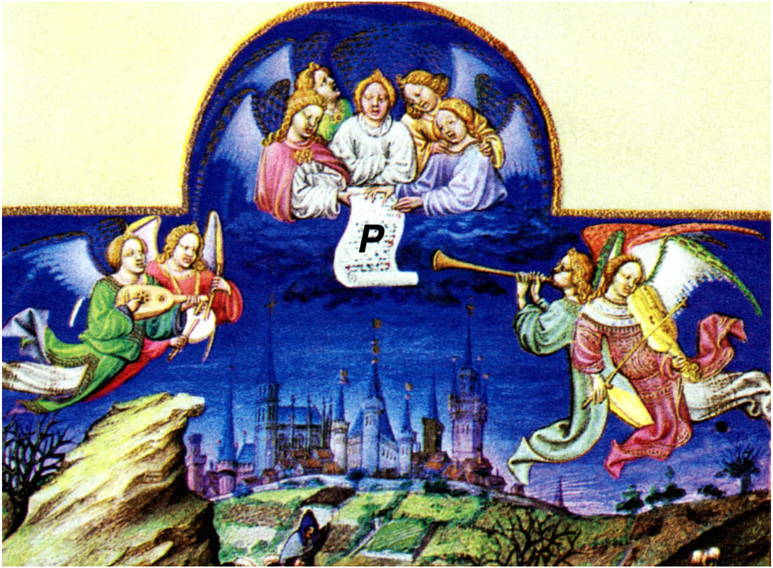
  $\rightsquigarrow$ see later in the course.

- How do we classify "efficient" in terms of complexity classes?

  $\rightsquigarrow$ see next section

- Are there any tools by which we can show that a problem is in any of these classes but not in another?
- Are there any other interesting models of computation?
  - Non-deterministic computation $\rightsquigarrow$ next part of course
  - Randomised algorithms $\rightsquigarrow$ last part of course (time permitting)

PTIME, usually called P



Polynomial Time

"Intuitive" definition of "efficient":

- Any linear time computation is "efficient"
- Any program that
  - performs "efficient" operations (e.g. linear number of iterations) and
  - only uses "efficient" subprograms

  is "efficient".

Polynomial Time

"Intuitive" definition of "efficient":

- Any linear time computation is "efficient"
- Any program that
  - performs "efficient" operations (e.g. linear number of iterations) and
  - only uses "efficient" subprograms

  is "efficient".

This turns out to be equivalent to PTIME.

$$\text{PTIME} := \bigcup_{d \in \mathbb{N}} \text{DTIME}(n^d)$$

PTIME serves as a mathematical model of "efficient" computation.

Robustness of the Definition

If PTIME is to be the mathematical model of efficient computation, it should not depend on

- the exact computation-model we are using,
- or how we encode the input (within reason).

Robustness of the Definition

If PTIME is to be the mathematical model of efficient computation, it should not depend on

- the exact computation-model we are using,
- or how we encode the input (within reason).

Different Models of Computation:

1. We can simulate t steps of a k-tape Turing machine with an equivalent 1-tape TM in t^2 steps.
2. We can simulate t steps of a two-way infinite k-tape Turing machine with an equivalent standard k-tape TM in $\mathcal{O}(t)$ steps.
3. We can simulate t steps of a RAM-machine with a 3-tape TM in $\mathcal{O}(t^3)$ steps. Vice-versa in $\mathcal{O}(t)$ steps.

Robustness of the Definition

If PTIME is to be the mathematical model of efficient computation, it should not depend on

- the exact computation-model we are using,
- or how we encode the input (within reason).

Different Models of Computation:

1. We can simulate t steps of a k-tape Turing machine with an equivalent 1-tape TM in t^2 steps.
2. We can simulate t steps of a two-way infinite k-tape Turing machine with an equivalent standard k-tape TM in $\mathcal{O}(t)$ steps.
3. We can simulate t steps of a RAM-machine with a 3-tape TM in $\mathcal{O}(t^3)$ steps. Vice-versa in $\mathcal{O}(t)$ steps.

Consequence: PTIME is the same for all these models (unlike linear time)

Robustness of the Definition

Strong Church-Turing Hypothesis:

Any function which can be computed by any well-defined procedure can be computed by a Turing machine with only polynomial overhead.

(may be challenged by Quantum Computers)

Different Encodings

Lemma.

1. For any $n \in \mathbb{N}$, the length of the encoding of n in base b_1 and base b_2 are related by a constant factor, for all $b_1, b_2 \geq 2$.
2. For any graph G, the length of its encoding as an
   - adjacency matrix
   - list of edges
   - adjacency list
   - ...

   are all related by a polynomial factor.

Different Encodings

Lemma.

1. For any $n \in \mathbb{N}$, the length of the encoding of n in base b_1 and base b_2 are related by a constant factor, for all $b_1, b_2 \geq 2$.
2. For any graph G, the length of its encoding as an
   - adjacency matrix
   - list of edges
   - adjacency list
   - ...

   are all related by a polynomial factor.

Consequence: PTIME is the same for all these encoding (unlike linear time)

PTIME = tractable?

The class PTIME is a reasonable mathematical model of the class of problems which are tractable or solvable in practice.

However: This correspondence is not exact:

- When the degree of polynomials is very high, the time grows so quickly that in practice the problem is not solvable.
- The constants may also be very large

However:

For many concrete PTIME-problems arising in practice, algorithms with moderate exponents and constants have been found.

Growth Rate of Functions (Garey/Johnson '79)

Time complexity function	Size n: 10	20	30	40	50	60
n	.00001 second	.00002 second	.00003 second	.00004 second	.00005 second	.00006 second
n^2	.0001 second	.0004 second	.0009 second	.0016 second	.0025 second	.0036 second
n^3	.001 second	.008 second	.027 second	.064 second	.125 second	.216 second
n^5	.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

Figure 1.2 Comparison of several polynomial and exponential time complexity

Proving a problem is in PTIME

You have done this before

- The most direct way to show that a problem is in PTIME is to exhibit a polynomial time algorithm that solves it.
- Even a naive polynomial-time algorithm often provides a good insight into how the problem can be solved efficiently.
- Because of robustness, we do not generally need to specify all the details of the machine model or the encoding.

  $\rightsquigarrow$ pseudo-code is sufficient.

"in PTIME" less specific than, e.g. "in DTIME(n^2)"; some technical details are avoided

Example: Satisfiability

Some of the most important problems concern logical formulae

Recall propositional logic

Formulae of propositional logic are built up inductively

- Variables: $X_i \qquad i \in \mathbb{N}$
- Boolean connectives:
  If φ, ψ are propositional formulae then so are
  - $(\psi \vee \varphi)$
  - $(\psi \wedge \varphi)$
  - $\neg\varphi$

Example:

$$(X_1 \vee X_2 \vee \neg X_5) \ \wedge \ (\neg X_2 \vee \neg X_4 \vee \neg X_5) \ \wedge \ (X_2 \vee X_3 \vee X_4)$$

Conjunctive Normal Form

A propositional logic formula φ is in conjunctive normal form (CNF) if

$$\varphi := C_1 \wedge \cdots \wedge C_m$$

where each C_i is a clause, that is, a disjunction of literals

$$C_i := (L_{i1} \vee \cdots \vee L_{ik})$$

A literal is a variable X_i or a negation $\neg X_i$ thereof.

k-CNF: If φ has at most k literals per clause.

Example:

$(X_1 \vee X_2 \vee \neg X_5) \ \wedge \ (\neg X_2 \vee \neg X_4 \vee \neg X_5) \ \wedge \ (X_2 \vee X_3 \vee X_4)$
(3-CNF)

Conjunctive Normal Form

A propositional logic formula φ is in conjunctive normal form (CNF) if

$$\varphi := C_1 \wedge \cdots \wedge C_m$$

where each C_i is a clause, that is, a disjunction of literals

$$C_i := (L_{i1} \vee \cdots \vee L_{ik})$$

A literal is a variable X_i or a negation $\neg X_i$ thereof.

***k*-CNF:** If φ has at most k literals per clause.

Example:

$(X_1 \vee X_2 \vee \neg X_5) \ \wedge \ (\neg X_2 \vee \neg X_4 \vee \neg X_5) \ \wedge \ (X_2 \vee X_3 \vee X_4)$
(3-CNF)

Notation:

$\varphi := \{ \ \{X_1, X_2, \neg X_5\}, \quad \{\neg X_2, \neg X_4, \neg X_5\}, \quad \{X_2, X_3, X_4\} \ \}$

Satisfiability

Definition. A formula φ is satisfiable if there is a satisfying assignment for φ.

In the case of formulae in CNF:
An assignment β assigning values 0 or 1 to the variables of φ so that every clause contains at least

- one variable to which β assigns 1 or
- one negated variable to which β assigns 0.

Example:

$$(X_1 \vee X_2 \vee \neg X_5) \;\wedge\; (\neg X_2 \vee \neg X_4 \vee \neg X_5) \;\wedge\; (X_2 \vee X_3 \vee X_4)$$

Satisfying assignment:

$X_1 \mapsto 1 \qquad X_2 \mapsto 0 \qquad X_3 \mapsto 1 \qquad X_4 \mapsto 0 \qquad X_5 \mapsto 1$

In association with propositional formulae, the following two problems are the most important:

SAT

Input:	Propositional formula φ in CNF
Problem:	Is φ satisfiable?

k-**SAT**

Input:	Propositional formula φ in k-CNF
Problem:	Is φ satisfiable?

2-SAT

Lemma. 2-SAT is in PTIME.

Lemma. 2-SAT is in PTIME.

Proof. The following algorithm solves the problem in poly time.

Main: Input Γ in CNF

```
bcp(Γ)
if conflict return UNSAT
while Γ ≠ ∅ do
    choose var. X from Γ
    set Γ' := Γ
    assign(Γ, X, 1)
    bcp(Γ)
    if conflict
        Γ := Γ'
        assign(Γ, X, 0)
        bcp(Γ)
        if conflict
            return UNSAT
```

bcp(Γ) (boolean constraint propagation)

```
while Γ contains unit-clause C do
    if C = {X}     assign(Γ, X, 1)
    if C = {¬X}    assign(Γ, X, 0)
od
if Γ contains empty clause return conflict
```

assign(Γ, X, c)

```
if c = 1 do
    remove from Γ all clauses C with X ∈ C
    remove ¬X from all remaining clauses
if c = 0 do
    remove from Γ all clauses C with ¬X ∈ C
    remove X from all remaining clauses
```

Polynomial-Time Reductions

As for decidability we can use many-one reductions to show membership in PTIME.

Definition. A language $\mathcal{L}_1 \subseteq \Sigma^*$ is polynomially reducible to $\mathcal{L}_2 \subseteq \Sigma^*$, denoted $\mathcal{L}_1 \leq_p \mathcal{L}_2$, if there is a polynomial-time computable function f such that for all $w \in \Sigma^*$

$$w \in \mathcal{L}_1 \quad \iff \quad f(w) \in \mathcal{L}_2.$$

Polynomial-Time Reductions

As for decidability we can use many-one reductions to show membership in PTIME.

Definition. A language $\mathcal{L}_1 \subseteq \Sigma^*$ is polynomially reducible to $\mathcal{L}_2 \subseteq \Sigma^*$, denoted $\mathcal{L}_1 \leq_p \mathcal{L}_2$, if there is a polynomial-time computable function f such that for all $w \in \Sigma^*$

$$w \in \mathcal{L}_1 \qquad \iff \qquad f(w) \in \mathcal{L}_2.$$

Lemma.

If $\mathcal{L}_1 \leq_p \mathcal{L}_2$ and $\mathcal{L}_2 \in \text{PTIME}$ then $\mathcal{L}_1 \in \text{PTIME}$.

Proof. The sum and composition of polynomials is a polynomial.

Reductions in PTIME

All non-trivial members of PTIME can be reduced to each other

Lemma. If $\mathcal{B}$ is any language in PTIME, $\mathcal{B} \neq \emptyset$, $\mathcal{B} \neq \Sigma^*$, then $\mathcal{A} \leq_p \mathcal{B}$ for any $\mathcal{A} \in$ PTIME.

Proof. Choose $w \in \mathcal{B}$ and $w' \notin \mathcal{B}$

Define the function f by setting

$$\begin{array}{lcll} f(x) & := & w & x \in \mathcal{A} \\ f(x) & := & w' & x \notin \mathcal{A} \end{array}$$

Since $\mathcal{A} \in$ PTIME, f is computable in polynomial time, and is a reduction from $\mathcal{A}$ to $\mathcal{B}$.

Example: Colourability

Vertex Colouring:

A vertex colouring of G with k colours is a function

$$c : V(G) \longrightarrow \{1, \ldots, k\}$$

such that adjacent nodes have different colours

i.e. $\{u, v\} \in E(G)$ implies $c(u) \neq c(v)$

k-**COLOURABILITY**	
Input:	Graph G, $k \in \mathbb{N}$
Problem:	Does G have a vertex colouring with k colours?

For $k = 2$ this is the same as BIPARTITE.

A reduction to 2-SAT

Lemma. 2-COLOURABILITY $\leq_p$ 2-SAT

Proof. We define a reduction as follows: Given graph G

- For each vertex $v \in V(G)$ of the graph introduce variable X_v
- For each $\{u, v\} \in E(G)$ add clauses $(X_u \vee X_v)$ and $(\neg X_u \vee \neg X_v)$

This is obviously computable in polynomial time.

We check that it is a reduction:

- If G is 2-colourable, use colouring to assign truth values. (One colour is *true*, the other *false*)
- If the formula is satisfiable, the truth assignment defines valid 2-colouring.

  For every edge $\{u, v\} \in E(G)$, one variable X_u, X_v must be set to *true*, the other to *false*.

A reduction to 2-SAT

Lemma. 2-COLOURABILITY $\leq_p$ 2-SAT

Proof. We define a reduction as follows: Given graph G

- For each vertex $v \in V(G)$ of the graph introduce variable X_v
- For each $\{u, v\} \in E(G)$ add clauses $(X_u \vee X_v)$ and $(\neg X_u \vee \neg X_v)$

This is obviously computable in polynomial time.

We check that it is a reduction:

- If G is 2-colourable, use colouring to assign truth values. (One colour is *true*, the other *false*)
- If the formula is satisfiable, the truth assignment defines valid 2-colouring.
  For every edge $\{u, v\} \in E(G)$, one variable X_u, X_v must be set to *true*, the other to *false*.

Corollary. 2-COLOURABILITY $\in$ PTIME

A reduction to 3-SAT

Lemma. k-COLOURABILITY $\leq_p$ 3-SAT

Proof. I will do this on board (going via k-SAT).

Reducible to 2-SAT ??